

Data structures

Contents

1	Introduction	1
1.1	Data structure	1
1.1.1	Overview	1
1.1.2	Examples	1
1.1.3	Language support	2
1.1.4	See also	2
1.1.5	References	2
1.1.6	Further reading	2
1.1.7	External links	3
1.2	Linked data structure	3
1.2.1	Common types of linked data structures	3
1.2.2	Advantages and disadvantages	4
1.2.3	See also	4
1.2.4	References	4
1.3	Succinct data structure	4
1.3.1	Succinct dictionaries	5
1.3.2	Examples	5
1.3.3	References	6
1.4	Implicit data structure	6
1.4.1	Efficiency concerns	6
1.4.2	Weighted element	6
1.4.3	Examples	6
1.4.4	Further reading	6
1.5	Compressed data structure	7
1.5.1	References	7
1.6	Search data structure	7
1.6.1	Classification	7
1.6.2	Footnotes	8
1.6.3	See also	8
1.7	Persistent data structure	8
1.7.1	Partially persistent	8
1.7.2	Fully persistent	9

1.7.3	Confluently persistent	9
1.7.4	Examples of persistent data structures	9
1.7.5	Reference cycles	12
1.7.6	See also	12
1.7.7	References	12
1.7.8	Further reading	12
1.7.9	External links	12
1.8	Concurrent data structure	12
1.8.1	Basic principles	12
1.8.2	Design and Implementation	13
1.8.3	See also	13
1.8.4	References	13
1.8.5	Further reading	13
1.8.6	External links	14
2	Abstract data types	15
2.1	Abstract data type	15
2.1.1	Examples	15
2.1.2	Introduction	16
2.1.3	Defining an abstract data type	16
2.1.4	Advantages of abstract data typing	18
2.1.5	Typical operations	18
2.1.6	Examples	19
2.1.7	Implementation	19
2.1.8	See also	20
2.1.9	Notes	20
2.1.10	References	20
2.1.11	Further	21
2.1.12	External links	21
2.2	List	21
2.2.1	Operations	21
2.2.2	Implementations	21
2.2.3	Programming language support	22
2.2.4	Applications	22
2.2.5	Abstract definition	22
2.2.6	References	23
2.2.7	See also	23
2.3	Stack	23
2.3.1	History	23
2.3.2	Non-essential operations	23
2.3.3	Software stacks	23
2.3.4	Hardware stacks	24

2.3.5	Applications	26
2.3.6	Security	27
2.3.7	See also	27
2.3.8	References	27
2.3.9	Further reading	27
2.3.10	External links	27
2.4	Queue	28
2.4.1	Queue implementation	28
2.4.2	See also	29
2.4.3	References	29
2.4.4	External links	29
2.5	Deque	29
2.5.1	Naming conventions	29
2.5.2	Distinctions and sub-types	29
2.5.3	Operations	30
2.5.4	Implementations	30
2.5.5	Language support	30
2.5.6	Complexity	30
2.5.7	Applications	30
2.5.8	See also	31
2.5.9	References	31
2.5.10	External links	31
2.6	Priority queue	31
2.6.1	Operations	31
2.6.2	Similarity to queues	32
2.6.3	Implementation	32
2.6.4	Equivalence of priority queues and sorting algorithms	32
2.6.5	Libraries	33
2.6.6	Applications	33
2.6.7	See also	34
2.6.8	References	34
2.6.9	Further reading	34
2.6.10	External links	34
2.7	Map	35
2.7.1	Operations	35
2.7.2	Example	35
2.7.3	Implementation	36
2.7.4	Language support	36
2.7.5	Permanent storage	36
2.7.6	See also	37
2.7.7	References	37

2.7.8	External links	37
2.8	Bidirectional map	37
2.8.1	External links	37
2.9	Multimap	37
2.9.1	Examples	37
2.9.2	Language support	38
2.9.3	See also	38
2.9.4	References	38
2.10	Set	38
2.10.1	Type theory	39
2.10.2	Operations	39
2.10.3	Implementations	40
2.10.4	Language support	40
2.10.5	Multiset	41
2.10.6	See also	42
2.10.7	Notes	42
2.10.8	References	42
2.11	Tree	43
2.11.1	Definition	43
2.11.2	Terminologies used in Trees	43
2.11.3	Terminology	45
2.11.4	Drawing Trees	45
2.11.5	Representations	45
2.11.6	Generalizations	45
2.11.7	Traversal methods	46
2.11.8	Common operations	46
2.11.9	Common uses	46
2.11.10	See also	46
2.11.11	Notes	47
2.11.12	References	47
2.11.13	External links	47
3	Arrays	48
3.1	Array data structure	48
3.1.1	History	48
3.1.2	Applications	48
3.1.3	Element identifier and addressing formulas	49
3.1.4	Efficiency	50
3.1.5	Dimension	51
3.1.6	See also	51
3.1.7	References	52
3.1.8	External links	52

3.2	Row-major order	52
3.2.1	Explanation and example	52
3.2.2	Programming Languages	52
3.2.3	Transposition	53
3.2.4	Address calculation in general	53
3.2.5	See also	53
3.2.6	References	53
3.3	Dope vector	53
3.3.1	See also	53
3.3.2	References	53
3.4	Ilfie vector	54
3.4.1	See also	54
3.4.2	References	54
3.5	Dynamic array	54
3.5.1	Bounded-size dynamic arrays and capacity	54
3.5.2	Geometric expansion and amortized cost	55
3.5.3	Performance	55
3.5.4	Variants	55
3.5.5	Language support	56
3.5.6	References	56
3.5.7	External links	56
3.6	Hashed array tree	56
3.6.1	Definitions	56
3.6.2	Expansions and size reductions	57
3.6.3	Related data structures	57
3.6.4	References	57
3.7	Gap buffer	57
3.7.1	Example	58
3.7.2	See also	58
3.7.3	References	58
3.7.4	External links	58
3.8	Circular buffer	58
3.8.1	Uses	58
3.8.2	How it works	59
3.8.3	Circular buffer mechanics	59
3.8.4	Difficulties	60
3.8.5	Variants	62
3.8.6	External links	62
3.9	Sparse array	62
3.9.1	Representation	62
3.9.2	Sparse Array as Linked List	62

3.9.3	See also	63
3.9.4	External links	63
3.10	Bit array	63
3.10.1	Definition	63
3.10.2	Basic operations	63
3.10.3	More complex operations	63
3.10.4	Compression	64
3.10.5	Advantages and disadvantages	64
3.10.6	Applications	64
3.10.7	Language support	65
3.10.8	See also	66
3.10.9	References	66
3.10.10	External links	66
3.11	Bitboard	66
3.11.1	Short description	66
3.11.2	History	66
3.11.3	Description for all games or applications	67
3.11.4	General technical advantages and disadvantages	67
3.11.5	Chess bitboards	67
3.11.6	Other bitboards	68
3.11.7	See also	68
3.11.8	External links	68
3.12	Parallel array	69
3.12.1	Pros and cons	69
3.12.2	See also	70
3.12.3	References	70
3.13	Lookup table	70
3.13.1	History	70
3.13.2	Examples	71
3.13.3	Other usage of lookup tables	73
3.13.4	See also	73
3.13.5	References	73
3.13.6	External links	74
4	Lists	75
4.1	Linked list	75
4.1.1	Advantages	75
4.1.2	Disadvantages	75
4.1.3	History	75
4.1.4	Basic concepts and nomenclature	76
4.1.5	Tradeoffs	77
4.1.6	Linked list operations	79

4.1.7	Linked lists using arrays of node	80
4.1.8	Language support	81
4.1.9	Internal and external storage	81
4.1.10	Related data structures	83
4.1.11	Notes	83
4.1.12	Footnotes	83
4.1.13	References	83
4.1.14	External links	84
4.2	XOR linked list	84
4.2.1	Description	84
4.2.2	Features	85
4.2.3	Drawbacks	85
4.2.4	Variations	85
4.2.5	See also	86
4.2.6	References	86
4.2.7	External links	86
4.3	Unrolled linked list	86
4.3.1	Overview	86
4.3.2	Performance	86
4.3.3	See also	87
4.3.4	References	87
4.3.5	External links	87
4.4	VList	87
4.4.1	Operations	87
4.4.2	Advantages and Disadvantages	87
4.4.3	Structure	88
4.4.4	Variants	88
4.4.5	See also	88
4.4.6	References	88
4.4.7	External links	88
4.5	Skip list	88
4.5.1	Description	89
4.5.2	History	90
4.5.3	Usages	90
4.5.4	See also	91
4.5.5	References	91
4.5.6	External links	91
4.6	Self-organizing list	92
4.6.1	History	92
4.6.2	Introduction	92
4.6.3	Implementation of a self-organizing list	92

4.6.4	Analysis of Running Times for Access/ Search in a List	92
4.6.5	Techniques for Rearranging Nodes	93
4.6.6	Applications of self-organizing lists	94
4.6.7	References	94
5	Binary trees	96
5.1	Binary tree	96
5.1.1	Definitions	96
5.1.2	Types of binary trees	97
5.1.3	Properties of binary trees	98
5.1.4	Combinatorics	98
5.1.5	Methods for storing binary trees	98
5.1.6	Encodings	99
5.1.7	Common operations	100
5.1.8	See also	101
5.1.9	References	101
5.1.10	External links	102
5.2	Binary search tree	102
5.2.1	Definition	102
5.2.2	Operations	103
5.2.3	Types	106
5.2.4	See also	106
5.2.5	References	106
5.2.6	Further reading	107
5.2.7	External links	107
5.3	Self-balancing binary search tree	107
5.3.1	Overview	107
5.3.2	Implementations	108
5.3.3	Applications	108
5.3.4	See also	109
5.3.5	References	109
5.3.6	External links	109
5.4	Tree rotation	109
5.4.1	Illustration	109
5.4.2	Detailed illustration	110
5.4.3	Inorder Invariance	110
5.4.4	Rotations for rebalancing	110
5.4.5	Rotation distance	110
5.4.6	See also	111
5.4.7	References	111
5.4.8	External links	111
5.5	Weight-balanced tree	111

5.5.1	Description	111
5.5.2	References	112
5.6	Threaded binary tree	112
5.6.1	Motivation	112
5.6.2	Definition	113
5.6.3	Types of threaded binary trees	113
5.6.4	The array of Inorder traversal	113
5.6.5	Example	113
5.6.6	Null link	114
5.6.7	Non recursive Inorder traversal for a Threaded Binary Tree	114
5.6.8	References	114
5.6.9	External links	115
5.7	AVL tree	115
5.7.1	Operations	115
5.7.2	Comparison to other structures	117
5.7.3	See also	117
5.7.4	References	118
5.7.5	Further reading	118
5.7.6	External links	118
5.8	Red-black tree	118
5.8.1	History	118
5.8.2	Terminology	119
5.8.3	Properties	119
5.8.4	Analogy to B-trees of order 4	119
5.8.5	Applications and related data structures	120
5.8.6	Operations	121
5.8.7	Proof of asymptotic bounds	123
5.8.8	Parallel algorithms	124
5.8.9	See also	124
5.8.10	Notes	124
5.8.11	References	125
5.8.12	External links	125
5.9	AA tree	125
5.9.1	Balancing rotations	125
5.9.2	Insertion	126
5.9.3	Deletion	126
5.9.4	Performance	127
5.9.5	See also	127
5.9.6	References	127
5.9.7	External links	127
5.10	Scapegoat tree	127

5.10.1 Theory	127
5.10.2 Operations	128
5.10.3 See also	129
5.10.4 References	129
5.10.5 External links	129
5.11 Splay tree	129
5.11.1 Advantages	129
5.11.2 Disadvantages	130
5.11.3 Operations	130
5.11.4 Implementation and variants	131
5.11.5 Analysis	132
5.11.6 Performance theorems	133
5.11.7 Dynamic optimality conjecture	133
5.11.8 Variants	134
5.11.9 See also	134
5.11.10 Notes	134
5.11.11 References	134
5.11.12 External links	135
5.12 T-tree	135
5.12.1 Performance	135
5.12.2 Node structures	135
5.12.3 Algorithms	135
5.12.4 Notes	136
5.12.5 See also	136
5.12.6 References	137
5.12.7 External links	137
5.13 Rope	137
5.13.1 Description	137
5.13.2 Operations	137
5.13.3 Comparison with monolithic arrays	139
5.13.4 See also	139
5.13.5 References	139
5.13.6 External links	139
5.14 Top Trees	139
5.14.1 Glossary	139
5.14.2 Introduction	140
5.14.3 Dynamic Operations	141
5.14.4 Internal Operations	141
5.14.5 Non local search	141
5.14.6 Interesting Results and Applications	142
5.14.7 Implementation	142

5.14.8 References	143
5.14.9 External links	143
5.15 Tango tree	143
5.15.1 Structure	143
5.15.2 Algorithm	144
5.15.3 Analysis	144
5.15.4 See also	145
5.15.5 References	145
5.16 Van Emde Boas tree	145
5.16.1 Supported operations	145
5.16.2 How it works	145
5.16.3 References	147
5.17 Cartesian tree	147
5.17.1 Definition	147
5.17.2 Range searching and lowest common ancestors	148
5.17.3 Treaps	149
5.17.4 Efficient construction	149
5.17.5 Application in sorting	149
5.17.6 History	150
5.17.7 Notes	150
5.17.8 References	150
5.18 Treap	151
5.18.1 Description	151
5.18.2 Operations	152
5.18.3 Randomized binary search tree	152
5.18.4 Comparison	153
5.18.5 See also	153
5.18.6 References	153
5.18.7 External links	153
6 B-trees	154
6.1 B-tree	154
6.1.1 Overview	154
6.1.2 The database problem	155
6.1.3 Technical description	156
6.1.4 Best case and worst case heights	157
6.1.5 Algorithms	157
6.1.6 In filesystems	160
6.1.7 Variations	160
6.1.8 See also	161
6.1.9 Notes	161
6.1.10 References	161

6.1.11	External links	161
6.2	B+ tree	162
6.2.1	Overview	162
6.2.2	Algorithms	162
6.2.3	Characteristics	163
6.2.4	Implementation	163
6.2.5	History	164
6.2.6	See also	164
6.2.7	References	164
6.2.8	External links	164
6.3	Dancing tree	165
6.3.1	References	165
6.3.2	External links	165
6.4	2-3 tree	165
6.4.1	Definitions	165
6.4.2	Properties	166
6.4.3	Operations	166
6.4.4	See also	166
6.4.5	References	166
6.4.6	External links	166
6.5	2-3-4 tree	166
6.5.1	Properties	167
6.5.2	Insertion	167
6.5.3	Deletion	167
6.5.4	See also	168
6.5.5	References	168
6.5.6	External links	168
6.6	Queaps	168
6.6.1	Description	169
6.6.2	Operations	169
6.6.3	Analysis	170
6.6.4	Code example	170
6.6.5	See also	170
6.6.6	References	170
6.7	Fusion tree	170
6.7.1	How it works	171
6.7.2	Sketching	171
6.7.3	Approximating the sketch	171
6.7.4	Parallel comparison	171
6.7.5	Desketching	172
6.7.6	References	172

6.8	Bx-tree	172
6.8.1	Index structure	172
6.8.2	Utilizing the B+ tree for moving objects	173
6.8.3	Insertion, update and deletion	173
6.8.4	Queries	173
6.8.5	Adapting relational database engines to accommodate moving objects	174
6.8.6	Performance tuning	174
6.8.7	See also	174
6.8.8	References	175
7	Heaps	176
7.1	Heap	176
7.1.1	Operations	176
7.1.2	Implementation	177
7.1.3	Variants	177
7.1.4	Comparison of theoretic bounds for variants	178
7.1.5	Applications	178
7.1.6	Implementations	178
7.1.7	See also	178
7.1.8	References	179
7.1.9	External links	179
7.2	Binary heap	179
7.2.1	Heap operations	180
7.2.2	Building a heap	181
7.2.3	Heap implementation	181
7.2.4	Derivation of index equations	182
7.2.5	See also	183
7.2.6	Notes	183
7.2.7	References	183
7.2.8	External links	184
7.3	Binomial heap	184
7.3.1	Binomial heap	184
7.3.2	Structure of a binomial heap	184
7.3.3	Implementation	185
7.3.4	Summary of running times	186
7.3.5	Applications	186
7.3.6	See also	186
7.3.7	References	186
7.3.8	External links	187
7.4	Fibonacci heap	187
7.4.1	Structure	187
7.4.2	Implementation of operations	188

7.4.3	Proof of degree bounds	189
7.4.4	Worst case	189
7.4.5	Summary of running times	190
7.4.6	Practical considerations	190
7.4.7	References	190
7.4.8	External links	190
7.5	2-3 heap	190
7.5.1	References	190
7.6	Pairing heap	191
7.6.1	Structure	191
7.6.2	Operations	191
7.6.3	Summary of running times	192
7.6.4	References	192
7.6.5	External links	192
7.7	Beap	193
7.7.1	Performance	193
7.7.2	References	193
7.8	Leftist tree	193
7.8.1	Bias	193
7.8.2	S-value	193
7.8.3	Merging height biased leftist trees	194
7.8.4	Initializing a height biased leftist tree	194
7.8.5	References	195
7.8.6	External links	195
7.8.7	Further reading	195
7.9	Skew heap	195
7.9.1	Definition	195
7.9.2	Operations	195
7.9.3	References	197
7.9.4	External links	197
7.10	Soft heap	197
7.10.1	Applications	197
7.10.2	References	198
7.11	d-ary heap	198
7.11.1	Data structure	198
7.11.2	Analysis	199
7.11.3	Applications	199
7.11.4	References	199
7.11.5	External links	200
8	Tries	201
8.1	Trie	201

8.1.1	Applications	201
8.1.2	Implementation strategies	202
8.1.3	See also	204
8.1.4	Notes	204
8.1.5	References	204
8.1.6	External links	205
8.2	Radix tree	205
8.2.1	Applications	205
8.2.2	Operations	205
8.2.3	History	206
8.2.4	Comparison to other data structures	206
8.2.5	Variants	207
8.2.6	See also	207
8.2.7	References	207
8.2.8	External links	207
8.3	Suffix tree	208
8.3.1	History	208
8.3.2	Definition	208
8.3.3	Generalized suffix tree	209
8.3.4	Functionality	209
8.3.5	Applications	209
8.3.6	Implementation	210
8.3.7	External construction	210
8.3.8	See also	210
8.3.9	Notes	210
8.3.10	References	211
8.3.11	External links	211
8.4	Suffix array	212
8.4.1	Definition	212
8.4.2	Example	212
8.4.3	Correspondence to suffix trees	212
8.4.4	Space Efficiency	212
8.4.5	Construction Algorithms	213
8.4.6	Applications	213
8.4.7	Notes	214
8.4.8	References	214
8.4.9	External links	215
8.5	Compressed suffix array	215
8.5.1	Open Source Implementations	215
8.5.2	See also	215
8.5.3	References	215

8.5.4	External links	216
8.6	FM-index	216
8.6.1	Background	216
8.6.2	FM-index data structure	216
8.6.3	Count	216
8.6.4	Locate	217
8.6.5	Applications	217
8.6.6	See also	217
8.6.7	References	217
8.7	Generalized suffix tree	217
8.7.1	Functionality	217
8.7.2	Example	217
8.7.3	Alternatives	218
8.7.4	References	218
8.7.5	External links	218
8.8	B-trie	218
8.8.1	References	218
8.9	Judy array	218
8.9.1	Terminology	218
8.9.2	Benefits	218
8.9.3	Drawbacks	219
8.9.4	References	219
8.9.5	External links	219
8.10	Ctrie	219
8.10.1	Operation	219
8.10.2	Advantages of Ctries	220
8.10.3	Problems with Ctries	220
8.10.4	Implementations	220
8.10.5	History	221
8.10.6	References	221
8.11	Directed acyclic word graph	221
8.11.1	Comparison to tries	222
8.11.2	References	222
8.11.3	External links	222
9	Multiway trees	223
9.1	Ternary search tree	223
9.1.1	Description	223
9.1.2	Ternary search tree operations	223
9.1.3	Comparison to other data structures	224
9.1.4	Uses	224
9.1.5	See also	224

9.1.6	References	224
9.1.7	External links	224
9.2	And-or tree	224
9.2.1	Example	224
9.2.2	Definitions	224
9.2.3	Search strategies	225
9.2.4	Relationship with logic programming	225
9.2.5	Relationship with two-player games	225
9.2.6	Bibliography	225
9.3	(a,b)-tree	225
9.3.1	Definition	225
9.3.2	Inner node representation	225
9.3.3	See also	225
9.3.4	References	225
9.4	Link/cut tree	226
9.4.1	Structure	226
9.4.2	Operations	226
9.4.3	Analysis	228
9.4.4	Application	228
9.4.5	See also	228
9.4.6	Further reading	228
9.5	SPQR tree	229
9.5.1	Structure	229
9.5.2	Construction	230
9.5.3	Usage	230
9.5.4	See also	231
9.5.5	Notes	231
9.5.6	References	231
9.5.7	External links	231
9.6	Spaghetti stack	231
9.6.1	Use as call stacks	231
9.6.2	See also	232
9.6.3	References	232
9.7	Disjoint-set data structure	232
9.7.1	Disjoint-set linked lists	233
9.7.2	Disjoint-set forests	233
9.7.3	Applications	234
9.7.4	History	234
9.7.5	See also	234
9.7.6	References	234
9.7.7	External links	235

10 Space-partitioning trees	236
10.1 Space partitioning	236
10.1.1 Overview	236
10.1.2 Use in computer graphics	236
10.1.3 Other uses	236
10.1.4 Types of space partitioning data structures	236
10.1.5 References	236
10.2 Binary space partitioning	237
10.2.1 Overview	237
10.2.2 Generation	237
10.2.3 Traversal	238
10.2.4 Brushes	239
10.2.5 Timeline	239
10.2.6 References	240
10.2.7 Additional references	240
10.2.8 External links	240
10.3 Segment tree	240
10.3.1 Structure description	241
10.3.2 Storage requirements	241
10.3.3 Construction	242
10.3.4 Query	242
10.3.5 Generalization for higher dimensions	242
10.3.6 Notes	243
10.3.7 History	243
10.3.8 References	243
10.3.9 Sources cited	243
10.4 Interval tree	243
10.4.1 Naive approach	243
10.4.2 Centered interval tree	244
10.4.3 Augmented tree	245
10.4.4 Medial- or length-oriented tree	247
10.4.5 References	247
10.4.6 External links	247
10.5 Range tree	247
10.5.1 Description	248
10.5.2 Operations	248
10.5.3 See also	248
10.5.4 References	249
10.5.5 External links	249
10.6 Bin	249
10.6.1 Operations	249

10.6.2 Efficiency and tuning	250
10.6.3 Compared to other range query data structures	250
10.6.4 See also	250
10.7 k-d tree	250
10.7.1 Informal description	250
10.7.2 Operations on <i>k</i> -d trees	250
10.7.3 High-dimensional data	253
10.7.4 Complexity	253
10.7.5 Variations	253
10.7.6 See also	253
10.7.7 References	254
10.7.8 External links	254
10.8 Implicit k-d tree	255
10.8.1 Nomenclature and references	255
10.8.2 Construction	255
10.8.3 Applications	256
10.8.4 Complexity	256
10.8.5 See also	256
10.8.6 References	256
10.9 min/max kd-tree	256
10.9.1 Construction	257
10.9.2 Properties	257
10.9.3 Applications	257
10.9.4 See also	257
10.9.5 References	257
10.10 Adaptive k-d tree	257
10.10.1 References	257
10.11 Quadtree	257
10.11.1 Types	258
10.11.2 Some common uses of quadtrees	259
10.11.3 Pseudo code	259
10.11.4 See also	259
10.11.5 References	260
10.11.6 External links	260
10.12 Octree	260
10.12.1 For spatial representation	260
10.12.2 History	261
10.12.3 Common uses	261
10.12.4 Application to color quantization	261
10.12.5 Implementation for point decomposition	261
10.12.6 Example color quantization	261

10.12.7 See also	262
10.12.8 References	262
10.12.9 External links	262
10.13 Linear octrees	262
10.14 Z-order curve	263
10.14.1 Coordinate values	263
10.14.2 Efficiently building quadtrees	264
10.14.3 Use with one-dimensional data structures for range searching	264
10.14.4 Related structures	265
10.14.5 Applications in linear algebra	265
10.14.6 See also	265
10.14.7 References	265
10.14.8 External links	266
10.15 UB-tree	266
10.15.1 References	266
10.15.2 External links	266
10.16 R-tree	266
10.16.1 R-tree idea	267
10.16.2 Variants	267
10.16.3 Algorithm	267
10.16.4 See also	269
10.16.5 References	269
10.16.6 External links	269
10.17 R+ tree	269
10.17.1 Difference between R+ trees and R trees	270
10.17.2 Advantages	270
10.17.3 Disadvantages	270
10.17.4 Notes	270
10.17.5 References	270
10.18 R* tree	270
10.18.1 Difference between R*-trees and R-trees	270
10.18.2 Performance	271
10.18.3 Algorithm and complexity	271
10.18.4 References	271
10.18.5 External links	271
10.19 Hilbert R-tree	271
10.19.1 The basic idea	272
10.19.2 Packed Hilbert R-trees	272
10.19.3 Dynamic Hilbert R-trees	273
10.19.4 Notes	276
10.19.5 References	276

10.20 X-tree	276
10.20.1 References	276
10.21 Metric tree	276
10.21.1 Multidimensional search	277
10.21.2 Metric data structures	277
10.21.3 References	277
10.22 Vp-tree	277
10.22.1 Understanding a VP tree	277
10.22.2 Searching through a VP tree	278
10.22.3 Advantages of a VP tree	278
10.22.4 Implementation examples	278
10.22.5 References	278
10.22.6 External links	278
10.22.7 Further reading	278
10.23 BK-tree	278
10.23.1 See also	278
10.23.2 References	278
10.23.3 External links	279
11 Hashes	280
11.1 Hash table	280
11.1.1 Hashing	280
11.1.2 Key statistics	281
11.1.3 Collision resolution	281
11.1.4 Dynamic resizing	284
11.1.5 Performance analysis	285
11.1.6 Features	286
11.1.7 Uses	286
11.1.8 Implementations	287
11.1.9 History	288
11.1.10 See also	288
11.1.11 References	288
11.1.12 Further reading	289
11.1.13 External links	289
11.2 Hash function	289
11.2.1 Uses	290
11.2.2 Properties	291
11.2.3 Hash function algorithms	292
11.2.4 Locality-sensitive hashing	295
11.2.5 Origins of the term	295
11.2.6 List of hash functions	295
11.2.7 See also	296

11.2.8 References	296
11.2.9 External links	296
11.3 Open addressing	296
11.3.1 Example pseudo code	297
11.3.2 See also	297
11.3.3 References	298
11.4 Lazy deletion	298
11.4.1 References	298
11.5 Linear probing	298
11.5.1 Algorithm	298
11.5.2 Properties	298
11.5.3 Dictionary operation in constant time	298
11.5.4 See also	298
11.5.5 References	298
11.5.6 External links	299
11.6 Quadratic probing	299
11.6.1 Quadratic function	299
11.6.2 Quadratic probing insertion	299
11.6.3 Quadratic probing search	300
11.6.4 Limitations	300
11.6.5 See also	300
11.6.6 References	300
11.6.7 External links	301
11.7 Double hashing	301
11.7.1 Classical applied data structure	301
11.7.2 Implementation details for caching	301
11.7.3 See also	301
11.7.4 Notes	301
11.7.5 External links	302
11.8 Cuckoo hashing	302
11.8.1 History	302
11.8.2 Theory	302
11.8.3 Example	303
11.8.4 Generalizations and applications	303
11.8.5 See also	303
11.8.6 References	303
11.8.7 External links	304
11.9 Coalesced hashing	304
11.9.1 Performance	305
11.9.2 References	305
11.10 Perfect hash function	305

11.10.1 Properties and uses	305
11.10.2 Minimal perfect hash function	306
11.10.3 See also	306
11.10.4 References	306
11.10.5 Further reading	306
11.10.6 External links	306
11.11 Universal hashing	307
11.11.1 Introduction	307
11.11.2 Mathematical guarantees	307
11.11.3 Constructions	308
11.11.4 See also	310
11.11.5 References	310
11.11.6 Further reading	311
11.11.7 External links	311
11.12 Linear hashing	311
11.12.1 Algorithm Details	311
11.12.2 Adoption in language systems	312
11.12.3 Adoption in database systems	312
11.12.4 References	312
11.12.5 External links	312
11.12.6 See also	312
11.13 Extendible hashing	312
11.13.1 Example	312
11.13.2 Example implementation	314
11.13.3 Notes	314
11.13.4 See also	315
11.13.5 References	315
11.13.6 External links	315
11.14 2-choice hashing	315
11.14.1 How It Works	315
11.14.2 Implementation	315
11.14.3 Performance	315
11.14.4 See also	315
11.14.5 References	315
11.14.6 Further reading	315
11.15 Pearson hashing	316
11.15.1 C implementation to generate 64-bit (16 hex chars) hash	316
11.15.2 References	317
11.16 Fowler–Noll–Vo hash function	317
11.16.1 Overview	317
11.16.2 The hash	317

11.16.3 Non-cryptographic hash	318
11.16.4 See also	318
11.16.5 Notes	318
11.16.6 External links	318
11.17 Bitstate hashing	318
11.17.1 Use	318
11.17.2 References	319
11.18 Bloom filter	319
11.18.1 Algorithm description	319
11.18.2 Space and time advantages	320
11.18.3 Probability of false positives	320
11.18.4 Approximating the number of items in a Bloom filter	322
11.18.5 The union and intersection of sets	322
11.18.6 Interesting properties	322
11.18.7 Examples	322
11.18.8 Alternatives	323
11.18.9 Extensions and applications	323
11.18.10 See also	325
11.18.11 Notes	325
11.18.12 References	326
11.18.13 External links	328
11.19 Locality preserving hashing	328
11.19.1 External links	328
11.20 Zobrist hashing	328
11.20.1 Calculation of the hash value	328
11.20.2 Use of the hash value	328
11.20.3 Updating the hash value	329
11.20.4 Wider usage	329
11.20.5 See also	329
11.20.6 References	329
11.21 Rolling hash	329
11.21.1 Rabin-Karp rolling hash	329
11.21.2 Content based slicing using Rabin-Karp hash	330
11.21.3 Cyclic polynomial	330
11.21.4 Computational complexity	330
11.21.5 Software	330
11.21.6 See also	330
11.21.7 External links	330
11.21.8 Footnotes	330
11.22 Hash list	330
11.22.1 Root hash	331

11.22.2 See also	331
11.23 Hash tree	331
11.24 Prefix hash tree	331
11.24.1 External links	331
11.24.2 See also	331
11.25 Hash trie	332
11.25.1 See also	332
11.25.2 References	332
11.26 Hash array mapped trie	332
11.26.1 Operation	332
11.26.2 Advantages of HAMTs	332
11.26.3 Implementation details	332
11.26.4 Implementations	332
11.26.5 References	332
11.27 Distributed hash table	333
11.27.1 History	333
11.27.2 Properties	333
11.27.3 Structure	334
11.27.4 DHT implementations	335
11.27.5 Examples	335
11.27.6 See also	335
11.27.7 References	335
11.27.8 External links	336
11.28 Consistent hashing	336
11.28.1 History	336
11.28.2 Need for consistent hashing	336
11.28.3 Technique	337
11.28.4 Monotonic keys	337
11.28.5 Properties	337
11.28.6 Examples of use	337
11.28.7 References	337
11.28.8 External links	338
11.29 Stable hashing	338
11.29.1 See also	338
11.30 Koorde	338
11.30.1 De Bruijn's graphs	338
11.30.2 Routing example	339
11.30.3 Non-constant degree Koorde	339
11.30.4 References	339
12 Graphs	340
12.1 Graph	340

12.1.1	Algorithms	340
12.1.2	Operations	340
12.1.3	Representations	340
12.1.4	See also	341
12.1.5	References	341
12.1.6	External links	341
12.2	Adjacency list	341
12.2.1	Implementation details	341
12.2.2	Operations	342
12.2.3	Trade-offs	342
12.2.4	Data structures	342
12.2.5	References	342
12.2.6	Further reading	342
12.2.7	External links	343
12.3	Adjacency matrix	343
12.3.1	Examples	343
12.3.2	Adjacency matrix of a bipartite graph	343
12.3.3	Properties	343
12.3.4	Variations	344
12.3.5	Data structures	344
12.3.6	References	344
12.3.7	Further reading	344
12.3.8	External links	344
12.4	And-inverter graph	344
12.4.1	Implementations	345
12.4.2	References	345
12.4.3	See also	345
12.5	Binary decision diagram	346
12.5.1	Definition	346
12.5.2	History	346
12.5.3	Applications	346
12.5.4	Variable ordering	347
12.5.5	Logical operations on BDDs	347
12.5.6	See also	347
12.5.7	References	347
12.5.8	Further reading	348
12.5.9	External links	348
12.6	Binary moment diagram	348
12.6.1	Pointwise and linear decomposition	348
12.6.2	Edge weights	348
12.6.3	References	349

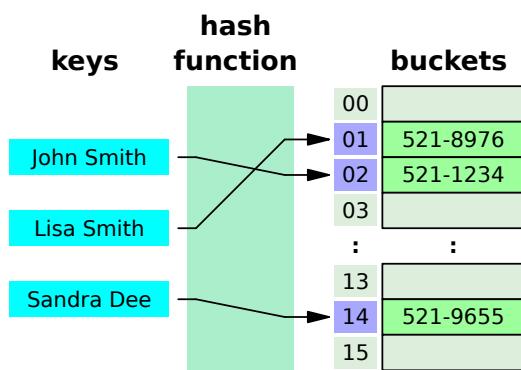
12.7	Zero-suppressed decision diagram	349
12.7.1	Available packages	349
12.7.2	References	349
12.7.3	External links	349
12.8	Propositional directed acyclic graph	350
12.8.1	PDAG, BDD, and NNF	350
12.8.2	See also	350
12.8.3	References	350
12.9	Graph-structured stack	350
12.9.1	References	351
12.10	Scene graph	351
12.10.1	Scene graphs in graphics editing tools	351
12.10.2	Scene graphs in games and 3D applications	351
12.10.3	Scene graph implementation	351
12.10.4	Scene graphs and bounding volume hierarchies (BVHs)	352
12.10.5	Scene graphs and spatial partitioning	353
12.10.6	Standards	353
12.10.7	See also	353
12.10.8	References	353
12.10.9	External links	354
13	Appendix	355
13.1	Big O notation	355
13.1.1	Formal definition	355
13.1.2	Example	356
13.1.3	Usage	356
13.1.4	Properties	357
13.1.5	Multiple variables	357
13.1.6	Matters of notation	358
13.1.7	Orders of common functions	359
13.1.8	Related asymptotic notations	359
13.1.9	Generalizations and related usages	361
13.1.10	History (Bachmann–Landau, Hardy, and Vinogradov notations)	361
13.1.11	See also	362
13.1.12	References and Notes	362
13.1.13	Further reading	362
13.1.14	External links	363
13.2	Amortized analysis	363
13.2.1	History	363
13.2.2	Method	363
13.2.3	Examples	363
13.2.4	Common use	364

13.2.5 References	364
13.3 Locality of reference	364
13.3.1 Types of locality	365
13.3.2 Reasons for locality	365
13.3.3 General locality usage	365
13.3.4 Spatial and temporal locality usage	366
13.3.5 See also	367
13.3.6 Bibliography	367
13.3.7 References	367
13.4 Standard Template Library	367
13.4.1 Composition	367
13.4.2 History	368
13.4.3 Criticisms	369
13.4.4 Implementations	370
13.4.5 See also	370
13.4.6 Notes	370
13.4.7 References	370
13.4.8 External links	370
14 Text and image sources, contributors, and licenses	371
14.1 Text	371
14.2 Images	388
14.3 Content license	398

Chapter 1

Introduction

1.1 Data structure



A *hash table*

In computer science, a **data structure** is a particular way of organizing data in a computer so that it can be used **efficiently**.^{[1][2]} Data structures can implement one or more particular **abstract data types**, which are the means of specifying the contract of operations and their **complexity**. In comparison, a data structure is a concrete implementation of the contract provided by an **ADT**.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, databases use **B-tree** indexes for small percentages of data retrieval and **compilers** and databases use dynamic **hash tables** as look up tables.

Data structures provide a means to manage large amounts of data efficiently for uses such as large **databases** and **internet indexing services**. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and **programming languages** emphasize data structures, rather than algorithms, as the key organizing factor in software design. Storing and retrieving can be carried out on data stored in both **main memory** and in **secondary memory**.

1.1.1 Overview

Data structures are generally based on the ability of a computer to fetch and store data at any place in its mem-

ory, specified by a pointer – a bit string, representing a **memory address**, that can be itself stored in memory and manipulated by the program. Thus, the **array** and **record** data structures are based on computing the addresses of data items with arithmetic operations; while the **linked** data structures are based on storing addresses of data items within the structure itself. Many data structures use both principles, sometimes combined in non-trivial ways (as in **XOR linking**).

The implementation of a data structure usually requires writing a set of **procedures** that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an **abstract data type**, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).

1.1.2 Examples

Main article: [List of data structures](#)

There are numerous types of data structures, generally built upon simpler **primitive data types**:

- An **array** is a number of elements in a specific order, typically all of the same type. Elements are accessed using an integer index to specify which element is required (although the elements may be of almost any type). Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable.
- A **record** (also called a *tuple* or *struct*) is an aggregate data structure. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called **fields** or **members**.
- An **associative array** (also called a *dictionary* or *map*) is a more flexible variation on an array, in which **name-value pairs** can be added and deleted

freely. A hash table is a common implementation of an associative array.

- A *union* type specifies which of a number of permitted primitive types may be stored in its instances, e.g. *float* or *long integer*. Contrast with a *record*, which could be defined to contain a *float* and an *integer*; whereas in a *union*, there is only one value at a time. Enough space is allocated to contain the widest member datatype.
- A *tagged union* (also called a *variant*, *variant record*, *discriminated union*, or *disjoint union*) contains an additional field indicating its current type, for enhanced type safety.
- A *set* is an abstract data structure that can store specific values, in no particular order and with no duplicate values.
- *Graphs* and *trees* are linked abstract data structures composed of *nodes*. Each node contains a value and one or more pointers to other nodes arranged in a hierarchy. Graphs can be used to represent networks, while variants of trees can be used for *sorting* and *searching*, having their nodes arranged in some relative order based on their values.
- An *object* contains data fields, like a record, as well as various *methods* which operate on the contents of the record. In the context of *object-oriented programming*, records are known as plain old data structures to distinguish them from objects.

1.1.3 Language support

Most assembly languages and some low-level languages, such as **BCPL** (Basic Combined Programming Language), lack built-in support for data structures. On the other hand, many **high-level programming languages** and some higher-level assembly languages, such as **MASM**, have special syntax or other built-in support for certain data structures, such as records and arrays. For example, the **C** and **Pascal** languages support **structs** and **records**, respectively, in addition to vectors (one-dimensional arrays) and multi-dimensional arrays.^{[3][4]}

Most programming languages feature some sort of **library** mechanism that allows data structure implementations to be reused by different programs. Modern languages usually come with standard libraries that implement the most common data structures. Examples are the **C++ Standard Template Library**, the **Java Collections Framework**, and Microsoft's **.NET Framework**.

Modern languages also generally support **modular programming**, the separation between the **interface** of a library module and its implementation. Some provide **opaque data types** that allow clients to hide implementation details. Object-oriented programming languages,

such as **C++**, **Java** and **Smalltalk** may use **classes** for this purpose.

Many known data structures have **concurrent** versions that allow multiple computing threads to access the data structure simultaneously.

1.1.4 See also

- Abstract data type
- Concurrent data structure
- Data model
- Dynamization
- Linked data structure
- List of data structures
- Persistent data structure
- Plain old data structure

1.1.5 References

- [1] Paul E. Black (ed.), entry for *data structure* in *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology. 15 December 2004. Online version Accessed May 21, 2009.
- [2] Entry *data structure* in the *Encyclopædia Britannica* (2009) Online entry accessed on May 21, 2009.
- [3] “The GNU C Manual”. Free Software Foundation. Retrieved 15 October 2014.
- [4] “Free Pascal: Reference Guide”. Free Pascal. Retrieved 15 October 2014.

1.1.6 Further reading

- Peter Brass, *Advanced Data Structures*, Cambridge University Press, 2008.
- Donald Knuth, *The Art of Computer Programming*, vol. 1. Addison-Wesley, 3rd edition, 1997.
- Dinesh Mehta and Sartaj Sahni *Handbook of Data Structures and Applications*, Chapman and Hall/CRC Press, 2007.
- Niklaus Wirth, *Algorithms and Data Structures*, Prentice Hall, 1985.

1.1.7 External links

- course on data structures
- Data structures Programs Examples in c,java
- UC Berkeley video course on data structures
- Descriptions from the Dictionary of Algorithms and Data Structures
- Data structures course
- An Examination of Data Structures from .NET perspective
- Schaffer, C. *Data Structures and Algorithm Analysis*

1.2 Linked data structure

In computer science, a **linked data structure** is a **data structure** which consists of a set of **data records (nodes)** linked together and organized by **references (links or pointers)**. The link between data can also be called a **connector**.

In linked data structures, the links are usually treated as special **data types** that can only be dereferenced or compared for equality. Linked data structures are thus contrasted with **arrays** and other data structures that require performing arithmetic operations on pointers. This distinction holds even when the nodes are actually implemented as elements of a single array, and the references are actually array indices: as long as no arithmetic is done on those indices, the data structure is essentially a linked one.

Linking can be done in two ways - Using dynamic allocation and using array index linking.

Linked data structures include **linked lists**, **search trees**, **expression trees**, and many other widely used data structures. They are also key building blocks for many efficient algorithms, such as **topological sort**^[1] and **set union-find**.^[2]

1.2.1 Common types of linked data structures

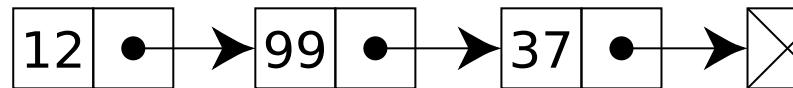
Linked lists

A linked list is a collection of structures ordered not by their physical placement in memory but by logical links that are stored as part of the data in the structure itself. It is not necessary that it should be stored in the adjacent memory locations. Every structure has a data field and an address field. The Address field contains the address of its successor.

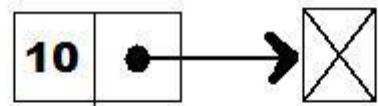
Linked list can be singly, doubly or multiply linked and can either be linear or circular.

Basic Properties

- Objects, called **nodes**, are linked in a linear sequence
- A reference to the first node of the list is always kept. This is called the 'head' or 'front'.^[3]



A linked list with three nodes contain two fields each: an integer value and a link to the next node



A linked list with a single node.

Example in Java This is an example of the node class used to store integers in a Java implementation of a linked list.

```
public class IntNode { public int value; public IntNode link; public IntNode(int v) { value = v; } }
```

Example in C This is an example of the node structure used for implementation of linked list in C.

```
struct node { int val; struct node *next; };
```

This is an example using **typedefs**.

```
typedef struct node_s node_t; struct node_s { int val; node_t *next; };
```

Note: A structure like this which contains a member that points to the same structure is called a **self-referential structure**.

Example in C++ This is an example of the node class structure used for implementation of linked list in C++.

```
class Node { int val Node *next; };
```

Search trees

A search tree is a tree data structure in whose nodes data values can be stored from some **ordered** set, which is such that in an in-order traversal of the tree the nodes are visited in ascending order of the stored values.

Basic Properties

- Objects, called nodes, are stored in an ordered set.
- **In-order traversal** provides an ascending readout of the data in the tree
- Sub trees of the tree are in themselves, trees.

1.2.2 Advantages and disadvantages

Linked list versus arrays

Compared to arrays, linked data structures allow more flexibility in organizing the data and in allocating space for it. In arrays, the size of the array must be specified precisely at the beginning, which can be a potential waste of memory. A linked data structure is built dynamically and never needs to be bigger than the programmer requires. It also requires no guessing in terms of how much space must be allocated when using a linked data structure. This is a feature that is key in saving wasted memory.

In an array, the array elements have to be in a **contiguous** (connected and sequential) portion of memory. But in a linked data structure, the reference to each node gives users the information needed to find the next one. The nodes of a linked data structure can also be moved individually to different locations without affecting the logical connections between them, unlike arrays. With due care, a **process** can add or delete nodes to one part of a data structure even while other processes are working on other parts.

On the other hand, access to any particular node in a linked data structure requires following a chain of references that stored in it. If the structure has n nodes, and each node contains at most b links, there will be some nodes that cannot be reached in less than $\log b n$ steps. For many structures, some nodes may require **worst case** up to $n-1$ steps. In contrast, many array data structures allow access to any element with a constant number of operations, independent of the number of entries.

Broadly the implementation of these linked data structure is through **dynamic data structures**. It gives us the chance to use particular space again. Memory can be utilized more efficiently by using this data structures. Memory is allocated as per the need and when memory is not further needed, deallocation is done.

General disadvantages

Linked data structures may also incur in substantial memory allocation overhead (if nodes are allocated individually) and frustrate **memory paging** and **processor caching** algorithms (since they generally have poor **locality of reference**). In some cases, linked data structures may also use more memory (for the link fields) than competing array structures. This is because linked data structures are not contiguous. Instances of data can be found all over in memory, unlike arrays.

In arrays, n th element can be accessed immediately, while in a linked data structure we have to follow multiple pointers so element access time varies according to where in the structure the element is.

In some **theoretical models of computation** that enforce the constraints of linked structures, such as the **pointer machine**, many problems require more steps than in the unconstrained **random access machine** model.

1.2.3 See also

- List of data structures

1.2.4 References

- [1] Donald Knuth, *The Art of Computer Programming*
- [2] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, Volume 7, Issue 5 (May 1964), pages 301-303. The paper originating disjoint-set forests. *ACM Digital Library*
- [3] <http://www.cs.toronto.edu/~{}hojjat/148s07/lectures/week5/07linked.pdf>

1.3 Succinct data structure

In computer science, a **succinct data structure** is a data structure which uses an amount of space that is “close” to the **information-theoretic** lower bound, but (unlike other compressed representations) still allows for efficient query operations. The concept was originally introduced by Jacobson ^[1] to encode **bit vectors**, (unlabeled) **trees**, and **planar graphs**. Unlike general **lossless data compression** algorithms, succinct data structures retain the ability to use them in-place, without decompressing them first. A related notion is that of a **compressed data structure**, in which the size of the data structure depends upon the particular data being represented.

Suppose that Z is the information-theoretical optimal number of bits needed to store some data. A representation of this data is called

- **implicit** if it takes $Z + O(1)$ bits of space,

- *succinct* if it takes $Z + o(Z)$ bits of space, and
- *compact* if it takes $O(Z)$ bits of space.

Implicit structures are thus usually reduced to storing information using some permutation of the input data; the most well-known example of this is the **heap**.

1.3.1 Succinct dictionaries

Succinct indexable dictionaries, also called *rank/select* dictionaries, form the basis of a number of succinct representation techniques, including **binary trees**, k -ary trees and **multisets**,^[2] as well as **suffix trees** and **arrays**.^[3] The basic problem is to store a subset S of a universe $U = [0 \dots n] = \{0, 1, \dots, n - 1\}$, usually represented as a bit array $B[0 \dots n]$ where $B[i] = 1$ iff $i \in S$. An indexable dictionary supports the usual methods on dictionaries (queries, and insertions/deletions in the dynamic case) as well as the following operations:

- **rank** _{q} (x) = $|\{k \in [0 \dots x] : B[k] = q\}|$
- **select** _{q} (x) = $\min\{k \in [0 \dots n] : \text{rank}_q(k) = x\}$

for $q \in \{0, 1\}$.

In other words, **rank** _{q} (x) returns the number of elements equal to q up to position x while **select** _{q} (x) returns the position of the x -th occurrence of q .

There is a simple representation^[4] which uses $n + o(n)$ bits of storage space (the original bit array and an $o(n)$ auxiliary structure) and supports **rank** and **select** in constant time. It uses an idea similar to that for **range-minimum queries**; there are a constant number of recursions before stopping at a subproblem of a limited size. The bit array B is partitioned into *large blocks* of size $l = \lg^2 n$ bits and *small blocks* of size $s = \lg n/2$ bits. For each large block, the rank of its first bit is stored in a separate table $R_l[0 \dots n/l]$; each such entry takes $\lg n$ bits for a total of $(n/l) \lg n = n/\lg n$ bits of storage. Within a large block, another directory $R_s[0 \dots l/s]$ stores the rank of each of the $l/s = 2 \lg n$ small blocks it contains. The difference here is that it only needs $\lg l = \lg \lg^2 n = 2 \lg \lg n$ bits for each entry, since only the differences from the rank of the first bit in the containing large block need to be stored. Thus, this table takes a total of $(n/s) \lg l = 4n \lg \lg n/\lg n$ bits. A lookup table R_p can then be used that stores the answer to every possible rank query on a bit string of length s for $i \in [0, s)$; this requires $2^s s \lg s = O(\sqrt{n} \lg n \lg \lg n)$ bits of storage space. Thus, since each of these auxiliary tables take $o(n)$ space, this data structure supports rank queries in $O(1)$ time and $n + o(n)$ bits of space.

To answer a query for **rank**₁(x) in constant time, a constant time algorithm computes

$$\text{rank}_1(x) = R_l[\lfloor x/l \rfloor] + R_s[\lfloor x/s \rfloor] + R_p[x \bmod s]$$

In practice, the lookup table R_p can be replaced by bit-wise operations and smaller tables to perform find the number of bits set in the small blocks. This is often beneficial, since succinct data structures find their uses in large data sets, in which case cache misses become much more frequent and the chances of the lookup table being evicted from closer CPU caches becomes higher.^[5] Select queries can be easily supported by doing a binary search on the same auxiliary structure used for **rank**; however, this takes $O(\lg n)$ time in the worst case. A more complicated structure using $3n/\lg \lg n + O(\sqrt{n} \lg n \lg \lg n) = o(n)$ bits of additional storage can be used to support **select** in constant time.^[6] In practice, many of these solutions have hidden constants in the $O(\cdot)$ notation which dominate before any asymptotic advantage becomes apparent; implementations using broadword operations and word-aligned blocks often perform better in practice.^[7]

Entropy-compressed dictionaries

The $n + o(n)$ space approach can be improved by noting that there are $\binom{n}{m}$ distinct m -subsets of $[n]$ (or binary strings of length n with exactly m 1's), and thus $\mathcal{B}(m, n) = \lceil \lg \binom{n}{m} \rceil$ is an information theoretic lower bound on the number of bits needed to store B . There is a succinct (static) dictionary which attains this bound, namely using $\mathcal{B}(m, n) + o(\mathcal{B}(m, n))$ space.^[8] This structure can be extended to support **rank** and **select** queries and takes $\mathcal{B}(m, n) + O(m + n \lg \lg n / \lg n)$ space.^[2] This bound can be reduced to a space/time tradeoff by reducing the storage space of the dictionary to $\mathcal{B}(m, n) + O(nt^t / \lg^t n + n^{3/4})$ with queries taking $O(t)$ time.^[9]

1.3.2 Examples

A **null-terminated string** (C string) takes $Z + 1$ space, and is thus implicit. A string with an arbitrary length (Pascal string) takes $Z + \log(Z)$ space, and is thus succinct. If there is a maximum length – which is the case in practice, since $2^{32} = 4$ GiB of data is a very long string, and $2^{64} = 16$ EiB of data is larger than any string in practice – then a string with a length is also implicit, taking $Z + k$ space, where k is the number of data to represent the maximum length (e.g., bytes in a word).

When a sequence of variable-length items (such as strings) needs to be encoded, there are various possibilities. A direct approach is to store a length and an item in each record – these can then be placed one after another. This allows efficient next, but not finding the k th item. An alternative is to place the items in order with a delimiter (e.g., **null-terminated string**). This use a delimiter instead of a length, and is substantially slower, since the entire sequence must be scanned for delimiters. Both of these are space-efficient. An alternative approach is out-of-band separation: the items can simply be placed one after another, with no delimiters. Item bounds can

then be stored as a sequence of length, or better, offsets into this sequence. Alternatively, a separate binary string consisting of 1s in the positions where an item begins, and 0s everywhere else is encoded along with it. Given this string, the *select* function can quickly determine where each item begins, given its index.^[10] This is *compact* but not *succinct*, as it takes $2Z$ space, which is $O(Z)$.

Another example is the representation of a **binary tree**: an arbitrary binary tree on n nodes can be represented in $2n + o(n)$ bits while supporting a variety of operations on any node, which includes finding its parent, its left and right child, and returning the size of its subtree, each in constant time. The number of different binary trees on n nodes is $\binom{2n}{n} / (n + 1)$. For large n , this is about 4^n ; thus we need at least about $\log_2(4^n) = 2n$ bits to encode it. A succinct binary tree therefore would occupy only 2 bits per node.

1.3.3 References

- [1] Jacobson, G. J (1988). “Succinct static data structures”.
- [2] Raman, R.; V. Raman; S. S Rao (2002). “Succinct indexable dictionaries with applications to encoding k-ary trees and multisets” (PDF). *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. pp. 233–242. ISBN 0-89871-513-X.
- [3] Sadakane, K.; R. Grossi (2006). “Squeezing succinct data structures into entropy bounds” (PDF). *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. pp. 1230–1239. ISBN 0-89871-605-5.
- [4] Jacobson, G. (1989). “Space-efficient static trees and graphs” (PDF).
- [5] González, R.; S. Grabowski; V. Mäkinen; G. Navarro (2005). “Practical implementation of rank and select queries” (PDF). *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*. pp. 27–38.
- [6] Clark, D. (1998). “Compact pat trees” (PDF).
- [7] Vigna, S. (2008). “Broadword implementation of rank/select queries” (PDF). *Experimental Algorithms*. Lecture Notes in Computer Science **5038**: 154–168. doi:10.1007/978-3-540-68552-4_12. ISBN 978-3-540-68548-7.
- [8] Brodnik, A.; J. I Munro (1999). “Membership in constant time and almost-minimum space” (PDF). *SIAM J. Comput.* **28** (5): 1627–1640. doi:10.1137/S0097539795294165.
- [9] Pătrașcu, M. (2008). “Succincter” (PDF). *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*. pp. 305–313.
- [10] Belazzougui, Djamel. “Hash, displace, and compress” (PDF).

1.4 Implicit data structure

In computer science, an **implicit data structure** stores very little information other than the main or required data. These storage schemes retain no **pointers**, represent a file of n k -key records as an n by k array. In implicit data structures, the only structural information is to allow the array to grow and shrink. It is called “implicit” because the order of the elements carries meaning. Another term used interchangeably is **space efficient**. Definitions of “very little” are vague and can mean from $O(1)$ to $O(\log n)$ extra space. Everything is accessed in-place, by reading bits at various positions in the data. To achieve memory-optimal coding, appropriate data items use bits instead of bytes. Implicit data structures are also **succinct** data structures.

1.4.1 Efficiency concerns

Implicit data structures are designed to improve **main memory** utilization, concomitantly reducing access to slower storage. A greater fraction of data in an implicit data structure can fit in main memory, reducing administrative processing. Implicit data structures can improve cache-efficiency and thus running speed, especially if the method used improves **locality of reference**.

1.4.2 Weighted element

For presentation of elements with different weights, several data structures are required. The structure uses one more location besides those required for element values. The first structure supports **worst case** search time in terms of rank of weight of elements with respect to set of weights. If the elements are drawn from uniform distribution, then variation of this structure will take average time. The same result obtains for the data structures in which the intervals between consecutive values have access probabilities.

1.4.3 Examples

Examples of implicit data structures include:

- Ahnentafel
- Binary heap
- Beap
- Null-terminated string

1.4.4 Further reading

- See publications of Hervé Brönnimann, J. Ian Munro, Greg Frederickson

1.5 Compressed data structure

The term **compressed data structure** arises in the computer science subfields of algorithms, data structures, and theoretical computer science. It refers to a data structure whose operations are roughly as fast as those of a conventional data structure for the problem, but whose size can be substantially smaller. The size of the compressed data structure is typically highly dependent upon the entropy of the data being represented.

Important examples of compressed data structures include the **compressed suffix array**^{[1][2]} and the **FM-index**,^[3] both of which can represent an arbitrary text of characters T for **pattern matching**. Given any input pattern P , they support the operation of finding if and where P appears in T . The search time is proportional to the sum of the length of pattern P , a very slow-growing function of the length of the text T , and the number of reported matches. The space they occupy is roughly equal to the size of the text T in entropy-compressed form, such as that obtained by **Prediction by Partial Matching** or **gzip**. Moreover, both data structures are **self-indexing**, in that they can reconstruct the text T in a random access manner, and thus the underlying text T can be discarded. In other words, they simultaneously provide a compressed and quickly searchable representation of the text T . They represent a substantial space improvement over the conventional **suffix tree** and **suffix array**, which occupy many times more space than the size of T . They also support searching for arbitrary patterns, as opposed to the **inverted index**, which can support only word-based searches. In addition, inverted indexes do not have the self-indexing feature.

An important related notion is that of a **succinct data structure**, which uses space roughly equal to the information-theoretic minimum, which is a worst-case notion of the space needed to represent the data. In contrast, the size of a compressed data structure depends upon the particular data being represented. When the data are compressible, as is often the case in practice for natural language text, the compressed data structure can occupy space very close to the information-theoretic minimum, and significantly less space than most compression schemes.

1.5.1 References

- [1] R. Grossi and J. S. Vitter, Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching], *Proceedings of the 32nd ACM Symposium on Theory of Computing*, May 2000, 397-406. Journal version in *SIAM Journal on Computing*, 35(2), 2005, 378-407.
- [2] R. Grossi, A. Gupta, and J. S. Vitter, High-Order Entropy-Compressed Text Indexes, *Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms*, January 2003, 841-850.

- [3] P. Ferragina and G. Manzini, Opportunistic Data Structures with Applications, *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*, November 2000, 390-398. Journal version in Indexing Compressed Text, *Journal of the ACM*, 52(4), 2005, 552-581.

1.6 Search data structure

In computer science, a **search data structure** is any data structure that allows the efficient retrieval of specific items from a set of items, such as a specific record from a database.

The simplest, most general, and least efficient search structure is merely an unordered sequential list of all the items. Locating the desired item in such a list, by the **linear search** method, inevitably requires a number of operations proportional to the number n of items, in the worst case as well as in the average case. Useful search data structures allow faster retrieval; however, they are limited to queries of some specific kind. Moreover, since the cost of building such structures is at least proportional to n , they only pay off if several queries are to be performed on the same database (or on a database that changes little between queries).

Static search structures are designed for answering many queries on a fixed database; **dynamic** structures also allow insertion, deletion, or modification of items between successive queries. In the dynamic case, one must also consider the cost of fixing the search structure to account for the changes in the database.

1.6.1 Classification

The simplest kind of query is to locate a record that has a specific field (the *key*) equal to a specified value v . Other common kinds of query are “find the item with smallest (or largest) key value”, “find the item with largest key value not exceeding v ”, “find all items with key values between specified bounds v_{\min} and v_{\max} ”.

In certain databases the key values may be points in some multi-dimensional space. For example, the key may be a geographic position (**latitude** and **longitude**) on the Earth. In that case, common kinds of queries are *find the record with a key closest to a given point v* , or *“find all items whose key lies at a given distance from v ”*, or *“find all items within a specified region R of the space”*.

A common special case of the latter are simultaneous range queries on two or more simple keys, such as “find all employee records with salary between 50,000 and 100,000 and hired between 1995 and 2007”.

Single ordered keys

- **Array** if the key values span a moderately compact

interval.

- Priority-sorted list; see [linear search](#)
- Key-sorted array; see [binary search](#)
- Self-balancing binary search tree
- Hash table

Finding the smallest element

- [Heap](#)

Asymptotic amortized worst-case analysis In this table, the [asymptotic notation \$O\(f\(n\)\)\$](#) means “not exceeding some fixed multiple of $f(n)$ in the worst case.”

This table is only an approximate summary; for each data structure there are special situations and variants that may lead to different costs. Also two or more data structures can be combined to obtain lower costs.

1.6.2 Footnotes

1.6.3 See also

- [List of data structures](#)

1.7 Persistent data structure

Not to be confused with [persistent storage](#).

In computing, a **persistent data structure** is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively **immutable**, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure.

A data structure is partially persistent if all versions can be accessed but only the newest version can be modified. The data structure is fully persistent if every version can be both accessed and modified. If there is also a meld or merge operation that can create a new version from two previous versions, the data structure is called confluently persistent. Structures that are not persistent are called [ephemeral](#).^[1]

These types of data structures are particularly common in [logical](#) and [functional](#) programming, and in a [purely functional](#) program all data is immutable, so all data structures are automatically fully persistent.^[1] Persistent data structures can also be created using in-place updating of data and these may, in general, use less time or storage space than their purely functional counterparts.

While persistence can be achieved by simple copying, this is inefficient in CPU and RAM usage, because most operations make only small changes to a data structure. A better method is to exploit the similarity between the new and old versions to share structure between them, such as using the same subtree in a number of [tree structures](#). However, because it rapidly becomes infeasible to determine how many previous versions share which parts of the structure, and because it is often desirable to discard old versions, this necessitates an environment with [garbage collection](#).

1.7.1 Partially persistent

In the partial persistence model, we may query any previous version of the data structure, but we may only update the latest version. This implies a [linear ordering](#) among the versions.

Three methods on balanced binary search tree:

Fat Node

Fat node method is to record all changes made to node fields in the nodes themselves, without erasing old values of the fields. This requires that we allow nodes to become arbitrarily “fat”. In other words, each fat node contains the same information and [pointer](#) fields as an ephemeral node, along with space for an arbitrary number of extra field values. Each extra field value has an associated field name and a version stamp which indicates the version in which the named field was changed to have the specified value. Besides, each fat node has its own version stamp, indicating the version in which the node was created. The only purpose of nodes having version stamps is to make sure that each node only contains one value per field name per version. In order to navigate through the structure, each original field value in a node has a version stamp of zero.

Complexity of Fat Node With using fat node method, it requires $O(1)$ space for every modification: just store the new data. Each modification takes $O(1)$ additional time to store the modification at the end of the modification history. This is an [amortized time bound](#), assuming we store the modification history in a growable [array](#). For [access time](#), we must find the right version at each node as we traverse the structure. If we made m modifications, then each access operation has $O(\log m)$ slowdown resulting from the cost of finding the nearest modification in the array.

Path Copying

Path copy is to make a copy of all nodes on the path which contains the node we are about to insert or delete. Then

we must **cascade** the change back through the data structure: all nodes that pointed to the old node must be modified to point to the new node instead. These modifications cause more cascading changes, and so on, until we reach to the root. We maintain an array of roots indexed by timestamp. The data structure pointed to by time t 's root is exactly time t 's data structure.

Complexity of Path Copying With m modifications, this costs $O(\log m)$ additive **lookup** time. Modification time and space are bounded by the size of the structure, since a single modification may cause the entire structure to be copied. That is $O(m)$ for one update, and thus $O(n^2)$ preprocessing time.

A combination

Sleator, Tarjan et al. came up with a way to combine the advantages of fat nodes and path copying, getting $O(1)$ access slowdown and $O(1)$ modification space and time.

In each node, we store one modification box. This box can hold one modification to the node—either a modification to one of the pointers, or to the node's key, or to some other piece of node-specific data—and a timestamp for when that modification was applied. Initially, every node's modification box is empty.

Whenever we access a node, we check the modification box, and compare its timestamp against the access time. (The access time specifies the version of the data structure that we care about.) If the modification box is empty, or the access time is before the modification time, then we ignore the modification box and just deal with the normal part of the node. On the other hand, if the access time is after the modification time, then we use the value in the modification box, overriding that value in the node. (Say the modification box has a new left pointer. Then we'll use it instead of the normal left pointer, but we'll still use the normal right pointer.)

Modifying a node works like this. (We assume that each modification touches one pointer or similar field.) If the node's modification box is empty, then we fill it with the modification. Otherwise, the modification box is full. We make a copy of the node, but using only the latest values. (That is, we overwrite one of the node's fields with the value that was stored in the modification box.) Then we perform the modification directly on the new node, without using the modification box. (We overwrite one of the new node's fields, and its modification box stays empty.) Finally, we cascade this change to the node's parent, just like path copying. (This may involve filling the parent's modification box, or making a copy of the parent recursively. If the node has no parent—it's the root—we add the new root to a **sorted array** of roots.)

With this algorithm, given any time t , at most one modification box exists in the data structure with time t . Thus,

a modification at time t splits the tree into three parts: one part contains the data from before time t , one part contains the data from after time t , and one part was unaffected by the modification.

Complexity of the combination Time and space for modifications require amortized analysis. A modification takes $O(1)$ amortized space, and $O(1)$ amortized time. To see why, use a **potential function** ϕ , where $\phi(T)$ is the number of full live nodes in T . The live nodes of T are just the nodes that are reachable from the current root at the current time (that is, after the last modification). The full live nodes are the live nodes whose modification boxes are full.

Each modification involves some number of copies, say k , followed by 1 change to a modification box. (Well, not quite—you could add a new root—but that doesn't change the argument.) Consider each of the k copies. Each costs $O(1)$ space and time, but decreases the potential function by one. (First, the node we copy must be full and live, so it contributes to the potential function. The potential function will only drop, however, if the old node isn't reachable in the new tree. But we know it isn't reachable in the new tree—the next step in the algorithm will be to modify the node's parent to point at the copy. Finally, we know the copy's modification box is empty. Thus, we've replaced a full live node with an empty live node, and ϕ goes down by one.) The final step fills a modification box, which costs $O(1)$ time and increases ϕ by one.

Putting it all together, the change in ϕ is $\Delta\phi = 1 - k$. Thus, we've paid $O(k + \Delta\phi) = O(1)$ space and $O(k + \Delta\phi + 1) = O(1)$ time.

1.7.2 Fully persistent

In fully persistent model, both updates and queries are allowed on any version of the data structure.

1.7.3 Confluently persistent

In confluently persistent model, we use combinator to combine input of more than one previous version to output a new single version. Rather than a branching tree, combinations of versions induce a **DAG** (directed acyclic graph) structure on the version graph.

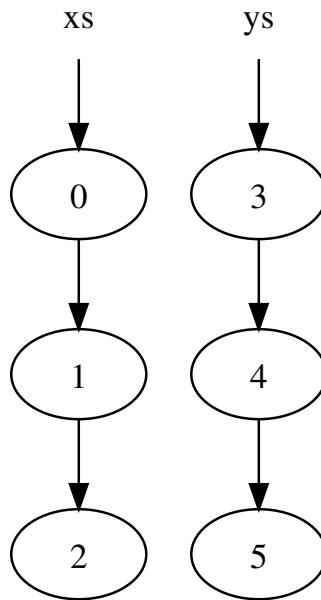
1.7.4 Examples of persistent data structures

Perhaps the simplest persistent data structure is the **singly linked list** or **cons-based list**, a simple list of objects formed by each carrying a reference to the next in the list. This is persistent because we can take a *tail* of the

list, meaning the last k items for some k , and add new nodes on to the front of it. The tail will not be duplicated, instead becoming shared between both the old list and the new list. So long as the contents of the tail are immutable, this sharing will be invisible to the program.

Many common reference-based data structures, such as red-black trees,^[2] stacks,^[3] and treaps,^[4] can easily be adapted to create a persistent version. Some others need slightly more effort, for example: queues, deques, and extensions including min-deques (which have an additional $O(1)$ operation *min* returning the minimal element) and random access deques (which have an additional operation of random access with sub-linear, most often logarithmic, complexity).

There also exist persistent data structures which use destructible operations, making them impossible to implement efficiently in purely functional languages (like Haskell outside specialized monads like state or IO), but possible in languages like C or Java. These types of data structures can often be avoided with a different design. One primary advantage to using purely persistent data structures is that they often behave better in multi-threaded environments.



where a circle indicates a node in the list (the arrow out representing the second element of the node which is a pointer to another node).

Now concatenating the two lists:

`zs = xs ++ ys`

results in the following memory structure:

Linked lists

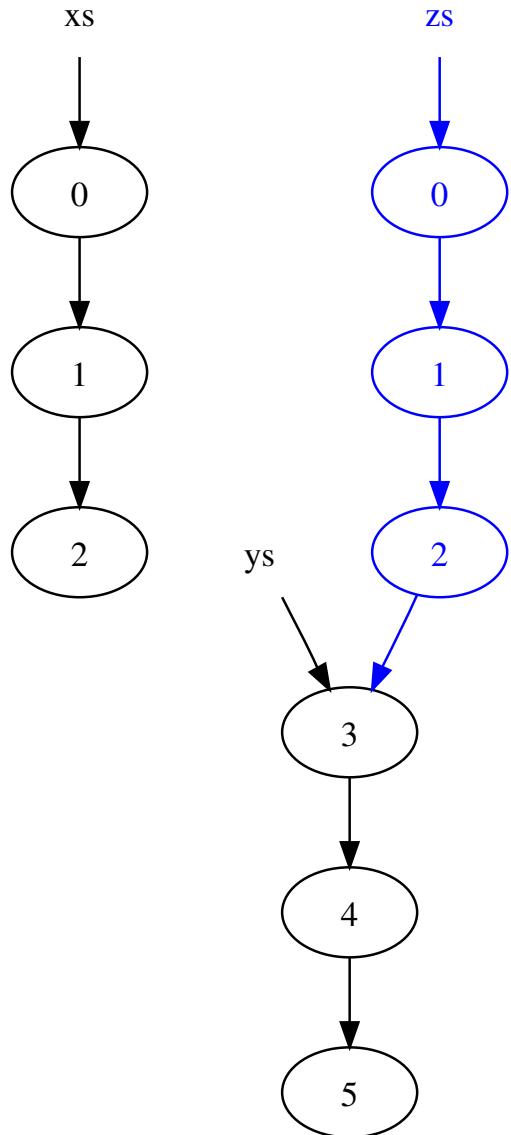
This example is taken from Okasaki. See the bibliography.

Singly linked lists are the bread-and-butter data structure in functional languages. In ML-derived languages and Haskell, they are purely functional because once a node in the list has been allocated, it cannot be modified, only copied or destroyed. Note that ML itself is **not** purely functional.

Consider the two lists:

`xs = [0, 1, 2] ys = [3, 4, 5]`

These would be represented in memory by:



Notice that the nodes in list `xs` have been copied, but the nodes in `ys` are shared. As a result, the original lists (`xs` and `ys`) persist and have not been modified.

The reason for the copy is that the last node in `xs` (the node containing the original value 2) cannot be modified to point to the start of `ys`, because that would change the value of `xs`.

Trees

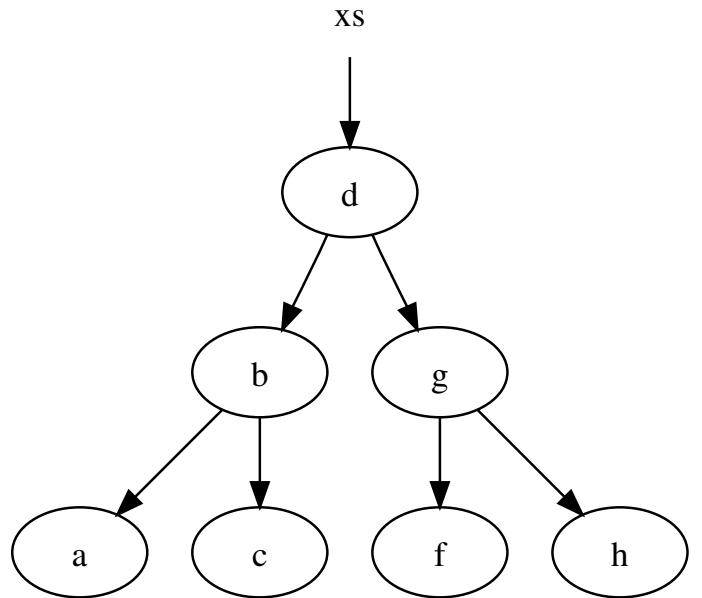
This example is taken from Okasaki. See the bibliography.

Consider a binary tree used for fast searching, where every node has the recursive invariant that subnodes on the left are less than the node, and subnodes on the right are greater than the node.

For instance, the set of data

`xs = [a, b, c, d, f, g, h]`

might be represented by the following binary search tree:



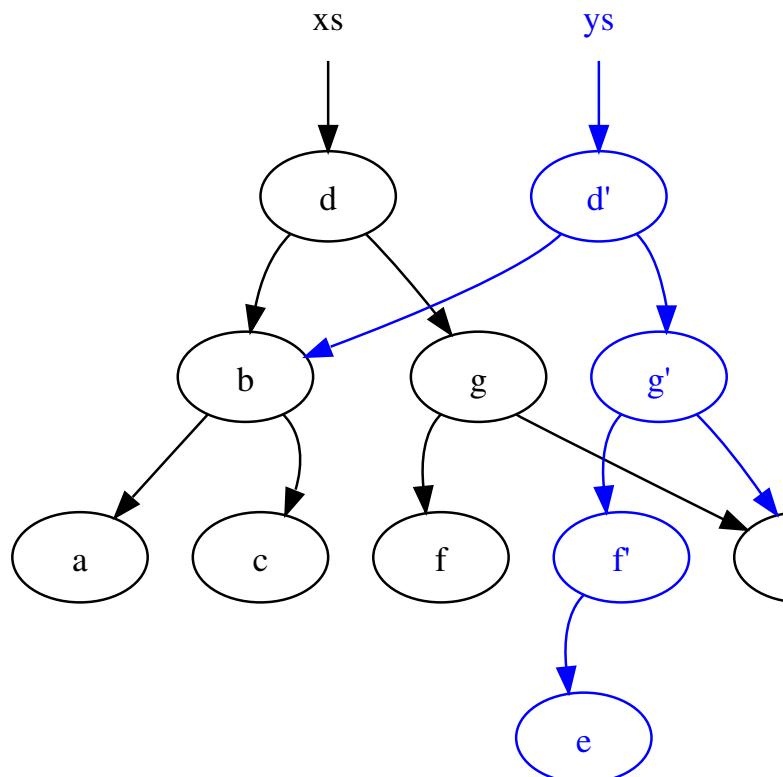
A function which inserts data into the binary tree and maintains the invariant is:

```
fun insert (x, E) = T (E, x, E) | insert (x, s as T (a, y, b))  
= if x < y then T (insert (x, a), y, b) else if x > y then T  
(a, y, insert (x, b)) else s
```

After executing

`ys = insert ("e", xs)`

we end up with the following:



Notice two points: Firstly the original tree (xs) persists. Secondly many common nodes are shared between the old tree and the new tree. Such persistence and sharing is difficult to manage without some form of **garbage collection** (GC) to automatically free up nodes which have no live references, and this is why GC is a feature commonly found in **functional programming languages**.

1.7.5 Reference cycles

Since every value in a purely functional computation is built up out of existing values, it would seem that it is impossible to create a cycle of references. In that case, the reference graph (the graph of the references from object to object) could only be a **directed acyclic graph**. However, in most functional languages, functions can be defined **recursively**; this capability allows recursive structures using functional suspensions. In **lazy** languages, such as **Haskell**, all data structures are represented as implicitly suspended **thunks**; in these languages any data structure can be recursive because a value can be defined in terms of itself. Some other languages, such as **OCaml**, allow the explicit definition of recursive values.

1.7.6 See also

- Persistent data
- Navigational database
- Copy-on-write
- Retroactive data structures

1.7.7 References

- [1] Kaplan, Haim (2001). “Persistent data structures”. *Handbook on Data Structures and Applications* (CRC Press).
- [2] Neil Sarnak, Robert E. Tarjan (1986). “Planar Point Location Using Persistent Search Trees” (PDF). *Communications of the ACM* **29** (7): 669–679. doi:10.1145/6138.6151.
- [3] Chris Okasaki. “Purely Functional Data Structures (thesis)” (PDF).
- [4] Liljenzin, Olle. “Confluently Persistent Sets and Maps”.

1.7.8 Further reading

- Persistent Data Structures and Managed References - video presentation by Rich Hickey on Clojure’s use of persistent data structures and how they support concurrency
- Making Data Structures Persistent by James R. Driscoll, Neil Sarnak, Daniel D. Sleator, Robert E. Tarjan

- Fully persistent arrays for efficient incremental updates and voluminous reads
- Real-Time Deques, Multihead Turing Machines, and Purely Functional Programming
- *Purely functional data structures* by Chris Okasaki, Cambridge University Press, 1998, ISBN 0-521-66350-4.
- Purely Functional Data Structures thesis by Chris Okasaki (PDF format)
- Fully Persistent Lists with Catenation by James R. Driscoll, Daniel D. Sleator, Robert E. Tarjan (PDF)
- Persistent Data Structures from MIT open course Advanced Algorithms

1.7.9 External links

- Lightweight Java implementation of Persistent Red-Black Trees
- •

1.8 Concurrent data structure

In **computer science**, a **concurrent data structure** is a particular way of storing and organizing **data** for access by multiple computing **threads** (or processes) on a **computer**.

Historically, such data structures were used on **uniprocessor** machines with **operating systems** that supported multiple computing threads (or processes). The term **concurrency** captured the **multiplexing/interleaving** of the threads’ operations on the data by the operating system, even though the processors never issued two operations that accessed the data simultaneously.

Today, as **multiprocessor** computer architectures that provide **parallelism** become the dominant computing platform (through the proliferation of **multi-core** processors), the term has come to stand mainly for data structures that can be accessed by multiple threads which may actually access the data simultaneously because they run on different processors that communicate with one another. The concurrent data structure (sometimes also called a **shared data structure**) is usually considered to reside in an abstract storage environment called **shared memory**, though this memory may be physically implemented as either a “tightly coupled” or a distributed collection of storage modules.

1.8.1 Basic principles

Concurrent data structures, intended for use in parallel or distributed computing environments, differ from

“sequential” data structures, intended for use on a uniprocessor machine, in several ways.^[1] Most notably, in a sequential environment one specifies the data structure’s properties and checks that they are implemented correctly, by providing **safety properties**. In a concurrent environment, the specification must also describe **liveness properties** which an implementation must provide. Safety properties usually state that something bad never happens, while liveness properties state that something good keeps happening. These properties can be expressed, for example, using **Linear Temporal Logic**.

The type of liveness requirements tend to define the data structure. The method calls can be **blocking** or **non-blocking**. Data structures are not restricted to one type or the other, and can allow combinations where some method calls are blocking and others are non-blocking (examples can be found in the **Java concurrency** software library).

The safety properties of concurrent data structures must capture their behavior given the many possible interleavings of methods called by different threads. It is quite intuitive to specify how abstract data structures behave in a sequential setting in which there are no interleavings. Therefore, many mainstream approaches for arguing the safety properties of a concurrent data structure (such as **serializability**, **linearizability**, **sequential consistency**, and **quiescent consistency**^[1]) specify the structures properties sequentially, and map its concurrent executions to a collection of sequential ones.

In order to guarantee the safety and liveness properties, concurrent data structures must typically (though not always) allow threads to reach **consensus** as to the results of their simultaneous data access and modification requests. To support such agreement, concurrent data structures are implemented using special primitive synchronization operations (see **synchronization primitives**) available on modern **multiprocessor** machines that allow multiple threads to reach consensus. This consensus can be achieved in a blocking manner by using **locks**, or without locks, in which case it is **non-blocking**. There is a wide body of theory on the design of concurrent data structures (see bibliographical references).

1.8.2 Design and Implementation

Concurrent data structures are significantly more difficult to design and to verify as being correct than their sequential counterparts.

The primary source of this additional difficulty is concurrency, exacerbated by the fact that threads must be thought of as being completely asynchronous: they are subject to operating system preemption, page faults, interrupts, and so on.

On today’s machines, the layout of processors and memory, the layout of data in memory, the communication

load on the various elements of the multiprocessor architecture all influence performance. Furthermore, there is a tension between correctness and performance: algorithmic enhancements that seek to improve performance often make it more difficult to design and verify a correct data structure implementation.^[2]

A key measure for performance is scalability, captured by the **speedup** of the implementation. Speedup is a measure of how effectively the application is utilizing the machine it is running on. On a machine with P processors, the speedup is the ratio of the structures execution time on a single processor to its execution time on T processors. Ideally, we want linear speedup: we would like to achieve a speedup of P when using P processors. Data structures whose speedup grows with P are called **scalable**. The extent to which one can scale the performance of a concurrent data structure is captured by a formula known as **Amdahl’s law** and more refined versions of it such as **Gustafson’s law**.

A key issue with the performance of concurrent data structures is the level of memory contention: the overhead in traffic to and from memory as a result of multiple threads concurrently attempting to access the same locations in memory. This issue is most acute with blocking implementations in which locks control access to memory. In order to acquire a lock, a thread must repeatedly attempt to modify that location. On a **cache-coherent** multiprocessor (one in which processors have local caches that are updated by hardware in order to keep them consistent with the latest values stored) this results in long waiting times for each attempt to modify the location, and is exacerbated by the additional memory traffic associated with unsuccessful attempts to acquire the lock.

1.8.3 See also

- Java concurrency (JSR 166)

1.8.4 References

- [1] Mark Moir and Nir Shavit (2007). “*Concurrent Data Structures*”. In Dinesh Metha and Sartaj Sahni. ‘*Handbook of Data Structures and Applications*’ (1st ed.). Chapman and Hall/CRC Press. pp. 47–14–47–30.
- [2] Gramoli, V. (2015). *More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms* (PDF). Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM. pp. 1–10.

1.8.5 Further reading

- Nancy Lynch “Distributed Computing”

- Hagit Attiya and Jennifer Welch “Distributed Computing: Fundamentals, Simulations And Advanced Topics, 2nd Ed”
- Doug Lea, “Concurrent Programming in Java: Design Principles and Patterns”
- Maurice Herlihy and Nir Shavit, “The Art of Multi-processor Programming”
- Mattson, Sanders, and Massingil “Patterns for Parallel Programming”

1.8.6 External links

- Multithreaded data structures for parallel computing, Part 1 (Designing concurrent data structures) by Arpan Sen
- Multithreaded data structures for parallel computing: Part 2 (Designing concurrent data structures without mutexes) by Arpan Sen
- libcds - C++ library of lock-free containers and safe memory reclamation schema
- Synchrobench - C/C++ and Java librairies and benchmarks of lock-free, lock-based, TM-based and RCU/COW-based data structures.

Chapter 2

Abstract data types

2.1 Abstract data type

Not to be confused with **Algebraic data type**.

In computer science, an **abstract data type (ADT)** is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a *user* of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This contrasts with **data structures**, which are concrete representations of data, and are the point of view of an implementer, not a user.

Formally, an ADT may be defined as a “class of objects whose logical behavior is defined by a set of values and a set of operations”;^[1] this is analogous to an **algebraic structure** in mathematics. What is meant by “behavior” varies by author, with the two main types of formal specifications for behavior being *axiomatic (algebraic) specification* and an *abstract model*;^[2] these correspond to **axiomatic semantics** and **operational semantics** of an **abstract machine**, respectively. Some authors also include the **computational complexity** (“cost”), both in terms of time (for computing operations) and space (for representing values).

In practice many common data types are not ADTs, as the abstraction is not perfect, and users must be aware of issues like **arithmetic overflow** that are due to the representation. For example, integers are often implemented as fixed width (32-bit or 64-bit binary numbers), and thus experience **integer overflow** if the maximum value is exceeded.

ADTs are a theoretical concept in computer science, used in the design and analysis of algorithms, data structures, and software systems, and do not correspond to specific features of **computer languages** – mainstream computer languages do not directly support formally specified ADTs. However, various language features correspond to certain aspects of ADTs, and are easily confused with ADTs proper; these include **abstract types**, **opaque data types**, **protocols**, and **design by contract**. ADTs were first proposed by Barbara Liskov and Stephen N. Zilles in 1974, as part of the development of the **CLU** language.^[3]

2.1.1 Examples

For example, integers are an ADT, defined as the values $0, 1, -1, 2, \dots$, and by the operations of addition, subtraction, multiplication, and division, together with greater than, less than, etc., which behave according to familiar mathematics (with care for **integer division**), independently of how the integers are represented by the computer.^[lower-alpha 1] Explicitly, “behavior” includes obeying various axioms (associativity and commutativity of addition etc.), and preconditions on operations (cannot divide by zero). Typically integers are represented in a data structure as **binary numbers**, most often as two’s complement, but might be **binary-coded decimal** or in ones’ complement, but the user is abstracted from the concrete choice of representation, and can simply use the data as integers.

An ADT consists not only of operations, but also of values of the underlying data and of constraints on the operations. An “interface” typically refers only to the operations, and perhaps some of the constraints on the operations, notably pre-conditions and post-conditions, but not other constraints, such as relations between the operations.

For example, an abstract **stack**, which is a last-in-first-out structure, could be defined by three operations: **push**, that inserts some data item onto the structure, **pop**, that extracts an item from it, and **peek** or **top**, that allows data on top of the structure to be examined without removal. An abstract **queue** data structure, which is a first-in-first-out structure, would also have three operations, **enqueue** to join the queue; **dequeue**, to remove the first element from the queue; and **front**, in order to access and serve the first element in the queue. There would be no way of differentiating these two data types, unless a mathematical constraint is introduced that for a stack specifies that each **pop** always returns the most recently pushed item that has not been popped yet. When analyzing the **efficiency** of algorithms that use stacks, one may also specify that all operations take the same time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.

2.1.2 Introduction

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages. However, an ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs.

The term **abstract data type** can also be regarded as a generalised approach of a number of algebraic structures, such as lattices, groups, and rings.^[4] The notion of abstract data types is related to the concept of **data abstraction**, important in **object-oriented programming** and **design by contract** methodologies for software development.

2.1.3 Defining an abstract data type

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between “imperative” and “functional” definition styles.

Imperative definition style

In the “imperative” definition style, which is closer to the philosophy of **imperative programming** languages, an abstract data structure is conceived as an entity that is *mutable* — meaning that it may be in different *states* at different times. Some operations may change the state of the ADT; therefore, the order in which operations are evaluated is important, and the same operation on the same entities may have different effects if executed at different times — just like the instructions of a computer, or the commands and procedures of an imperative language. To underscore this view, it is customary to say that the operations are *executed* or *applied*, rather than *evaluated*. The imperative style is often used when describing abstract algorithms. This is described by Donald E. Knuth and can be referenced from here [The Art of Computer Programming](#).

Abstract variable Imperative ADT definitions often depend on the concept of an *abstract variable*, which may be regarded as the simplest non-trivial ADT. An abstract variable V is a mutable entity that admits two operations:

- $\text{store}(V, x)$ where x is a *value* of unspecified nature; and
- $\text{fetch}(V)$, that yields a value;

with the constraint that

- $\text{fetch}(V)$ always returns the value x used in the most recent $\text{store}(V, x)$ operation on the same variable V .

As in so many programming languages, the operation $\text{store}(V, x)$ is often written $V \leftarrow x$ (or some similar notation), and $\text{fetch}(V)$ is implied whenever a variable V is used in a context where a value is required. Thus, for example, $V \leftarrow V + 1$ is commonly understood to be a shorthand for $\text{store}(V, \text{fetch}(V) + 1)$.

In this definition, it is implicitly assumed that storing a value into a variable U has no effect on the state of a distinct variable V . To make this assumption explicit, one could add the constraint that

- if U and V are distinct variables, the sequence $\{ \text{store}(U, x); \text{store}(V, y) \}$ is equivalent to $\{ \text{store}(V, y); \text{store}(U, x) \}$.

More generally, ADT definitions often assume that any operation that changes the state of one ADT instance has no effect on the state of any other instance (including other instances of the same ADT) — unless the ADT axioms imply that the two instances are connected (**aliased**) in that sense. For example, when extending the definition of abstract variable to include abstract **records**, the operation that selects a field from a record variable R must yield a variable V that is aliased to that part of R .

The definition of an abstract variable V may also restrict the stored values x to members of a specific set X , called the *range* or *type* of V . As in programming languages, such restrictions may simplify the description and analysis of algorithms, and improve their readability.

Note that this definition does not imply anything about the result of evaluating $\text{fetch}(V)$ when V is *un-initialized*, that is, before performing any store operation on V . An algorithm that does so is usually considered invalid, because its effect is not defined. (However, there are some important algorithms whose efficiency strongly depends on the assumption that such a fetch is legal, and returns some arbitrary value in the variable's range.)

Instance creation Some algorithms need to create new instances of some ADT (such as new variables, or new stacks). To describe such algorithms, one usually includes in the ADT definition a $\text{create}()$ operation that yields an instance of the ADT, usually with axioms equivalent to

- the result of $\text{create}()$ is distinct from any instance S in use by the algorithm.

This axiom may be strengthened to exclude also partial aliasing with other instances. On the other hand, this axiom still allows implementations of `create()` to yield a previously created instance that has become inaccessible to the program.

Preconditions, postconditions, and invariants In imperative-style definitions, the axioms are often expressed by *preconditions*, that specify when an operation may be executed; *postconditions*, that relate the states of the ADT before and after the execution of each operation; and *invariants*, that specify properties of the ADT that are *not* changed by the operations.

Example: abstract stack (imperative) As another example, an imperative definition of an abstract stack could specify that the state of a stack S can be modified only by the operations

- $\text{push}(S, x)$, where x is some *value* of unspecified nature; and
- $\text{pop}(S)$, that yields a value as a result;

with the constraint that

- For any value x and any abstract variable V , the sequence of operations $\{ \text{push}(S, x); V \leftarrow \text{pop}(S) \}$ is equivalent to $\{ V \leftarrow x \}$;

Since the assignment $\{ V \leftarrow x \}$, by definition, cannot change the state of S , this condition implies that $\{ V \leftarrow \text{pop}(S) \}$ restores S to the state it had before the $\{ \text{push}(S, x) \}$. From this condition and from the properties of abstract variables, it follows, for example, that the sequence

$$\{ \text{push}(S, x); \text{push}(S, y); U \leftarrow \text{pop}(S); \text{push}(S, z); V \leftarrow \text{pop}(S); W \leftarrow \text{pop}(S); \}$$

where x, y , and z are any values, and U, V, W are pairwise distinct variables, is equivalent to

$$\{ U \leftarrow y; V \leftarrow z; W \leftarrow x \}$$

Here it is implicitly assumed that operations on a stack instance do not modify the state of any other ADT instance, including other stacks; that is,

- For any values x, y , and any distinct stacks S and T , the sequence $\{ \text{push}(S, x); \text{push}(T, y) \}$ is equivalent to $\{ \text{push}(T, y); \text{push}(S, x) \}$.

A stack ADT definition usually includes also a Boolean-valued function `empty(S)` and a `create()` operation that returns a stack instance, with axioms equivalent to

- $\text{create}() \neq S$ for any stack S (a newly created stack is distinct from all previous stacks)
- $\text{empty}(\text{create}())$ (a newly created stack is empty)
- $\text{not empty}(\text{push}(S, x))$ (pushing something into a stack makes it non-empty)

Single-instance style Sometimes an ADT is defined as if only one instance of it existed during the execution of the algorithm, and all operations were applied to that instance, which is not explicitly notated. For example, the abstract stack above could have been defined with operations `push(x)` and `pop()`, that operate on “the” only existing stack. ADT definitions in this style can be easily rewritten to admit multiple coexisting instances of the ADT, by adding an explicit instance parameter (like S in the previous example) to every operation that uses or modifies the implicit instance.

On the other hand, some ADTs cannot be meaningfully defined without assuming multiple instances. This is the case when a single operation takes two distinct instances of the ADT as parameters. For an example, consider augmenting the definition of the stack ADT with an operation `compare(S, T)` that checks whether the stacks S and T contain the same items in the same order.

Functional ADT definitions

Another way to define an ADT, closer to the spirit of functional programming, is to consider each state of the structure as a separate entity. In this view, any operation that modifies the ADT is modeled as a mathematical function that takes the old state as an argument, and returns the new state as part of the result. Unlike the “imperative” operations, these functions have no *side effects*. Therefore, the order in which they are evaluated is immaterial, and the same operation applied to the same arguments (including the same input states) will always return the same results (and output states).

In the functional view, in particular, there is no way (or need) to define an “abstract variable” with the semantics of imperative variables (namely, with fetch and store operations). Instead of storing values into variables, one passes them as arguments to functions.

Example: abstract stack (functional) For example, a complete functional-style definition of a stack ADT could use the three operations:

- `push`: takes a stack state and an arbitrary value, returns a stack state;
- `top`: takes a stack state, returns a value;
- `pop`: takes a stack state, returns a stack state;

In a functional-style definition there is no need for a create operation. Indeed, there is no notion of “stack instance”. The stack states can be thought of as being potential states of a single stack structure, and two stack states that contain the same values in the same order are considered to be identical states. This view actually mirrors the behavior of some concrete implementations, such as linked lists with `hash cons`.

Instead of `create()`, a functional definition of a stack ADT may assume the existence of a special stack state, the *empty stack*, designated by a special symbol like Λ or "`()`"; or define a `bottom()` operation that takes no arguments and returns this special stack state. Note that the axioms imply that

- $\text{push}(\Lambda, x) \neq \Lambda$

In a functional definition of a stack one does not need an empty predicate: instead, one can test whether a stack is empty by testing whether it is equal to Λ .

Note that these axioms do not define the effect of `top(s)` or `pop(s)`, unless s is a stack state returned by a `push`. Since `push` leaves the stack non-empty, those two operations are undefined (hence invalid) when $s = \Lambda$. On the other hand, the axioms (and the lack of side effects) imply that $\text{push}(s, x) = \text{push}(t, y)$ if and only if $x = y$ and $s = t$.

As in some other branches of mathematics, it is customary to assume also that the stack states are only those whose existence can be proved from the axioms in a finite number of steps. In the stack ADT example above, this rule means that every stack is a *finite* sequence of values, that becomes the empty stack (Λ) after a finite number of pops. By themselves, the axioms above do not exclude the existence of infinite stacks (that can be popped forever, each time yielding a different state) or circular stacks (that return to the same state after a finite number of pops). In particular, they do not exclude states s such that $\text{pop}(s) = s$ or $\text{push}(s, x) = s$ for some x . However, since one cannot obtain such stack states with the given operations, they are assumed “not to exist”.

Whether to include complexity

Aside from the behavior in terms of axioms, it is also possible to include, in the definition of an ADT's operations, their *algorithmic complexity*. Alexander Stepanov, designer of the C++ Standard Template Library, included complexity guarantees in the STL's specification, arguing:

The reason for introducing the notion of abstract data types was to allow interchangeable software modules. You cannot have interchangeable modules unless these modules share similar complexity behavior. If I replace

one module with another module with the same functional behavior but with different complexity tradeoffs, the user of this code will be unpleasantly surprised. I could tell him anything I like about data abstraction, and he still would not want to use the code. Complexity assertions have to be part of the interface.

—Alexander Stepanov^[5]

2.1.4 Advantages of abstract data typing

- Encapsulation

Abstraction provides a promise that any implementation of the ADT has certain properties and abilities; knowing these is all that is required to make use of an ADT object. The user does not need any technical knowledge of how the implementation works to use the ADT. In this way, the implementation may be complex but will be encapsulated in a simple interface when it is actually used.

- Localization of change

Code that uses an ADT object will not need to be edited if the implementation of the ADT is changed. Since any changes to the implementation must still comply with the interface, and since code using an ADT may only refer to properties and abilities specified in the interface, changes may be made to the implementation without requiring any changes in code where the ADT is used.

- Flexibility

Different implementations of an ADT, having all the same properties and abilities, are equivalent and may be used somewhat interchangeably in code that uses the ADT. This gives a great deal of flexibility when using ADT objects in different situations. For example, different implementations of an ADT may be more efficient in different situations; it is possible to use each in the situation where they are preferable, thus increasing overall efficiency.

2.1.5 Typical operations

Some operations that are often specified for ADTs (possibly under other names) are

- `compare(s, t)`, that tests whether two structures are equivalent in some sense;
- `hash(s)`, that computes some standard `hash` function from the instance's state;

- print(*s*) or show(*s*), that produces a human-readable representation of the structure's state.

In imperative-style ADT definitions, one often finds also

- create(), that yields a new instance of the ADT;
- initialize(*s*), that prepares a newly created instance *s* for further operations, or resets it to some "initial state";
- copy(*s,t*), that puts instance *s* in a state equivalent to that of *t*;
- clone(*t*), that performs $s \leftarrow \text{create}()$, $\text{copy}(s,t)$, and returns *s*;
- free(*s*) or destroy(*s*), that reclaims the memory and other resources used by *s*;

The free operation is not normally relevant or meaningful, since ADTs are theoretical entities that do not "use memory". However, it may be necessary when one needs to analyze the storage used by an algorithm that uses the ADT. In that case one needs additional axioms that specify how much memory each ADT instance uses, as a function of its state, and how much of it is returned to the pool by free.

2.1.6 Examples

Some common ADTs, which have proved useful in a great variety of applications, are

- Container
- Deque
- List
- Map
- Multimap
- Multiset
- Priority queue
- Queue
- Set
- Stack
- Tree
- Graph

Each of these ADTs may be defined in many ways and variants, not necessarily equivalent. For example, a stack ADT may or may not have a count operation that tells how many items have been pushed and not yet popped. This choice makes a difference not only for its clients but also for the implementation.

2.1.7 Implementation

Further information: [Opaque data type](#)

Implementing an ADT means providing one procedure or function for each abstract operation. The ADT instances are represented by some concrete **data structure** that is manipulated by those procedures, according to the ADT's specifications.

Usually there are many ways to implement the same ADT, using several different concrete data structures. Thus, for example, an abstract stack can be implemented by a [linked list](#) or by an [array](#).

In order to prevent clients from depending on the implementation, an ADT is often packaged as an *opaque data type* in one or more **modules**, whose interface contains only the signature (number and types of the parameters and results) of the operations. The implementation of the module — namely, the bodies of the procedures and the concrete data structure used — can then be hidden from most clients of the module. This makes it possible to change the implementation without affecting the clients. If the implementation is exposed, it is known instead as a *transparent data type*.

When implementing an ADT, each instance (in imperative-style definitions) or each state (in functional-style definitions) is usually represented by a **handle** of some sort.^[6]

Modern object-oriented languages, such as [C++](#) and [Java](#), support a form of abstract data types. When a class is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a **class**, and each instance of the ADT is usually an **object** of that class. The module's interface typically declares the constructors as ordinary procedures, and most of the other ADT operations as methods of that class. However, such an approach does not easily encapsulate multiple representational variants found in an ADT. It also can undermine the extensibility of object-oriented programs. In a pure object-oriented program that uses interfaces as types, types refer to behaviors not representations.

Example: implementation of the stack ADT

As an example, here is an implementation of the stack ADT above in the C programming language.

Imperative-style interface

An imperative-style interface might be:

```
typedef struct stack_Rep stack_Rep; /* Type: instance representation (an opaque record). */
/* Type: handle to a stack instance (an opaque pointer). */
/* Type: value that can be stored in stack (arbitrary address). */
stack_T
```

```
stack_create(void); /* Create new stack instance, initially
empty. */ void stack_push(stack_T s, stack_Item e);
/* Add an item at the top of the stack. */ stack_Item
stack_pop(stack_T s); /* Remove the top item from the
stack and return it. */ int stack_empty(stack_T ts); /* Check whether stack is empty. */
```

This implementation could be used in the following manner:

```
#include <stack.h> /* Include the stack interface. */
stack_T t = stack_create(); /* Create a stack instance.
*/ int foo = 17; /* An arbitrary datum. */ stack_push(t,
&foo); /* Push the address of 'foo' onto the stack. */ ...
void *e = stack_pop(t); /* Get the top item and delete
it from the stack. */ if (stack_empty(t)) { ... } /* Do
something if stack is empty. */ ...
```

This interface can be implemented in many ways. The implementation may be arbitrarily inefficient, since the formal definition of the ADT, above, does not specify how much space the stack may use, nor how long each operation should take. It also does not specify whether the stack state t continues to exist after a call $s \leftarrow \text{pop}(t)$.

In practice the formal definition should specify that the space is proportional to the number of items pushed and not yet popped; and that every one of the operations above must finish in a constant amount of time, independently of that number. To comply with these additional specifications, the implementation could use a linked list, or an array (with dynamic resizing) together with two integers (an item count and the array size)

Functional-style interface Functional-style ADT definitions are more appropriate for functional programming languages, and vice versa. However, one can provide a functional style interface even in an imperative language like C. For example:

```
typedef struct stack_Rep stack_Rep; /* Type: stack state
representation (an opaque record). */ typedef stack_Rep
*stack_T; /* Type: handle to a stack state (an opaque
pointer). */ typedef void *stack_Item; /* Type: item
(arbitrary address). */ stack_T stack_empty(void);
/* Returns the empty stack state. */ stack_T
stack_push(stack_T s, stack_Item x); /* Adds x at
the top of s, returns the resulting state. */ stack_Item
stack_top(stack_T s); /* Returns the item currently
at the top of s. */ stack_T stack_pop(stack_T s); /* Remove
the top item from s, returns the resulting state. */
```

The main problem is that C lacks garbage collection, and this makes this style of programming impractical; moreover, memory allocation routines in C are slower than allocation in a typical garbage collector, thus the performance impact of so many allocations is even greater.

ADT libraries

Many modern programming languages, such as C++ and Java, come with standard libraries that implement several common ADTs, such as those listed above.

Built-in abstract data types

The specification of some programming languages is intentionally vague about the representation of certain **built-in data types**, defining only the operations that can be done on them. Therefore, those types can be viewed as “built-in ADTs”. Examples are the arrays in many scripting languages, such as **Awk**, **Lua**, and **Perl**, which can be regarded as an implementation of the Map or Table ADT.

2.1.8 See also

- Initial algebra
- Concept (generic programming)
- Design by contract
- Formal methods
- Functional specification
- Liskov substitution principle
- Object-oriented programming
- Opaque data type
- Type system
- Type theory
- Algebraic data type
- Generalized algebraic data type

2.1.9 Notes

- [1] Compare to the characterization of integers in abstract algebra

2.1.10 References

- [1] Dale & Walker 1996, p. 3.
[2] Dale & Walker 1996, p. 4.
[3] Liskov & Zilles 1974.
[4] Rudolf Lidl (2004). *Abstract Algebra*. Springer. ISBN 81-8128-149-7., Chapter 7, section 40.
[5] Stevens, Al (March 1995). “Al Stevens Interviews Alex Stepanov”. *Dr. Dobb's Journal*. Retrieved 31 January 2015.

[6] Robert Sedgewick (1998). *Algorithms in C*. Addison/Wesley. ISBN 0-201-31452-5., definition 4.4.

- Liskov, Barbara; Zilles, Stephen (1974). “Programming with abstract data types”. *Proceedings of the ACM SIGPLAN symposium on Very high level languages*. pp. 50–59. doi:10.1145/800233.807045.
- Dale, Nell; Walker, Henry M. (1996). *Abstract Data Types: Specifications, Implementations, and Applications*. Jones & Bartlett Learning. ISBN 978-0-66940000-7.

2.1.11 Further

- Mitchell, John C.; Plotkin, Gordon (July 1988). “Abstract Types Have Existential Type” (PDF). *ACM Transactions on Programming Languages and Systems* 10 (3).

2.1.12 External links

- Abstract data type in NIST Dictionary of Algorithms and Data Structures
- Walls and Mirrors, the classic textbook

2.2 List

This article is about sequential data structures. For random-access data structures, see [Array data structure](#).

In computer science, a **list** or **sequence** is an abstract data type that represents a sequence of **values**, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a finite **sequence**; the (potentially) infinite analog of a list is a **stream**.^{[1]:§3.5} Lists are a basic example of **containers**, as they contain other values. If the same value occurs multiple times, each occurrence is considered a distinct item.



A singly linked list structure, implementing a list with 3 integer elements.

The name **list** is also used for several concrete **data structures** that can be used to implement abstract lists, especially **linked lists**.

Many **programming languages** provide support for **list data types**, and have special syntax and semantics for lists and list operations. A list can often be constructed by writing the items in sequence, separated by **commas**, **semicolons**, or **spaces**, within a pair of delimiters such as

parentheses '()', brackets '[]', braces '{}', or angle brackets '<>'. Some languages may allow list types to be indexed or sliced like **array types**, in which case the data type is more accurately described as an array. In **object-oriented programming languages**, lists are usually provided as instances of subclasses of a generic “list” class, and traversed via separate **iterators**. List data types are often implemented using **array data structures** or linked lists of some sort, but other **data structures** may be more appropriate for some applications. In some contexts, such as in **Lisp** programming, the term **list** may refer specifically to a linked list rather than an array.

In **type theory** and **functional programming**, abstract lists are usually defined inductively by two operations: **nil** that yields the empty list, and **cons**, which adds an item at the beginning of a list.^[2]

2.2.1 Operations

Implementation of the list data structure may provide some of the following operations:

- a constructor for creating an empty list;
- an operation for testing whether or not a list is empty;
- an operation for prepending an entity to a list
- an operation for appending an entity to a list
- an operation for determining the first component (or the “head”) of a list
- an operation for referring to the list consisting of all the components of a list except for its first (this is called the “tail” of the list.)

2.2.2 Implementations

Lists are typically implemented either as **linked lists** (either singly or doubly linked) or as **arrays**, usually **variable length** or **dynamic arrays**.

The standard way of implementing lists, originating with the programming language **Lisp**, is to have each element of the list contain both its value and a pointer indicating the location of the next element in the list. This results in either a **linked list** or a **tree**, depending on whether the list has nested sublists. Some older Lisp implementations (such as the Lisp implementation of the **Symbolics 3600**) also supported “compressed lists” (using **CDR coding**) which had a special internal representation (invisible to the user). Lists can be manipulated using **iteration** or **recursion**. The former is often preferred in **imperative programming languages**, while the latter is the norm in **functional languages**.

Lists can be implemented as **self-balancing binary search trees** holding index-value pairs, providing equal-time access to any element (e.g. all residing in the fringe, and internal nodes storing the right-most child's index, used to guide the search), taking the time logarithmic in the list's size, but as long as it doesn't change much will provide the illusion of **random access** and enable swap, prefix and append operations in logarithmic time as well.^[3]

2.2.3 Programming language support

Some languages do not offer a list data structure, but offer the use of **associative arrays** or some kind of table to emulate lists. For example, **Lua** provides tables. Although **Lua** stores lists that have numerical indices as arrays internally, they still appear as hash tables.^[4]

In **Lisp**, lists are the fundamental data type and can represent both program code and data. In most dialects, the list of the first three prime numbers could be written as `(list 2 3 5)`. In several dialects of **Lisp**, including **Scheme**, a list is a collection of pairs, consisting of a value and a pointer to the next pair (or null value), making a singly linked list.^[5]

2.2.4 Applications

As the name implies, lists can be used to store a list of records.

Because in computing, lists are easier to realize than sets, a finite **set** in mathematical sense can be realized as a list with additional restrictions, that is, duplicate elements are disallowed and such that order is irrelevant. If the list is sorted, it speeds up determining if a given item is already in the set but in order to ensure the order, it requires more time to add new entry to the list. In efficient implementations, however, sets are implemented using **self-balancing binary search trees** or **hash tables**, rather than a list.

2.2.5 Abstract definition

The abstract list type L with elements of some type E (a monomorphic list) is defined by the following functions:

$\text{nil}: () \rightarrow L$
 $\text{cons}: E \times L \rightarrow L$
 $\text{first}: L \rightarrow E$
 $\text{rest}: L \rightarrow L$

with the axioms

$\text{first}(\text{cons}(e, l)) = e$
 $\text{rest}(\text{cons}(e, l)) = l$

for any element e and any list l . It is implicit that

$\text{cons}(e, l) \neq l$
 $\text{cons}(e, l) \neq e$
 $\text{cons}(e_1, l_1) = \text{cons}(e_2, l_2)$ if $e_1 = e_2$ and $l_1 = l_2$

Note that **first** (`nil()`) and **rest** (`nil()`) are not defined.

These axioms are equivalent to those of the abstract stack data type.

In **type theory**, the above definition is more simply regarded as an **inductive type** defined in terms of constructors: **nil** and **cons**. In algebraic terms, this can be represented as the transformation $1 + E \times L \rightarrow L$. **first** and **rest** are then obtained by **pattern matching** on the **cons** constructor and separately handling the **nil** case.

The list monad

The list type forms a **monad** with the following functions (using E^* rather than L to represent monomorphic lists with elements of type E):

$\text{return}: A \rightarrow A^* = a \mapsto \text{cons } a \text{ nil}$

$\text{bind}: A^* \rightarrow (A \rightarrow B^*) \rightarrow B^* = l \mapsto f \mapsto \begin{cases} \text{nil} \\ \text{append}(f a) (\text{bind } l' f) \end{cases}$

where **append** is defined as:

$\text{append}: A^* \rightarrow A^* \rightarrow A^* = l_1 \mapsto l_2 \mapsto \begin{cases} l_2 & \text{if } l_1 = \text{nil} \\ \text{cons } a (\text{append } l'_1 l_2) & \text{if } l_1 = \text{cons } a l'_1 \end{cases}$

Alternatively, the monad may be defined in terms of operations **return**, **fmap** and **join**, with:

$\text{fmap}: (A \rightarrow B) \rightarrow (A^* \rightarrow B^*) = f \mapsto l \mapsto \begin{cases} \text{nil} & \text{if } l = \text{nil} \\ \text{cons}(f a) (\text{fmap } f l') & \text{if } l = \text{cons } a l' \end{cases}$

$\text{join}: A^{**} \rightarrow A^* = l \mapsto \begin{cases} \text{nil} & \text{if } l = \text{nil} \\ \text{append } a (\text{join } l') & \text{if } l = \text{cons } a l' \end{cases}$

Note that **fmap**, **join**, **append** and **bind** are well-defined, since they're applied to progressively deeper arguments at each recursive call.

The list type is an additive monad, with **nil** as the monadic zero and **append** as monadic sum.

Lists form a **monoid** under the **append** operation. The identity element of the monoid is the empty list, **nil**. In fact, this is the **free monoid** over the set of list elements.

2.2.6 References

- [1] Abelson, Harold; Sussman, Gerald Jay (1996). *Structure and Interpretation of Computer Programs*. MIT Press.
- [2] Reingold, Edward; Nievergelt, Jurg; Narsingh, Deo (1977). *Combinatorial Algorithms: Theory and Practice*. Englewood Cliffs, New Jersey: Prentice Hall. pp. 38–41. ISBN 0-13-152447-X.
- [3] Barnett, Granville; Del tonga, Luca (2008). “Data Structures and Algorithms” (PDF). mta.ca. Retrieved 12 November 2014.
- [4] Lerusalimschy, Roberto (December 2003). *Programming in Lua (first edition)* (First ed.). Lua.org. ISBN 8590379817. Retrieved 12 November 2014.
- [5] Steele, Guy (1990). *Common Lisp* (Second Edition ed.). Digital Press. pp. 29–31. ISBN 1-55558-041-6.

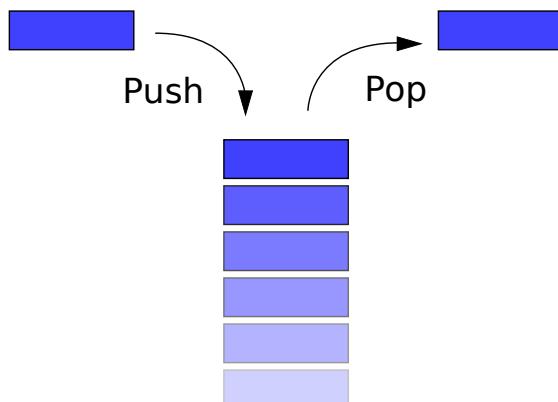
2.2.7 See also

- Array
- Queue
- Set
- Stream

2.3 Stack

For the use of the term LIFO in accounting, see **LIFO (accounting)**.

In computer science, a **stack** or **LIFO (last in, first**



Simple representation of a stack

out) is an **abstract data type** that serves as a collection of elements, with two principal operations: *push*, which adds an element to the collection, and *pop*, which removes the last element that was added.^[1]

The term LIFO stems from the fact that, using these operations, each element “popped off” a stack in series of pushes and pops is the last (most recent) element that was

“pushed into” within the sequence. This is equivalent to the requirement that, considered as a **linear data structure**, or more abstractly a sequential collection, the push and pop operations occur only at one end of the structure, referred to as the *top* of the stack. (Additionally, a *peek* operation may give access to the top.)

A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an **overflow** state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into **underflow** state, which means no items are present in stack to be removed.

A stack is a *restricted data structure*, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition. Therefore, the lower elements are those that have been on the stack the longest.^[2]

2.3.1 History

The stack was first proposed in 1946, in the computer design of **Alan M. Turing** (who used the terms “bury” and “unbury”) as a means of calling and returning from subroutines.^[3] Subroutines had already been implemented in **Konrad Zuse**’s **Z4** in 1945. **Klaus Samelson** and **Friedrich L. Bauer** of **Technical University Munich** proposed the idea in 1955 and filed a patent in 1957.^[4] The same concept was developed, independently, by the **Australian Charles Leonard Hamblin** in the first half of 1957.^[5]

The term *stack* may have originated by analogy to a spring-loaded stack of plates in a cafeteria.^[6] Clean plates are placed on top of the stack, pushing down any already there. When a plate is removed from the stack the one below it pops up to become the new top.

2.3.2 Non-essential operations

In many implementations, a stack has more operations than “push” and “pop”. An example is “top of stack”, or “*peek*”, which observes the top-most element *without* removing it from the stack.^[7] Since this can be done with a “pop” and a “push” with the same data, it is not essential. An underflow condition can occur in the “stack top” operation if the stack is empty, the same as “pop”. Also, implementations often have a function which just returns whether the stack is empty.

2.3.3 Software stacks

Implementation

In most **high level languages**, a stack can be easily implemented either through an **array** or a **linked list**. What identifies the data structure as a stack in either case is not the implementation but the interface: the user is only allowed to pop or push items onto the array or linked list, with few other helper operations. The following will demonstrate both implementations, using **pseudocode**.

Array An array can be used to implement a (bounded) stack, as follows. The first element (usually at the **zero offset**) is the bottom, resulting in `array[0]` being the first element pushed onto the stack and the last element popped off. The program must keep track of the size (length) of the stack, using a variable *top* that records the number of items pushed so far, therefore pointing to the place in the array where the next element is to be inserted (assuming a zero-based index convention). Thus, the stack itself can be effectively implemented as a three-element structure:

```
structure stack: maxsize : integer top : integer items : array of item
procedure initialize(stk : stack, size : integer): stk.items ← new array of size items, initially empty
stk.maxsize ← size stk.top ← 0
```

The *push* operation adds an element and increments the *top* index, after checking for overflow:

```
procedure push(stk : stack, x : item): if
stk.top = stk.maxsize: report overflow error else:
stk.items[stk.top] ← x stk.top ← stk.top + 1
```

Similarly, *pop* decrements the *top* index after checking for underflow, and return the item that was previously the top one:

```
procedure pop(stk : stack): if stk.top = 0: report underflow error else: stk.top ← stk.top - 1 r ←
stk.items[stk.top]
```

Using a **dynamic array**, it is possible to implement a stack that can grow or shrink as much as needed. The size of the stack is simply the size of the dynamic array, which is a very efficient implementation of a stack since adding items to or removing items from the end of a dynamic array requires amortized $O(1)$ time.

Linked list Another option for implementing stacks is to use a **singly linked list**. A stack is then a pointer to the “head” of the list, with perhaps a counter to keep track of the size of the list:

```
structure frame: data : item next : frame or nil
structure stack: head : frame or nil size : integer
procedure initialize(stk : stack): stk.head ← nil stk.size ← 0
```

Pushing and popping items happens at the head of the list; overflow is not possible in this implementation (unless memory is exhausted):

```
procedure push(stk : stack, x : item): newhead ←
new frame newhead.data ← x newhead.next ← stk.head
stk.head ← newhead procedure pop(stk : stack): if
stk.head = nil: report underflow error r ← stk.head.data
stk.head ← stk.head.next return r
```

Stacks and programming languages

Some languages, like Perl, **LISP** and **Python**, do not call for stack implementations, since **push** and **pop** functions are available for any list. All **Forth**-like languages (such as **Adobe PostScript**) are also designed around language-defined stacks that are directly visible to and manipulated by the programmer. Examples from **Common Lisp**:

```
(setf stack (list 'a 'b 'c)) ;; ⇒ (A B C) (pop stack) ;; ⇒ A
stack ;; ⇒ (B C) (push 'new stack) ;; ⇒ (NEW B C)
```

C++'s **Standard Template Library** provides a “stack” templated class which is restricted to only push/pop operations. **Java**'s library contains a **Stack** class that is a specialization of **Vector**. **PHP** has an **SplStack** class.

2.3.4 Hardware stacks

A common use of stacks at the architecture level is as a means of allocating and accessing memory.

Basic architecture of a stack

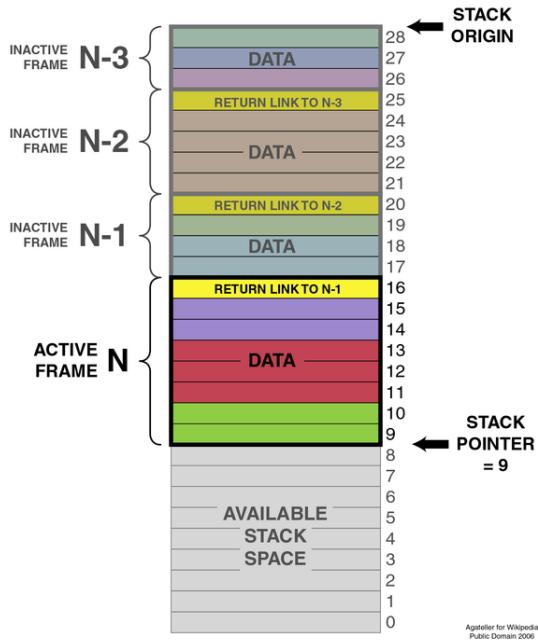
A typical stack is an area of computer memory with a fixed origin and a variable size. Initially the size of the stack is zero. A **stack pointer**, usually in the form of a hardware register, points to the most recently referenced location on the stack; when the stack has a size of zero, the stack pointer points to the origin of the stack.

The two operations applicable to all stacks are:

- a *push* operation, in which a data item is placed at the location pointed to by the stack pointer, and the address in the stack pointer is adjusted by the size of the data item;
- a *pop* or *pull* operation: a data item at the current location pointed to by the stack pointer is removed, and the stack pointer is adjusted by the size of the data item.

There are many variations on the basic principle of stack operations. Every stack has a fixed location in memory at which it begins. As data items are added to the stack, the stack pointer is displaced to indicate the current extent of the stack, which expands away from the origin.

Stack pointers may point to the origin of a stack or to a limited range of addresses either above or below the origin (depending on the direction in which the stack grows);



A typical stack, storing local data and call information for nested procedure calls (not necessarily *nested procedures*!). This stack grows downward from its origin. The stack pointer points to the current topmost datum on the stack. A push operation decrements the pointer and copies the data to the stack; a pop operation copies data from the stack and then increments the pointer. Each procedure called in the program stores procedure return information (in yellow) and local data (in other colors) by pushing them onto the stack. This type of stack implementation is extremely common, but it is vulnerable to buffer overflow attacks (see the text).

however, the stack pointer cannot cross the origin of the stack. In other words, if the origin of the stack is at address 1000 and the stack grows downwards (towards addresses 999, 998, and so on), the stack pointer must never be incremented beyond 1000 (to 1001, 1002, etc.). If a pop operation on the stack causes the stack pointer to move past the origin of the stack, a *stack underflow* occurs. If a push operation causes the stack pointer to increment or decrement beyond the maximum extent of the stack, a *stack overflow* occurs.

Some environments that rely heavily on stacks may provide additional operations, for example:

- *Duplicate*: the top item is popped, and then pushed again (twice), so that an additional copy of the former top item is now on top, with the original below it.
 - *Peek*: the topmost item is inspected (or returned), but the stack pointer is not changed, and the stack size does not change (meaning that the item remains on the stack). This is also called **top** operation in many articles.
 - *Swap* or *exchange*: the two topmost items on the stack exchange places.

- *Rotate (or Roll)*: the n topmost items are moved on the stack in a rotating fashion. For example, if $n=3$, items 1, 2, and 3 on the stack are moved to positions 2, 3, and 1 on the stack, respectively. Many variants of this operation are possible, with the most common being called *left rotate* and *right rotate*.

Stacks are often visualized growing from the bottom up (like real-world stacks). They may also be visualized growing from left to right, so that “topmost” becomes “rightmost”, or even growing from top to bottom. The important feature is that the bottom of the stack is in a fixed position. The image above and to the right is an example of a top to bottom growth visualization: the top (28) is the stack ‘bottom’, since the stack ‘top’ is where items are pushed or popped from.

A *right rotate* will move the first element to the third position, the second to the first and the third to the second. Here are two equivalent visualizations of this process:

apple banana banana ==right rotate==> cucumber cucumber apple cucumber apple banana ==left rotate==> cucumber apple banana

A stack is usually represented in computers by a block of memory cells, with the “bottom” at a fixed location, and the stack pointer holding the address of the current “top” cell in the stack. The top and bottom terminology are used irrespective of whether the stack actually grows towards lower memory addresses or towards higher memory addresses.

Pushing an item on to the stack adjusts the stack pointer by the size of the item (either decrementing or incrementing, depending on the direction in which the stack grows in memory), pointing it to the next cell, and copies the new top item to the stack area. Depending again on the exact implementation, at the end of a push operation, the stack pointer may point to the next unused location in the stack, or it may point to the topmost item in the stack. If the stack points to the current topmost item, the stack pointer will be updated before a new item is pushed onto the stack; if it points to the next available location in the stack, it will be updated *after* the new item is pushed onto the stack.

Popping the stack is simply the inverse of pushing. The topmost item in the stack is removed and the stack pointer is updated, in the opposite order of that used in the push operation.

Hardware support

Stack in main memory Most CPUs have registers that can be used as stack pointers. Processor families like the x86, Z80, 6502, and many others have special instructions that implicitly use a dedicated (hardware) stack pointer to conserve opcode space. Some processors, like the PDP-11 and the 68000, also have special addressing modes for implementation of stacks, typically with

a semi-dedicated stack pointer as well (such as A7 in the 68000). However, in most processors, several different registers may be used as additional stack pointers as needed (whether updated via addressing modes or via add/sub instructions).

Stack in registers or dedicated memory The x87 floating point architecture is an example of a set of registers organised as a stack where direct access to individual registers (relative the current top) is also possible. As with stack-based machines in general, having the top-of-stack as an implicit argument allows for a small machine code footprint with a good usage of bus bandwidth and code caches, but it also prevents some types of optimizations possible on processors permitting random access to the register file for all (two or three) operands. A stack structure also makes superscalar implementations with register renaming (for speculative execution) somewhat more complex to implement, although it is still feasible, as exemplified by modern x87 implementations.

Sun SPARC, AMD Am29000, and Intel i960 are all examples of architectures using register windows within a register-stack as another strategy to avoid the use of slow main memory for function arguments and return values.

There are also a number of small microprocessors that implements a stack directly in hardware and some microcontrollers have a fixed-depth stack that is not directly accessible. Examples are the PIC microcontrollers, the Computer Cowboys MuP21, the Harris RTX line, and the Novix NC4016. Many stack-based microprocessors were used to implement the programming language Forth at the microcode level. Stacks were also used as a basis of a number of mainframes and mini computers. Such machines were called stack machines, the most famous being the Burroughs B5000.

2.3.5 Applications

Stacks are present everyday life, from the books in a library, to the blank sheets of paper in a printer tray. All these applications follow the *Last In First Out* (LIFO) logic, which means that (for example) a book is added on top of a pile of books, while removing a book from a pile also takes the book on top of a pile.

Below are a few applications of stacks in computing.

Expression evaluation and syntax parsing

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations and conversion from one form to another may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low

level code. Most programming languages are context-free languages, allowing them to be parsed with stack based machines.

Backtracking

Main article: [Backtracking](#)

Another important application of stacks is backtracking. Consider a simple example of finding the correct path in a maze. There are a series of points, from the starting point to the destination. We start from one point. To reach the final destination, there are several paths. Suppose we choose a random path. After following a certain path, we realise that the path we have chosen is wrong. So we need to find a way by which we can return to the beginning of that path. This can be done with the use of stacks. With the help of stacks, we remember the point where we have reached. This is done by pushing that point into the stack. In case we end up on the wrong path, we can pop the last point from the stack and thus return to the last point and continue our quest to find the right path. This is called backtracking.

Runtime memory management

Main articles: [Stack-based memory allocation](#) and [Stack machine](#)

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack. Many virtual machines are also stack-oriented, including the p-code machine and the Java Virtual Machine.

Almost all calling conventions—the ways in which subroutines receive their parameters and return results—use a special stack (the "call stack") to hold information about procedure/function calling and nesting in order to switch to the context of the called function and restore to the caller function when the calling finishes. The functions follow a runtime protocol between caller and callee to save arguments and return value on the stack. Stacks are an important way of supporting nested or recursive function calls. This type of stack is used implicitly by the compiler to support CALL and RETURN statements (or their equivalents) and is not manipulated directly by the programmer.

Some programming languages use the stack to store data that is local to a procedure. Space for local data items is allocated from the stack when the procedure is entered, and is deallocated when the procedure exits. The C pro-

gramming language is typically implemented in this way. Using the same stack for both data and procedure calls has important security implications (see below) of which a programmer must be aware in order to avoid introducing serious security bugs into a program.

2.3.6 Security

Some computing environments use stacks in ways that may make them vulnerable to security breaches and attacks. Programmers working in such environments must take special care to avoid the pitfalls of these implementations.

For example, some programming languages use a common stack to store both data local to a called procedure and the linking information that allows the procedure to return to its caller. This means that the program moves data into and out of the same stack that contains critical return addresses for the procedure calls. If data is moved to the wrong location on the stack, or an oversized data item is moved to a stack location that is not large enough to contain it, return information for procedure calls may be corrupted, causing the program to fail.

Malicious parties may attempt a **stack smashing** attack that takes advantage of this type of implementation by providing oversized data input to a program that does not check the length of input. Such a program may copy the data in its entirety to a location on the stack, and in so doing it may change the return addresses for procedures that have called it. An attacker can experiment to find a specific type of data that can be provided to such a program such that the return address of the current procedure is reset to point to an area within the stack itself (and within the data provided by the attacker), which in turn contains instructions that carry out unauthorized operations.

This type of attack is a variation on the **buffer overflow** attack and is an extremely frequent source of security breaches in software, mainly because some of the most popular compilers use a shared stack for both data and procedure calls, and do not verify the length of data items. Frequently programmers do not write code to verify the size of data items, either, and when an oversized or undersized data item is copied to the stack, a security breach may occur.

2.3.7 See also

- List of data structures
- Queue
- Double-ended queue
- Call stack
- FIFO (computing and electronics)
- Stack-based memory allocation

- Stack machine
- Stack overflow
- Stack-oriented programming language

2.3.8 References

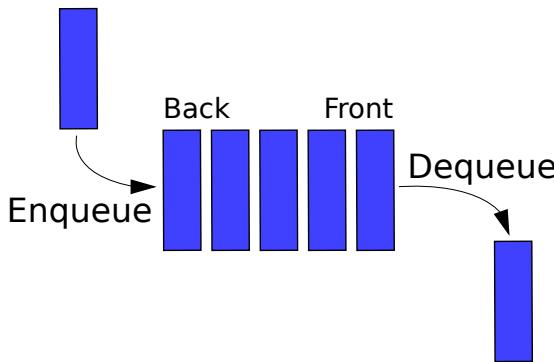
- [1] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.
- [2] <http://www.cprogramming.com/tutorial/computersciencetheory/stack.html> cprogramming.com
- [3] Newton, David E. (2003). *Alan Turing : a study in light and shadow*. [Philadelphia]: Xlibris. p. 82. ISBN 9781401090791. Retrieved 28 January 2015.
- [4] Dr. Friedrich Ludwig Bauer and Dr. Klaus Samelson (30 March 1957). “Verfahren zur automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens” (in German). Germany, Munich: Deutsches Patentamt. Retrieved 2010-10-01.
- [5] C. L. Hamblin, “An Addressless Coding Scheme based on Mathematical Notation”, N.S.W University of Technology, May 1957 (typescript)
- [6] Godse, A.P.; Godse, D.A. (January 1, 2010). *Computer Architecture*. Technical Publications. p. 1-56. ISBN 9788184315349. Retrieved January 30, 2015.
- [7] Horowitz, Ellis: “Fundamentals of Data Structures in Pascal”, page 67. Computer Science Press, 1984

2.3.9 Further reading

- Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.2.1: Stacks, Queues, and Deques, pp. 238–243.

2.3.10 External links

- Stacks and its Applications
- Stack Machines - the new wave
- Bounding stack depth
- Stack Size Analysis for Interrupt-driven Programs (322 KB)
- Black, Paul E. “Bounded stack”. *Dictionary of Algorithms and Data Structures*. NIST.



Representation of a **FIFO** (first in, first out) queue

2.4 Queue

In computer science, a **queue** (/'kju:/ **KEW**) is a particular kind of **abstract data type** or **collection** in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as *enqueue*, and removal of entities from the front terminal position, known as *dequeue*. This makes the queue a **First-In-First-Out (FIFO)** data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a *peek* or *front* operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a **linear data structure**, or more abstractly a **sequential collection**.

Queues provide services in computer science, transport, and **operations research** where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a **buffer**.

Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an **abstract data structure** or in object-oriented languages as classes. Common implementations are **circular buffers** and **linked lists**.

2.4.1 Queue implementation

Theoretically, one characteristic of a queue is that it does not have a specific capacity. Regardless of how many elements are already contained, a new element can always be added. It can also be empty, at which point removing an element will be impossible until a new element has been added again.

Fixed length arrays are limited in capacity, but it is not true that items need to be copied towards the head of the queue. The simple trick of turning the array into a closed circle and letting the head and tail drift around endlessly

in that circle makes it unnecessary to ever move items stored in the array. If n is the size of the array, then computing indices modulo n will turn the array into a circle. This is still the conceptually simplest way to construct a queue in a high level language, but it does admittedly slow things down a little, because the array indices must be compared to zero and the array size, which is comparable to the time taken to check whether an array index is out of bounds, which some languages do, but this will certainly be the method of choice for a quick and dirty implementation, or for any high level language that does not have pointer syntax. The array size must be declared ahead of time, but some implementations simply double the declared array size when overflow occurs. Most modern languages with objects or pointers can implement or come with libraries for dynamic lists. Such **data structures** may have not specified fixed capacity limit besides memory constraints. Queue *overflow* results from trying to add an element onto a full queue and queue *underflow* happens when trying to remove an element from an empty queue.

A *bounded queue* is a queue limited to a fixed number of items.^[1]

There are several efficient implementations of FIFO queues. An efficient implementation is one that can perform the operations—enqueueing and dequeuing—in $O(1)$ time.

- **Linked list**
 - A **doubly linked list** has $O(1)$ insertion and deletion at both ends, so is a natural choice for queues.
 - A regular singly linked list only has efficient insertion and deletion at one end. However, a small modification—keeping a pointer to the *last* node in addition to the first one—will enable it to implement an efficient queue.
- A **deque** implemented using a modified dynamic array

Queues and programming languages

Queues may be implemented as a separate data type, or may be considered a special case of a **double-ended queue** (**deque**) and not implemented separately. For example, **Perl** and **Ruby** allow pushing and popping an array from both ends, so one can use **push** and **shift** functions to enqueue and dequeue a list (or, in reverse, one can use **unshift** and **pop**), although in some cases these operations are not efficient.

C++'s Standard Template Library provides a “queue” templated class which is restricted to only push/pop operations. Since J2SE5.0, Java's library contains a **Queue** interface that specifies queue operations; implementing classes include **LinkedList** and (since J2SE 1.6)

ArrayDeque. PHP has an `SplQueue` class and third party libraries like `beanstalkd` and `Gearman`.

Examples

A simple queue implemented in Ruby:

```
class Queue
  def initialize
    @input = []
    @output = []
  end
  def enqueue(element)
    @input << element
  end
  def dequeue
    if @output.empty?
      while @input.any?
        @output << @input.pop
      end
    end
    @output.pop
  end
end
```

2.4.2 See also

- Circular buffer
- Deque
- Priority queue
- Queueing theory
- Stack – the “opposite” of a queue: **LIFO** (Last In First Out)

2.4.3 References

- [1] “Queue (Java Platform SE 7)”. Docs.oracle.com. 2014-03-26. Retrieved 2014-05-22.
- Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.2.1: Stacks, Queues, and Deques, pp. 238–243.
 - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 10.1: Stacks and queues, pp. 200–204.
 - William Ford, William Topp. *Data Structures with C++ and STL*, Second Edition. Prentice Hall, 2002. ISBN 0-13-085850-1. Chapter 8: Queues and Priority Queues, pp. 386–390.
 - Adam Drozdek. *Data Structures and Algorithms in C++*, Third Edition. Thomson Course Technology, 2005. ISBN 0-534-49182-0. Chapter 4: Stacks and Queues, pp. 137–169.

2.4.4 External links

- Queue Data Structure and Algorithm
- Queues with algo and 'c' programme

- STL Quick Reference
- VBScript implementation of stack, queue, deque, and Red-Black Tree

Black, Paul E. “Bounded queue”. *Dictionary of Algorithms and Data Structures*. NIST.

2.5 Deque

“Deque” redirects here. It is not to be confused with dequeuing, a **queue** operation.

Not to be confused with **Double-ended priority queue**.

In computer science, a **double-ended queue (deque)**, often abbreviated to **deque**, pronounced *deck*) is an abstract data type that generalizes a **queue**, for which elements can be added to or removed from either the front (head) or back (tail).^[1] It is also often called a **head-tail linked list**, though properly this refers to a specific data structure implementation (see below).

2.5.1 Naming conventions

Deque is sometimes written *dequeue*, but this use is generally deprecated in technical literature or technical writing because *dequeue* is also a verb meaning “to remove from a queue”. Nevertheless, several libraries and some writers, such as Aho, Hopcroft, and Ullman in their textbook *Data Structures and Algorithms*, spell it *dequeue*. John Mitchell, author of *Concepts in Programming Languages*, also uses this terminology.

2.5.2 Distinctions and sub-types

This differs from the queue abstract data type or First-In-First-Out List (**FIFO**), where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

- An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only.
- An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only.

Both the basic and most common list types in computing, **queues** and **stacks** can be considered specializations of deques, and can be implemented using deques.

2.5.3 Operations

The basic operations on a deque are *enqueue* and *dequeue* on either end. Also generally implemented are *peek* operations, which return the value at that end without dequeuing it.

Names vary between languages; major implementations include:

2.5.4 Implementations

There are at least two common ways to efficiently implement a deque: with a modified *dynamic array* or with a *doubly linked list*.

The dynamic array approach uses a variant of a *dynamic array* that can grow from both ends, sometimes called **array deques**. These array deques have all the properties of a dynamic array, such as constant-time *random access*, *good locality of reference*, and inefficient insertion/removal in the middle, with the addition of amortized constant-time insertion/removal at both ends, instead of just one end. Three common implementations include:

- Storing deque contents in a *circular buffer*, and only resizing when the buffer becomes full. This decreases the frequency of resizings.
- Allocating deque contents from the center of the underlying array, and resizing the underlying array when either end is reached. This approach may require more frequent resizings and waste more space, particularly when elements are only inserted at one end.
- Storing contents in multiple smaller arrays, allocating additional arrays at the beginning or end as needed. Indexing is implemented by keeping a dynamic array containing pointers to each of the smaller arrays.

2.5.5 Language support

Ada's containers provides the generic packages `Ada.Containers.Vectors` and `Ada.Containers.Doubly_Linked_Lists`, for the dynamic array and linked list implementations, respectively.

C++'s Standard Template Library provides the class templates `std::deque` and `std::list`, for the multiple array and linked list implementations, respectively.

As of Java 6, Java's Collections Framework provides a new *Deque* interface that provides the functionality of insertion and removal at both ends. It is implemented by classes such as `ArrayDeque` (also new in Java 6) and `LinkedList`, providing the dynamic array and linked list

implementations, respectively. However, the `ArrayDeque`, contrary to its name, does not support random access.

Python 2.4 introduced the `collections` module with support for *deque* objects. It is implemented using a doubly-linked list of fixed-length subarrays.

As of PHP 5.3, PHP's SPL extension contains the '`SplDoublyLinkedList`' class that can be used to implement Deque datastructures. Previously to make a Deque structure the array functions `array_shift`/`unshift`/`pop`/`push` had to be used instead.

GHC's `Data.Sequence` module implements an efficient, functional deque structure in Haskell. The implementation uses 2–3 finger trees annotated with sizes. There are other (fast) possibilities to implement purely functional (thus also *persistent*) double queues (most using heavily *lazy evaluation*).^{[2][3]} Kaplan and Tarjan were the first to implement optimal confluently persistent catenable deques.^[4] Their implementation was strictly purely functional in the sense that it did not use *lazy evaluation*. Okasaki simplified the data structure by using *lazy evaluation* with a bootstrapped data structure and degrading the performance bounds from worst-case to amortized. Kaplan, Okasaki, and Tarjan produced a simpler, non-bootstrapped, amortized version that can be implemented either using *lazy evaluation* or more efficiently using mutation in a broader but still restricted fashion. Mihaescu and Tarjan created a simpler (but still highly complex) strictly purely functional implementation of catenable deques, and also a much simpler implementation of strictly purely functional non-catenable deques, both of which have optimal worst-case bounds.

2.5.6 Complexity

- In a doubly linked list implementation and assuming no allocation/deallocation overhead, the *time complexity* of all deque operations is $O(1)$. Additionally, the time complexity of insertion or deletion in the middle, given an iterator, is $O(1)$; however, the time complexity of random access by index is $O(n)$.
- In a growing array, the amortized time complexity of all deque operations is $O(1)$. Additionally, the time complexity of random access by index is $O(1)$; but the time complexity of insertion or deletion in the middle is $O(n)$.

2.5.7 Applications

One example where a deque can be used is the *A-Steal* job scheduling algorithm.^[5] This algorithm implements task scheduling for several processors. A separate deque with threads to be executed is maintained for each processor. To execute the next thread, the processor gets the first element from the deque (using the “remove first element”

deque operation). If the current thread forks, it is put back to the front of the deque (“insert element at front”) and a new thread is executed. When one of the processors finishes execution of its own threads (i.e. its deque is empty), it can “steal” a thread from another processor: it gets the last element from the deque of another processor (“remove last element”) and executes it. The steal-job scheduling algorithm is used by Intel’s Threading Building Blocks (TBB) library for parallel programming.

2.5.8 See also

- Queue
- Priority queue

2.5.9 References

- [1] Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.2.1: Stacks, Queues, and Deques, pp. 238–243.
- [2] <http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf> C. Okasaki, “Purely Functional Data Structures”, September 1996
- [3] Adam L. Buchsbaum and Robert E. Tarjan. Confluently persistent deques via data structural bootstrapping. *Journal of Algorithms*, 18(3):513–547, May 1995. (pp. 58, 101, 125)
- [4] Haim Kaplan and Robert E. Tarjan. Purely functional representations of catenable sorted lists. In *ACM Symposium on Theory of Computing*, pages 202–211, May 1996. (pp. 4, 82, 84, 124)
- [5] Eitan Frachtenberg, Uwe Schwiegelshohn (2007). *Job Scheduling Strategies for Parallel Processing: 12th International Workshop, JSSPP 2006*. Springer. ISBN 3-540-71034-5. See p.22.

2.5.10 External links

- SGI STL Documentation: `deque<T, Alloc>`
- Code Project: An In-Depth Study of the STL Deque Container
- Deque implementation in C
- VBScript implementation of stack, queue, deque, and Red-Black Tree
- Multiple implementations of non-catenable deques in Haskell

2.6 Priority queue

In computer science, a **priority queue** is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a “priority” associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

While priority queues are often implemented with **heaps**, they are conceptually distinct from heaps. A priority queue is an abstract concept like “a list” or “a map”; just as a list can be implemented with a **linked list** or an **array**, a priority queue can be implemented with a heap or a variety of other methods such as an **unordered array**.

2.6.1 Operations

A priority queue must at least support the following operations:

- *insert_with_priority*: add an **element** to the **queue** with an associated **priority**.
- *pull_highest_priority_element*: remove the element from the queue that has the *highest priority*, and return it.

This is also known as “*pop_element(Off)*”, “*get_maximum_element*” or “*get_front(most)_element*”.

Some conventions reverse the order of priorities, considering lower values to be higher priority, so this may also be known as “*get_minimum_element*”, and is often referred to as “*get-min*” in the literature.

This may instead be specified as separate “*peek_at_highest_priority_element*” and “*delete_element*” functions, which can be combined to produce “*pull_highest_priority_element*”.

In addition, *peek* (in this context often called *find-max* or *find-min*), which returns the highest-priority element but does not modify the queue, is very frequently implemented, and nearly always executes in $O(1)$ time. This operation and its $O(1)$ performance is crucial to many applications of priority queues.

More advanced implementations may support more complicated operations, such as *pull_lowest_priority_element*, inspecting the first few highest- or lowest-priority elements, clearing the queue, clearing subsets of the queue, performing a batch insert, merging two or more queues into one, incrementing priority of any element, etc.

2.6.2 Similarity to queues

One can imagine a priority queue as a modified **queue**, but when one would get the next element off the queue, the highest-priority element is retrieved first.

Stacks and queues may be modeled as particular kinds of priority queues. As a reminder, here is how stacks and queues behave:

- *stack* – elements are pulled in last-in first-out-order (e.g., a stack of papers)
- *queue* – elements are pulled in first-in first-out-order (e.g., a line in a cafeteria)

In a stack, the priority of each inserted element is monotonically increasing; thus, the last element inserted is always the first retrieved. In a queue, the priority of each inserted element is monotonically decreasing; thus, the first element inserted is always the first retrieved.

2.6.3 Implementation

Naive implementations

There are a variety of simple, usually inefficient, ways to implement a priority queue. They provide an analogy to help one understand what a priority queue is. For instance, one can keep all the elements in an unsorted list. Whenever the highest-priority element is requested, search through all elements for the one with the highest priority. (In big O notation: $O(1)$ insertion time, $O(n)$ pull time due to search.)

Usual implementation

To improve performance, priority queues typically use a **heap** as their backbone, giving $O(\log n)$ performance for inserts and removals, and $O(n)$ to build initially. Variants of the basic heap data structure such as **pairing heaps** or **Fibonacci heaps** can provide better bounds for some operations.^[1]

Alternatively, when a **self-balancing binary search tree** is used, insertion and removal also take $O(\log n)$ time, although building trees from existing sequences of elements takes $O(n \log n)$ time; this is typical where one might already have access to these data structures, such as with third-party or standard libraries.

Note that from a computational-complexity standpoint, priority queues are congruent to sorting algorithms. See the next section for how efficient sorting algorithms can create efficient priority queues.

Specialized heaps

There are several specialized heap data structures that either supply additional operations or outperform heap-based implementations for specific types of keys, specifically integer keys.

- When the set of keys is $\{1, 2, \dots, C\}$, and only *insert*, *find-min* and *extract-min* are needed, a *bounded height priority queue* can be constructed as an array of C linked lists plus a pointer *top*, initially C . Inserting an item with key k appends the item to the k 'th, and updates $\text{top} \leftarrow \min(\text{top}, k)$, both in constant time. *Extract-min* deletes and returns one item from the list with index *top*, then increments *top* if needed until it again points to a non-empty list; this takes $O(C)$ time in the worst case. These queues are useful for sorting the vertices of a graph by their degree.^{[2]:374}
- For the set of keys $\{1, 2, \dots, C\}$, a **van Emde Boas tree** would support the *minimum*, *maximum*, *insert*, *delete*, *search*, *extract-min*, *extract-max*, *predecessor* and *successor* operations in $O(\log \log C)$ time, but has a space cost for small queues of about $O(2^{m/2})$, where m is the number of bits in the priority value.^[3]

- The **Fusion tree** algorithm by Fredman and Willard implements the *minimum* operation in $O(1)$ time and *insert* and *extract-min* operations in $O(\sqrt{\log n})$ time however it is stated by the author that, “Our algorithms have theoretical interest only; The constant factors involved in the execution times preclude practicality.”^[4]

For applications that do many “peek” operations for every “extract-min” operation, the time complexity for peek actions can be reduced to $O(1)$ in all tree and heap implementations by caching the highest priority element after every insertion and removal. For insertion, this adds at most a constant cost, since the newly inserted element is compared only to the previously cached minimum element. For deletion, this at most adds an additional “peek” cost, which is typically cheaper than the deletion cost, so overall time complexity is not significantly impacted.

Monotone priority queues are specialized queues that are optimized for the case where no item is ever inserted that has a lower priority (in the case of min-heap) than any item previously extracted. This restriction is met by several practical applications of priority queues.

2.6.4 Equivalence of priority queues and sorting algorithms

Using a priority queue to sort

The semantics of priority queues naturally suggest a sorting method: insert all the elements to be sorted into a

priority queue, and sequentially remove them; they will come out in sorted order. This is actually the procedure used by several [sorting algorithms](#), once the layer of abstraction provided by the priority queue is removed. This sorting method is equivalent to the following sorting algorithms:

Using a sorting algorithm to make a priority queue

A sorting algorithm can also be used to implement a priority queue. Specifically, Thorup says:^[5]

We present a general deterministic linear space reduction from priority queues to sorting implying that if we can sort up to n keys in $S(n)$ time per key, then there is a priority queue supporting *delete* and *insert* in $O(S(n))$ time and *find-min* in constant time.

That is, if there is a sorting algorithm which can sort in $O(S)$ time per key, where S is some function of n and [word size](#),^[6] then one can use the given procedure to create a priority queue where pulling the highest-priority element is $O(1)$ time, and inserting new elements (and deleting elements) is $O(S)$ time. For example, if one has an $O(n \log \log n)$ sort algorithm, one can create a priority queue with $O(1)$ pulling and $O(\log \log n)$ insertion.

2.6.5 Libraries

A priority queue is often considered to be a "container data structure".

The [Standard Template Library](#) (STL), and the [C++ 1998 standard](#), specifies `priority_queue` as one of the STL container adaptor class templates. It implements a max-priority-queue, and has three parameters: a comparison object for sorting such as a functor (defaults to `less<T>` if unspecified), the underlying container for storing the data structures (defaults to `std::vector<T>`), and two iterators to the beginning and end of a sequence. Unlike actual STL containers, it does not allow [iteration](#) of its elements (it strictly adheres to its abstract data type definition). STL also has utility functions for manipulating another random-access container as a binary max-heap. The [Boost \(C++ libraries\)](#) also have an implementation in the library heap.

Python's `heapq` module implements a binary min-heap on top of a list.

Java's library contains a `PriorityQueue` class, which implements a min-priority-queue.

Go's library contains a `container/heap` module, which implements a min-heap on top of any compatible data structure.

The [Standard PHP Library](#) extension contains the class `SplPriorityQueue`.

Apple's Core Foundation framework contains a `CFBinaryHeap` structure, which implements a min-heap.

2.6.6 Applications

Bandwidth management

Priority queuing can be used to manage limited resources such as [bandwidth](#) on a transmission line from a [network router](#). In the event of outgoing [traffic](#) queuing due to insufficient bandwidth, all other queues can be halted to send the traffic from the highest priority queue upon arrival. This ensures that the prioritized traffic (such as real-time traffic, e.g. an [RTP](#) stream of a [VoIP](#) connection) is forwarded with the least delay and the least likelihood of being rejected due to a queue reaching its maximum capacity. All other traffic can be handled when the highest priority queue is empty. Another approach used is to send disproportionately more traffic from higher priority queues.

Many modern protocols for [local area networks](#) also include the concept of priority queues at the [media access control](#) (MAC) sub-layer to ensure that high-priority applications (such as [VoIP](#) or [IPTV](#)) experience lower latency than other applications which can be served with [best effort](#) service. Examples include [IEEE 802.11e](#) (an amendment to [IEEE 802.11](#) which provides [quality of service](#)) and [ITU-T G.hn](#) (a standard for high-speed local area network using existing home wiring ([power lines](#), [phone lines](#) and [coaxial cables](#))).

Usually a limitation (policer) is set to limit the bandwidth that traffic from the highest priority queue can take, in order to prevent high priority packets from choking off all other traffic. This limit is usually never reached due to high level control instances such as the [Cisco Callmanager](#), which can be programmed to inhibit calls which would exceed the programmed bandwidth limit.

Discrete event simulation

Another use of a priority queue is to manage the events in a [discrete event simulation](#). The events are added to the queue with their simulation time used as the priority. The execution of the simulation proceeds by repeatedly pulling the top of the queue and executing the event thereon.

See also: [Scheduling \(computing\)](#), [queueing theory](#)

Dijkstra's algorithm

When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing [Dijkstra's algorithm](#), although one also needs the ability to alter the priority of a particular vertex in the priority queue efficiently.

Huffman coding

Huffman coding requires one to repeatedly obtain the two lowest-frequency trees. A priority queue makes this efficient.

Best-first search algorithms

Best-first search algorithms, like the A* search algorithm, find the shortest path between two vertices or nodes of a weighted graph, trying out the most promising routes first. A priority queue (also known as the *fringe*) is used to keep track of unexplored routes; the one for which the estimate (a lower bound in the case of A*) of the total path length is smallest is given highest priority. If memory limitations make best-first search impractical, variants like the SMA* algorithm can be used instead, with a double-ended priority queue to allow removal of low-priority items.

ROAM triangulation algorithm

The Real-time Optimally Adapting Meshes (ROAM) algorithm computes a dynamically changing triangulation of a terrain. It works by splitting triangles where more detail is needed and merging them where less detail is needed. The algorithm assigns each triangle in the terrain a priority, usually related to the error decrease if that triangle would be split. The algorithm uses two priority queues, one for triangles that can be split and another for triangles that can be merged. In each step the triangle from the split queue with the highest priority is split, or the triangle from the merge queue with the lowest priority is merged with its neighbours.

Prim's algorithm for minimum spanning tree

Using min heap priority queue in Prim's algorithm to find the minimum spanning tree of a connected and undirected graph, one can achieve a good running time. This min heap priority queue uses the min heap data structure which supports operations such as *insert*, *minimum*, *extract-min*, *decrease-key*.^[7] In this implementation, the weight of the edges is used to decide the priority of the vertices. Lower the weight, higher the priority and higher the weight, lower the priority.^[8]

2.6.7 See also

- Batch queue
- Command queue
- Job scheduler

2.6.8 References

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 20: Fibonacci Heaps, pp.476–497. Third edition p518.
- [2] Skiena, Steven (2010). *The Algorithm Design Manual* (2nd ed.). Springer Science+Business Media. ISBN 1-849-96720-2.
- [3] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84. IEEE Computer Society, 1975.
- [4] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 48(3):533–551, 1994
- [5] Thorup, Mikkel (2007). “Equivalence between priority queues and sorting”. *Journal of the ACM* **54** (6). doi:10.1145/1314690.1314692.
- [6] <http://courses.csail.mit.edu/6.851/spring07/scribe/lec17.pdf>
- [7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009). *INTRODUCTION TO ALGORITHMS* 3. MIT Press. p. 634. ISBN 978-81-203-4007-7. In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A. In the pseudo-code
- [8] “Prim's Algorithm”. Geek for Geeks. Retrieved 12 September 2014.

2.6.9 Further reading

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 6.5: Priority queues, pp. 138–142.

2.6.10 External links

- C++ reference for `std::priority_queue`
- Descriptions by Lee Killough
- PQlib - Open source Priority Queue library for C
- libpqueue is a generic priority queue (heap) implementation (in C) used by the Apache HTTP Server project.
- Survey of known priority queue structures by Stefan Xenos
- UC Berkeley - Computer Science 61B - Lecture 24: Priority Queues (video) - introduction to priority queues using binary heap

2.7 Map

“Dictionary (data structure)” redirects here. It is not to be confused with **data dictionary**.

In computer science, an **associative array**, **map**, **symbol table**, or **dictionary** is an abstract data type composed of a collection of $(key, value)$ pairs, such that each possible key appears just once in the collection.

Operations associated with this data type allow:^{[1][2]}

- the addition of pairs to the collection
- the removal of pairs from the collection
- the modification of the values of existing pairs
- the lookup of the value associated with a particular key

The **dictionary problem** is a classic computer science problem: the task of designing a **data structure** that maintains a set of data during ‘search’ ‘delete’ and ‘insert’ operations.^[3] A standard solution to the dictionary problem is a **hash table**; in some cases it is also possible to solve the problem using directly addressed **arrays**, **binary search trees**, or other more specialized structures.^{[1][2][4]}

Many programming languages include associative arrays as **primitive data types**, and they are available in **software libraries** for many others. **Content-addressable memory** is a form of direct hardware-level support for associative arrays.

Associative arrays have many applications including such fundamental **programming patterns** as **memoization** and the **decorator pattern**.^[5]

2.7.1 Operations

In an associative array, the association between a key and a value is often known as a “binding”, and the same word “binding” may also be used to refer to the process of creating a new association.

The operations that are usually defined for an associative array are:^{[1][2]}

- **Add or insert**: add a new $(key, value)$ pair to the collection, binding the new key to its new value. The arguments to this operation are the key and the value.
- **Reassign**: replace the value in one of the $(key, value)$ pairs that are already in the collection, binding an old key to a new value. As with an insertion, the arguments to this operation are the key and the value.

- **Remove or delete**: remove a $(key, value)$ pair from the collection, unbinding a given key from its value. The argument to this operation is the key.

- **Lookup**: find the value (if any) that is bound to a given key. The argument to this operation is the key, and the value is returned from the operation. If no value is found, some associative array implementations raise an exception.

In addition, associative arrays may also include other operations such as determining the number of bindings or constructing an **iterator** to loop over all the bindings. Usually, for such an operation, the order in which the bindings are returned may be arbitrary.

A **multimap** generalizes an associative array by allowing multiple values to be associated with a single key.^[6] A **bidirectional map** is a related abstract data type in which the bindings operate in both directions: each value must be associated with a unique key, and a second lookup operation takes a value as argument and looks up the key associated with that value.

2.7.2 Example

Suppose that the set of loans made by a library is to be represented in a data structure. Each book in a library may be checked out only by a single library patron at a time. However, a single patron may be able to check out multiple books. Therefore, the information about which books are checked out to which patrons may be represented by an associative array, in which the books are the keys and the patrons are the values. For instance (using notation from **Python**, or **JSON** (JavaScript Object Notation), in which a binding is represented by placing a colon between the key and the value), the current checkouts may be represented by an associative array

```
{ "Great Expectations": "John", "Pride and Prejudice": "Alice", "Wuthering Heights": "Alice" }
```

A lookup operation with the key “Great Expectations” in this array would return the name of the person who checked out that book, John. If John returns his book, that would cause a deletion operation in the associative array, and if Pat checks out another book, that would cause an insertion operation, leading to a different state:

```
{ "Pride and Prejudice": "Alice", "The Brothers Karamazov": "Pat", "Wuthering Heights": "Alice" }
```

In this new state, the same lookup as before, with the key “Great Expectations”, would raise an exception, because this key is no longer present in the array.

2.7.3 Implementation

For dictionaries with very small numbers of bindings, it may make sense to implement the dictionary using an association list, a [linked list](#) of bindings. With this implementation, the time to perform the basic dictionary operations is linear in the total number of bindings; however, it is easy to implement and the constant factors in its running time are small.^{[1][7]}

Another very simple implementation technique, usable when the keys are restricted to a narrow range of integers, is direct addressing into an array: the value for a given key k is stored at the array cell $A[k]$, or if there is no binding for k then the cell stores a special [sentinel](#) value that indicates the absence of a binding. As well as being simple, this technique is fast: each dictionary operation takes constant time. However, the space requirement for this structure is the size of the entire keyspace, making it impractical unless the keyspace is small.^[4]

The most frequently used general purpose implementation of an associative array is with a [hash table](#): an array of bindings, together with a [hash function](#) that maps each possible key into an array index. The basic idea of a hash table is that the binding for a given key is stored at the position given by applying the hash function to that key, and that lookup operations are performed by looking at that cell of the array and using the binding found there. However, hash table based dictionaries must be prepared to handle [collisions](#) that occur when two keys are mapped by the hash function to the same index, and many different collision resolution strategies have been developed for dealing with this situation, often based either on [open addressing](#) (looking at a sequence of hash table indices instead of a single index, until finding either the given key or an empty cell) or on [hash chaining](#) (storing a small association list instead of a single binding in each hash table cell).^{[1][2][4]}

Another common approach is to implement an associative array with a (self-balancing) [red-black tree](#).^[8]

Dictionaries may also be stored in [binary search trees](#) or in data structures specialized to a particular type of keys such as [radix trees](#), [tries](#), [Judy arrays](#), or [van Emde Boas trees](#), but these implementation methods are less efficient than hash tables as well as placing greater restrictions on the types of data that they can handle. The advantages of these alternative structures come from their ability to handle operations beyond the basic ones of an associative array, such as finding the binding whose key is the closest to a queried key, when the query is not itself present in the set of bindings.

2.7.4 Language support

Main article: [Comparison of programming languages \(mapping\)](#)

Associative arrays can be implemented in any programming language as a package and many language systems provide them as part of their standard library. In some languages, they are not only built into the standard system, but have special syntax, often using array-like subscripting.

Built-in syntactic support for associative arrays was introduced by [SNOBOL4](#), under the name “table”. [MUMPS](#) made multi-dimensional associative arrays, optionally persistent, its key data structure. [SETL](#) supported them as one possible implementation of sets and maps. Most modern scripting languages, starting with [AWK](#) and including [Rexx](#), [Perl](#), [Tcl](#), [JavaScript](#), [Python](#), [Ruby](#), and [Lua](#), support associative arrays as a primary container type. In many more languages, they are available as library functions without special syntax.

In [Smalltalk](#), [Objective-C](#), [.NET](#),^[9] [Python](#), [REALbasic](#), and [Swift](#) they are called *dictionaries*; in [Perl](#), [Ruby](#) and [Seed7](#) they are called *hashes*; in [C++](#), [Java](#), [Go](#), [Clojure](#), [Scala](#), [OCaml](#), [Haskell](#) they are called *maps* (see [map \(C++\)](#), [unordered_map \(C++\)](#), and [Map](#)); in [Common Lisp](#) and [Windows PowerShell](#), they are called *hash tables* (since both typically use this implementation). In [PHP](#), all arrays can be associative, except that the keys are limited to integers and strings. In [JavaScript](#) (see also [JSON](#)), all objects behave as associative arrays. In [Lua](#), they are called *tables*, and are used as the primitive building block for all data structures. In [Visual FoxPro](#), they are called *Collections*. The [D](#) language also has support for associative arrays^[10]

2.7.5 Permanent storage

Main article: [Key-value store](#)

Most programs using associative arrays will at some point need to store that data in a more permanent form, like in a [computer file](#). A common solution to this problem is a generalized concept known as [archiving](#) or [serialization](#), which produces a text or binary representation of the original objects that can be written directly to a file. This is most commonly implemented in the underlying object model, like [.Net](#) or [Cocoa](#), which include standard functions that convert the internal data into text form. The program can create a complete text representation of any group of objects by calling these methods, which are almost always already implemented in the base associative array class.^[11]

For programs that use very large data sets, this sort of individual file storage is not appropriate, and a [database management system](#) (DB) is required. Some DB systems natively store associative arrays by serializing the data and then storing that serialized data and the key. Individual arrays can then be loaded or saved from the database using the key to refer to them. These [key-value stores](#) have been used for many years and have a history as long as

that as the more common relational database (RDBs), but a lack of standardization, among other reasons, limited their use to certain niche roles. RDBs were used for these roles in most cases, although saving objects to a RDB can be complicated, a problem known as **object-relational impedance mismatch**.

After about 2010, the need for high performance databases suitable for **cloud computing** and more closely matching the internal structure of the programs using them led to a renaissance in the key-value store market. These systems can store and retrieve associative arrays in a native fashion, which can greatly improve performance in common web-related workflows.

2.7.6 See also

- Key-value database
- Tuple
- Function (mathematics)
- Key-value data store
- JSON

2.7.7 References

- [1] Goodrich, Michael T.; Tamassia, Roberto (2006), “9.1 The Map Abstract Data Type”, *Data Structures & Algorithms in Java* (4th ed.), Wiley, pp. 368–371.
- [2] Mehlhorn, Kurt; Sanders, Peter (2008), “4 Hash Tables and Associative Arrays”, *Algorithms and Data Structures: The Basic Toolbox*, Springer, pp. 81–98.
- [3] Anderson, Arne (1989). “Optimal Bounds on the Dictionary Problem”. *Proc. Symposium on Optimal Algorithms* (Springer Verlag): 106–114.
- [4] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), “11 Hash Tables”, *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 221–252, ISBN 0-262-03293-7 .
- [5] Goodrich & Tamassia (2006), pp. 597–599.
- [6] Goodrich & Tamassia (2006), pp. 389–397.
- [7] “When should I use a hash table instead of an association list?”. lisp-faq/part2. 1996-02-20.
- [8] Joel Adams and Larry Nyhoff. “Trees in STL”. quote: “The Standard Template library ... some of its containers -- the `set<T>`, `map<T1, T2>`, `multiset<T>`, and `multimap<T1, T2>` templates -- are generally built using a special kind of *self-balancing binary search tree* called a *red-black tree*.”
- [9] “Dictionary< TKey, TValue> Class”. MSDN.
- [10] “Associative Arrays, the D programming language”. Digital Mars.
- [11] “Archives and Serializations Programming Guide”, Apple Inc., 2012

2.7.8 External links

- NIST’s Dictionary of Algorithms and Data Structures: **Associative Array**

2.8 Bidirectional map

In computer science, a **bidirectional map**, or **hash bag**, is an associative data structure in which the $(key, value)$ pairs form a one-to-one correspondence. Thus the **binary relation** is **functional** in each direction: **value** can also act as a key to **key**. A pair (a, b) thus provides a **unique coupling** between a and b so that b can be found when a is used as a key and a can be found when b is used as a key.

2.8.1 External links

- Boost.org
- Commons.apache.org
- Cablemodem.fibertel.com.ar
- Codeproject.com
- Guava-libraries.googlecode.com

2.9 Multimap

This article is about the data type. For the mathematical concept, see **Multivalued function**. For the mapping website, see **Multimap.com**.

In computer science, a **multimap** (sometimes also **multihash**) is a generalization of a map or associative array abstract data type in which more than one value may be associated with and returned for a given key. Both map and multimap are particular cases of **containers** (for example, see **C++ Standard Template Library** containers). Often the multimap is implemented as a map with **lists** or **sets** as the map values.

2.9.1 Examples

- In a student enrollment system, where students may be enrolled in multiple classes simultaneously, there might be an association for each enrollment of a student in a course, where the key is the student ID and the value is the course ID. If a student is enrolled in three courses, there will be three associations containing the same key.
- The index of a book may report any number of references for a given index term, and thus may be

coded as a multimap from index terms to any number of reference locations.

- **Querystrings** may have multiple values associated with a single field. This is commonly generated when a **web form** allows multiple **check boxes** or selections to be chosen in response to a single form element.

2.9.2 Language support

C++

C++'s **Standard Template Library** provides the multimap container for the sorted multimap using a **self-balancing binary search tree**,^[1] and **SGI's STL extension** provides the **hash_multimap** container, which implements a multimap using a hash table.^[2]

Dart

Quiver provides a **Multimap** for Dart.^[3]

Java

Apache Commons Collections provides a **MultiMap** interface for Java.^[4] It also provides a **MultiValueMap** implementing class that makes a **MultiMap** out of a **Map** object and a type of **Collection**.^[5]

Google Guava provides an interface **Multimap** and implementations.^[6]

OCaml

OCaml's standard library module **Hashtbl** implements a hash table where it's possible to store multiple values for a key.

Scala

The **Scala** programming language's API also provides **Multimap** and implementations^[7]

2.9.3 See also

- Abstract data type for the concept of type in general
- **Associative array** for the more fundamental abstract data type
- **Multiset** for the case where same item can appear several times

2.9.4 References

- [1] “**multimap<Key, Data, Compare, Alloc>**”. *Standard Template Library Programmer’s Guide*. Silicon Graphics International.
- [2] “**hash_multimap<Key, HashFcn, EqualKey, Alloc>**”. *Standard Template Library Programmer’s Guide*. Silicon Graphics International.
- [3] “**Multimap**”. *Quiver API docs*.
- [4] “**Interface MultiMap**”. *Commons Collections 3.2.1 API*, Apache Commons.
- [5] “**Class MultiValueMap**”. *Commons Collections 3.2.1 API*, Apache Commons.
- [6] “**Interface Multimap<K,V>**”. *Guava Library 2.0*.
- [7] “**Scala.collection.mutable.Multimap**”. *Scala stable API*.

2.10 Set

In computer science, a **set** is an abstract data type that can store certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a **finite set**. Unlike most other collection types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set.

Some set data structures are designed for **static** or **frozen sets** that do not change after they are constructed. Static sets allow only query operations on their elements — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and deletion of elements from the set.

An abstract data structure is a collection, or aggregate, of data. The data may be booleans, numbers, characters, or other data structures. If one considers the structure yielded by **packaging**^[lower-alpha 1] or **indexing**,^[lower-alpha 2] there are four basic data structures:^{[1][2]}

1. unpackaged, unindexed: **bunch**
2. packaged, unindexed: **set**
3. unpackaged, indexed: **string (sequence)**
4. packaged, indexed: **list (array)**

In this view, the contents of a set are a **bunch**, and isolated data items are elementary bunches (elements). Whereas sets *contain* elements, bunches *consist of* elements.

Further structuring may be achieved by considering the multiplicity of elements (sets become multisets, bunches become hyperbunches)^[3] or their homogeneity (a record is a set of fields, not necessarily all of the same type).

2.10.1 Type theory

In type theory, sets are generally identified with their indicator function (characteristic function): accordingly, a set of values of type A may be denoted by 2^A or $\mathcal{P}(A)$. (Subtypes and subsets may be modeled by refinement types, and quotient sets may be replaced by setoids.) The characteristic function F of a set S is defined as:

$$F(x) = \begin{cases} 1, & \text{if } x \in S \\ 0, & \text{if } x \notin S \end{cases}$$

In theory, many other abstract data structures can be viewed as set structures with additional operations and/or additional axioms imposed on the standard operations. For example, an abstract **heap** can be viewed as a set structure with a $\text{min}(S)$ operation that returns the element of smallest value.

2.10.2 Operations

Core set-theoretical operations

One may define the operations of the algebra of sets:

- $\text{union}(S, T)$: returns the **union** of sets S and T .
- $\text{intersection}(S, T)$: returns the **intersection** of sets S and T .
- $\text{difference}(S, T)$: returns the **difference** of sets S and T .
- $\text{subset}(S, T)$: a predicate that tests whether the set S is a **subset** of set T .

Static sets

Typical operations that may be provided by a static set structure S are:

- $\text{is_element_of}(x, S)$: checks whether the value x is in the set S .
- $\text{is_empty}(S)$: checks whether the set S is empty.
- $\text{size}(S)$ or $\text{cardinality}(S)$: returns the number of elements in S .
- $\text{iterate}(S)$: returns a function that returns one more value of S at each call, in some arbitrary order.
- $\text{enumerate}(S)$: returns a list containing the elements of S in some arbitrary order.
- $\text{build}(x_1, x_2, \dots, x_n)$: creates a set structure with values x_1, x_2, \dots, x_n .
- $\text{create_from}(collection)$: creates a new set structure containing all the elements of the given collection or all the elements returned by the given **iterator**.

Dynamic sets

Dynamic set structures typically add:

- $\text{create}()$: creates a new, initially empty set structure.
- $\text{create_with_capacity}(n)$: creates a new set structure, initially empty but capable of holding up to n elements.
- $\text{add}(S, x)$: adds the element x to S , if it is not present already.
- $\text{remove}(S, x)$: removes the element x from S , if it is present.
- $\text{capacity}(S)$: returns the maximum number of values that S can hold.

Some set structures may allow only some of these operations. The cost of each operation will depend on the implementation, and possibly also on the particular values stored in the set, and the order in which they are inserted.

Additional operations

There are many other operations that can (in principle) be defined in terms of the above, such as:

- $\text{pop}(S)$: returns an arbitrary element of S , deleting it from S .^[4]
- $\text{pick}(S)$: returns an arbitrary element of S .^{[5][6][7]} Functionally, the mutator `pop` can be interpreted as the pair of selectors `(pick, rest)`, where `rest` returns the set consisting of all elements except for the arbitrary element.^[8] Can be interpreted in terms of `iterate`.^[lower-alpha 3]
- $\text{map}(F, S)$: returns the set of distinct values resulting from applying function F to each element of S .
- $\text{filter}(P, S)$: returns the subset containing all elements of S that satisfy a given predicate P .
- $\text{fold}(A_0, F, S)$: returns the value $A_1|S|$ after applying $A_{i+1} := F(A_i, e)$ for each element e of S , for some binary operation F . F must be associative and commutative for this to be well-defined.
- $\text{clear}(S)$: delete all elements of S .
- $\text{equal}(S_1, S_2)$: checks whether the two given sets are equal (i.e. contain all and only the same elements).
- $\text{hash}(S)$: returns a **hash value** for the static set S such that if $\text{equal}(S_1, S_2)$ then $\text{hash}(S_1) = \text{hash}(S_2)$

Other operations can be defined for sets with elements of a special type:

- $\text{sum}(S)$: returns the sum of all elements of S for some definition of “sum”. For example, over integers or reals, it may be defined as $\text{fold}(0, \text{add}, S)$.
- $\text{collapse}(S)$: given a set of sets, return the union.^[9] For example, $\text{collapse}(\{\{1\}, \{2, 3\}\}) == \{1, 2, 3\}$. May be considered a kind of sum.
- $\text{flatten}(S)$: given a set consisting of sets and atomic elements (elements that are not sets), returns a set whose elements are the atomic elements of the original top-level set or elements of the sets it contains. In other words, remove a level of nesting – like collapse, but allow atoms. This can be done a single time, or recursively flattening to obtain a set of only atomic elements.^[10] For example, $\text{flatten}(\{1, \{2, 3\}\}) == \{1, 2, 3\}$.
- $\text{nearest}(S, x)$: returns the element of S that is closest in value to x (by some metric).
- $\text{min}(S)$, $\text{max}(S)$: returns the minimum/maximum element of S .

2.10.3 Implementations

Sets can be implemented using various data structures, which provide different time and space trade-offs for various operations. Some implementations are designed to improve the efficiency of very specialized operations, such as nearest or union. Implementations described as “general use” typically strive to optimize the element_of, add, and delete operations. A simple implementation is to use a list, ignoring the order of the elements and taking care to avoid repeated values. This is simple but inefficient, as operations like set membership or element deletion are $O(n)$, as they require scanning the entire list.^[lower-alpha 4] Sets are often instead implemented using more efficient data structures, particularly various flavors of trees, tries, or hash tables.

As sets can be interpreted as a kind of map (by the indicator function), sets are commonly implemented in the same way as (partial) maps (associative arrays) – in this case in which the value of each key-value pair has the unit type or a sentinel value (like 1) – namely, a self-balancing binary search tree for sorted sets (which has $O(\log n)$ for most operations), or a hash table for unsorted sets (which has $O(1)$ average-case, but $O(n)$ worst-case, for most operations). A sorted linear hash table^[11] may be used to provide deterministically ordered sets.

Further, in languages that support maps but not sets, sets can be implemented in terms of maps. For example, a common programming idiom in Perl that converts an array to a hash whose values are the sentinel value 1, for use as a set, is:

```
my %elements = map { $_[0] => 1 } @elements;
```

Other popular methods include arrays. In particular a subset of the integers $1..n$ can be implemented efficiently as an n -bit bit array, which also support very efficient union and intersection operations. A Bloom map implements a set probabilistically, using a very compact representation but risking a small chance of false positives on queries.

The Boolean set operations can be implemented in terms of more elementary operations (pop, clear, and add), but specialized algorithms may yield lower asymptotic time bounds. If sets are implemented as sorted lists, for example, the naive algorithm for $\text{union}(S, T)$ will take time proportional to the length m of S times the length n of T ; whereas a variant of the list merging algorithm will do the job in time proportional to $m+n$. Moreover, there are specialized set data structures (such as the union-find data structure) that are optimized for one or more of these operations, at the expense of others.

2.10.4 Language support

One of the earliest languages to support sets was Pascal; many languages now include it, whether in the core language or in a standard library.

- In C++, the Standard Template Library (STL) provides the `set` template class, which is typically implemented using a binary search tree (e.g. red-black tree); SGI's STL also provides the `hash_set` template class, which implements a set using a hash table. In sets, the elements themselves are the keys, in contrast to sequenced containers, where elements are accessed using their (relative or absolute) position. Set elements must have a strict weak ordering.
- Java offers the `Set` interface to support sets (with the `HashSet` class implementing it using a hash table), and the `SortedSet` sub-interface to support sorted sets (with the `TreeSet` class implementing it using a binary search tree).
- Apple's Foundation framework (part of Cocoa) provides the Objective-C classes `NSSet`, `NSMutableSet`, `NSCountedSet`, `NSOrderedSet`, and `NSMutableOrderedSet`. The CoreFoundation APIs provide the `CFSet` and `CFMutableSet` types for use in C.
- Python has built-in `set` and `frozenset` types since 2.4, and since Python 3.0 and 2.7, supports non-empty set literals using a curly-bracket syntax, e.g.: `{x, y, z}`.
- The .NET Framework provides the generic `HashSet` and `SortedSet` classes that implement the generic `ISet` interface.
- Smalltalk's class library includes `Set` and `IdentitySet`, using equality and identity for inclusion test

respectively. Many dialects provide variations for compressed storage (NumberSet, CharacterSet), for ordering (OrderedSet, SortedSet, etc.) or for **weak references** (WeakIdentitySet).

- **Ruby**'s standard library includes a `set` module which contains `Set` and `SortedSet` classes that implement sets using hash tables, the latter allowing iteration in sorted order.
- **OCaml**'s standard library contains a `Set` module, which implements a functional set data structure using binary search trees.
- The **GHC** implementation of **Haskell** provides a `Data.Set` module, which implements immutable sets using binary search trees.^[12]
- The **Tcl Tclib** package provides a `set` module which implements a set data structure based upon **TCL** lists.
- The **Swift** standard library contains a `Set` type, since Swift 1.2.

As noted in the previous section, in languages which do not directly support sets but do support associative arrays, sets can be emulated using associative arrays, by using the elements as keys, and using a dummy value as the values, which are ignored.

2.10.5 Multiset

A generalization of the notion of a set is that of a **multiset** or **bag**, which is similar to a set but allows repeated (“equal”) values (duplicates). This is used in two distinct senses: either equal values are considered *identical*, and are simply counted, or equal values are considered *equivalent*, and are stored as distinct items. For example, given a list of people (by name) and ages (in years), one could construct a multiset of ages, which simply counts the number of people of a given age. Alternatively, one can construct a multiset of people, where two people are considered equivalent if their ages are the same (but may be different people and have different names), in which case each pair (name, age) must be stored, and selecting on a given age gives all the people of a given age.

Formally, it is possible for objects in computer science to be considered “equal” under some equivalence relation but still distinct under another relation. Some types of multiset implementations will store distinct equal objects as separate items in the data structure; while others will collapse it down to one version (the first one encountered) and keep a positive integer count of the multiplicity of the element.

As with sets, multisets can naturally be implemented using hash table or trees, which yield different performance characteristics.

The set of all bags over type `T` is given by the expression `bag T`. If by multiset one considers equal items identical and simply counts them, then a multiset can be interpreted as a function from the input domain to the non-negative integers (**natural numbers**), generalizing the identification of a set with its indicator function. In some cases a multiset in this counting sense may be generalized to allow negative values, as in Python.

- C++'s **Standard Template Library** implements both sorted and unsorted multisets. It provides the `multiset` class for the sorted multiset, as a kind of associative container, which implements this multiset using a **self-balancing binary search tree**. It provides the `unordered_multiset` class for the unsorted multiset, as a kind of **unordered associative containers**, which implements this multiset using a **hash table**. The unsorted multiset is standard as of **C++11**; previously SGI's STL provides the `hash_multiset` class, which was copied and eventually standardized.
- For Java, third-party libraries provide multiset functionality:
 - **Apache Commons Collections** provides the `Bag` and `SortedBag` interfaces, with implementing classes like `HashBag` and `TreeBag`.
 - **Google Guava** provides the `Multiset` interface, with implementing classes like `HashMultiset` and `TreeMultiset`.
- Apple provides the `NSCountedSet` class as part of **Cocoa**, and the `CFBag` and `CFMutableBag` types as part of **CoreFoundation**.
- Python's standard library includes `collections.Counter`, which is similar to a multiset.
- **Smalltalk** includes the `Bag` class, which can be instantiated to use either identity or equality as predicate for inclusion test.

Where a multiset data structure is not available, a workaround is to use a regular set, but override the equality predicate of its items to always return “not equal” on distinct objects (however, such will still not be able to store multiple occurrences of the same object) or use an **associative array** mapping the values to their integer multiplicities (this will not be able to distinguish between equal elements at all).

Typical operations on bags:

- `contains(B , x)`: checks whether the element x is present (at least once) in the bag B
- `is_sub_bag(B_1 , B_2)`: checks whether each element in the bag B_1 occurs in B_1 no more often than it occurs in the bag B_2 ; sometimes denoted as $B_1 \sqsubseteq B_2$.

- $\text{count}(B, x)$: returns the number of times that the element x occurs in the bag B ; sometimes denoted as $B \# x$.
- $\text{scaled_by}(B, n)$: given a natural number n , returns a bag which contains the same elements as the bag B , except that every element that occurs m times in B occurs $n * m$ times in the resulting bag; sometimes denoted as $n \otimes B$.
- $\text{union}(B_1, B_2)$: returns a bag that containing just those values that occur in either the bag B_1 or the bag B_2 , except that the number of times a value x occurs in the resulting bag is equal to $(B_1 \# x) + (B_2 \# x)$; sometimes denoted as $B_1 \uplus B_2$.

Multisets in SQL

In relational databases, a table can be a (mathematical) set or a multiset, depending on the presence on unicity constraints on some columns (which turns it into a candidate key).

SQL allows the selection of rows from a relational table: this operation will in general yield a multiset, unless the keyword DISTINCT is used to force the rows to be all different, or the selection includes the primary (or a candidate) key.

In ANSI SQL the MULTISET keyword can be used to transform a subquery into a collection expression:

```
SELECT expression1, expression2...    FROM TABLE_NAME...
```

is a general select that can be used as *subquery expression* of another more general query, while

```
MULTISET(SELECT expression1, expression2...
          FROM TABLE_NAME...)
```

transforms the subquery into a *collection expression* that can be used in another query, or in assignment to a column of appropriate collection type.

2.10.6 See also

- Bloom filter
- Disjoint set

2.10.7 Notes

- [1] “Packaging” consists in supplying a container for an aggregation of objects in order to turn them into a single object. Consider a function call: without packaging, a function can be called to act upon a bunch only by passing each bunch element as a separate argument, which complicates the function’s signature considerably (and is just

not possible in some programming languages). By packaging the bunch’s elements into a set, the function may now be called upon a single, elementary argument: the set object (the bunch’s package).

- [2] Indexing is possible when the elements being considered are **totally ordered**. Being without order, the elements of a multiset (for example) do not have lesser/greater or preceding/succeeding relationships: they can only be compared in absolute terms (same/different).
- [3] For example, in Python `pick` can be implemented on a derived class of the built-in set as follows:
- ```
class Set(set): def pick(self): return next(iter(self))
```
- [4] Element insertion can be done in  $O(1)$  time by simply inserting at an end, but if one avoids duplicates this takes  $O(n)$  time.

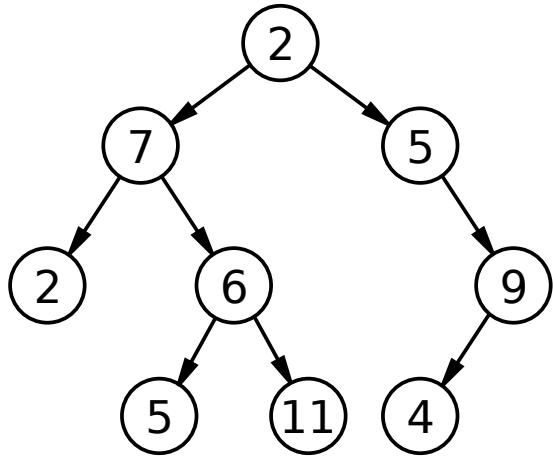
## 2.10.8 References

- [1] Hehner, Eric C. R. (1981), “Bunch Theory: A Simple Set Theory for Computer Science”, *Information Processing Letters* **12** (1): 26, doi:10.1016/0020-0190(81)90071-5
- [2] Hehner, Eric C. R. (2004), *A Practical Theory of Programming*, second edition
- [3] Hehner, Eric C. R. (2012), *A Practical Theory of Programming*, 2012-3-30 edition
- [4] Python: `pop()`
- [5] *Management and Processing of Complex Data Structures: Third Workshop on Information Systems and Artificial Intelligence, Hamburg, Germany, February 28 - March 2, 1994. Proceedings*, ed. Kai v. Luck, Heinz Marburger, p. 76
- [6] Python Issue7212: Retrieve an arbitrary element from a set without removing it; see [msg106593](#) regarding standard name
- [7] Ruby Feature #4553: Add `Set#pick` and `Set#pop`
- [8] *Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, Ute Schmid, Springer, Aug 21, 2003, p. 240
- [9] *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, S. Margherita, Italy, May 30 - June 3, 1994. Selected Papers, Volume 10*, ed. Egidio Astesiano, Gianna Reggio, Andrzej Tarlecki, p. 38
- [10] Ruby: `flatten()`
- [11] Wang, Thomas (1997), *Sorted Linear Hash Table*
- [12] Stephen Adams, "Efficient sets: a balancing act", *Journal of Functional Programming* 3(4):553-562, October 1993. Retrieved on 2015-03-11.

## 2.11 Tree

Not to be confused with **trie**, a specific type of tree data structure.

In computer science, a **tree** is a widely used **abstract data**



*A simple unordered tree; in this diagram, the node labeled 7 has two children, labeled 2 and 6, and one parent, labeled 2. The root node, at the top, has no parent.*

**type** (ADT) or **data structure** implementing this ADT that simulates a **hierarchical tree structure**, with a root value and subtrees of children with a parent node, represented as a set of **linked nodes**.

A tree data structure can be defined recursively (locally) as a collection of **nodes** (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the “children”), with the constraints that no reference is duplicated, and none points to the root.

Alternatively, a tree can be defined abstractly as a whole (globally) as an **ordered tree**, with a value assigned to each node. Both these perspectives are useful: while a tree can be analyzed mathematically as a whole, when actually represented as a data structure it is usually represented and worked with separately by node (rather than as a list of nodes and an **adjacency list** of edges between nodes, as one may represent a **digraph**, for instance). For example, looking at a tree as a whole, one can talk about “the parent node” of a given node, but in general as a data structure a given node only contains the list of its children, but does not contain a reference to its parent (if any).

### 2.11.1 Definition

A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

### 2.11.2 Terminologies used in Trees

- **Root** – The top node in a tree.
- **Parent** – The converse notion of *child*.
- **Siblings** – Nodes with the same parent.
- **Descendant** – a node reachable by repeated proceeding from parent to child.
- **Ancestor** – a node reachable by repeated proceeding from child to parent.
- **Leaf** – a node with no children.
- **Internal node** – a node with at least one child.
- **External node** – a node with no children.
- **Degree** – number of sub trees of a node.
- **Edge** – connection between one node to another.
- **Path** – a sequence of nodes and edges connecting a node with a descendant.
- **Level** – The level of a node is defined by  $1 +$  the number of connections between the node and the root.
- **Height of tree** – The height of a tree is the number of edges on the longest downward path between the root and a leaf.
- **Height of node** – The height of a node is the number of edges on the longest downward path between that node and a leaf.
- **Depth** – The depth of a node is the number of edges from the node to the tree’s root node.
- **Forest** – A forest is a set of  $n \geq 0$  disjoint trees.

### Data type vs. data structure

There is a distinction between a tree as an abstract data type and as a concrete data structure, analogous to the distinction between a **list** and a **linked list**. As a data type, a tree has a value and children, and the children are themselves trees; the value and children of the tree are interpreted as the value of the root node and the subtrees of the children of the root node. To allow finite trees, one must either allow the list of children to be empty (in which case trees can be required to be non-empty, an “empty tree” instead being represented by a forest of zero trees), or allow trees to be empty, in which case the list of children can be of fixed size (branching factor, especially 2 or “binary”), if desired.

As a data structure, a linked tree is a group of **nodes**, where each node has a value and a list of **references** to other nodes (its children). This data structure actually defines a **directed graph**,<sup>[lower-alpha 1]</sup> because it may have

loops or several references to the same node, just as a linked list may have a loop. Thus there is also the requirement that no two references point to the same node (that each node has at most a single parent, and in fact exactly one parent, except for the root), and a tree that violates this is “corrupt”.

Due to the use of *references* to trees in the linked tree data structure, trees are often discussed implicitly assuming that they are being represented by references to the root node, as this is often how they are actually implemented. For example, rather than an empty tree, one may have a null reference: a tree is always non-empty, but a reference to a tree may be null.

## Recursive

Recursively, as a data type a tree is defined as a value (of some data type, possibly empty), together with a list of trees (possibly an empty list), the subtrees of its children; symbolically:

$t: v [t[1], \dots, t[k]]$

(A tree  $t$  consists of a value  $v$  and a list of other trees.)

More elegantly, via **mutual recursion**, of which a tree is one of the most basic examples, a tree can be defined in terms of a forest (a list of trees), where a tree consists of a value and a forest (the subtrees of its children):

$f: [t[1], \dots, t[k]] t: v f$

Note that this definition is in terms of values, and is appropriate in **functional languages** (it assumes **referential transparency**); different trees have no connections, as they are simply lists of values.

As a data structure, a tree is defined as a node (the root), which itself consists of a value (of some data type, possibly empty), together with a list of references to other nodes (list possibly empty, references possibly null); symbolically:

$n: v [\&n[1], \dots, \&n[k]]$

(A node  $n$  consists of a value  $v$  and a list of references to other nodes.)

This data structure defines a **directed graph**,<sup>[lower-alpha 2]</sup> and for it to be a tree one must add a condition on its global structure (its **topology**), namely that at most one reference can point to any given node (a node has at most a single parent), and no node in the tree point to the root. In fact, every node (other than the root) must have exactly one parent, and the root must have no parents.

Indeed, given a list of nodes, and for each node a list of references to its children, one cannot tell if this structure is a tree or not without analyzing its global structure and that it is in fact topologically a tree, as defined below.

## Type theory

As an ADT, the abstract tree type  $T$  with values of some type  $E$  is defined, using the abstract forest type  $F$  (list of trees), by the functions:

value:  $T \rightarrow E$

children:  $T \rightarrow F$

nil:  $() \rightarrow F$

node:  $E \times F \rightarrow T$

with the axioms:

value(node( $e, f$ )) =  $e$

children(node( $e, f$ )) =  $f$

In terms of **type theory**, a tree is an inductive type defined by the constructors *nil* (empty forest) and *node* (tree with root node with given value and children).

## Mathematical

Viewed as a whole, a tree data structure is an **ordered tree**, generally with values attached to each node. Concretely, it is (if required to be non-empty):

- A rooted tree with the “away from root” direction (a more narrow term is an “**arborescence**”), meaning:
  - A directed graph,
  - whose underlying **undirected graph** is a **tree** (any two vertices are connected by exactly one simple path),
  - with a distinguished root (one vertex is designated as the root),
  - which determines the direction on the edges (arrows point away from the root; given an edge, the node that the edge points from is called the *parent* and the node that the edge points to is called the *child*),

together with:

- an ordering on the child nodes of a given node, and
- a value (of some data type) at each node.

Often trees have a fixed (more properly, bounded) **branching factor** (**outdegree**), particularly always having two child nodes (possibly empty, hence *at most two non-empty* child nodes), hence a “**binary tree**”.

Allowing empty trees makes some definitions simpler, some more complicated: a rooted tree must be non-empty, hence if empty trees are allowed the above definition instead becomes “an empty tree, or a rooted tree such

that ...". On the other hand, empty trees simplify defining fixed branching factor: with empty trees allowed, a binary tree is a tree such that every node has exactly two children, each of which is a tree (possibly empty). The complete sets of operations on tree must include fork operation.

### 2.11.3 Terminology

A **node** is a structure which may contain a value or condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more **child nodes**, which are below it in the tree (by convention, trees are drawn growing downwards). A node that has a child is called the child's **parent node** (or *ancestor node*, or *superior*). A node has at most one parent.

An **internal node** (also known as an **inner node**, **inode** for short, or **branch node**) is any node of a tree that has child nodes. Similarly, an **external node** (also known as an **outer node**, **leaf node**, or **terminal node**) is any node that does not have child nodes.

The topmost node in a tree is called the **root node**. Depending on definition, a tree may be required to have a root node (in which case all trees are non-empty), or may be allowed to be empty, in which case it does not necessarily have a root node. Being the topmost node, the root node will not have a parent. It is the node at which algorithms on the tree begin, since as a data structure, one can only pass from parents to children. Note that some algorithms (such as post-order depth-first search) begin at the root, but first visit leaf nodes (access the value of leaf nodes), only visit the root last (i.e., they first access the children of the root, but only access the *value* of the root last). All other nodes can be reached from it by following **edges** or **links**. (In the formal definition, each such path is also unique.) In diagrams, the root node is conventionally drawn at the top. In some trees, such as **heaps**, the root node has special properties. Every node in a tree can be seen as the root node of the subtree rooted at that node.

The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The **depth** of a node is the length of the path to its root (i.e., its *root path*). This is commonly needed in the manipulation of the various self-balancing trees, **AVL Trees** in particular. The root node has depth zero, leaf nodes have height zero, and a tree with only a single node (hence both a root and leaf) has depth and height zero. Conventionally, an empty tree (tree with no nodes, if such are allowed) has depth and height  $-1$ .

A **subtree** of a tree  $T$  is a tree consisting of a node in  $T$  and all of its descendants in  $T$ .<sup>[lower-alpha 3][1]</sup> Nodes thus correspond to subtrees (each node corresponds to the subtree of itself and all its descendants) – the subtree corresponding to the root node is the entire tree, and each node is the root node of the subtree it determines; the subtree

corresponding to any other node is called a **proper subtree** (by analogy to a proper subset).

### 2.11.4 Drawing Trees

Trees are often drawn in the plane. Ordered trees can be represented essentially uniquely in the plane, and are hence called *plane trees*, as follows: if one fixes a conventional order (say, counterclockwise), and arranges the child nodes in that order (first incoming parent edge, then first child edge, etc.), this yields an embedding of the tree in the plane, unique up to **ambient isotopy**. Conversely, such an embedding determines an ordering of the child nodes.

If one places the root at the top (parents above children, as in a **family tree**) and places all nodes that are a given distance from the root (in terms of number of edges: the "level" of a tree) on a given horizontal line, one obtains a standard drawing of the tree. Given a binary tree, the first child is on the left (the "left node"), and the second child is on the right (the "right node").

### 2.11.5 Representations

There are many different ways to represent trees; common representations represent the nodes as **dynamically allocated records** with pointers to their children, their parents, or both, or as items in an **array**, with relationships between them determined by their positions in the array (e.g., **binary heap**).

Indeed, a binary tree can be implemented as a list of lists (a list where the values are lists): the head of a list (the value of the first term) is the left child (subtree), while the tail (the list of second and future terms) is the right child (subtree). This can be modified to allow values as well, as in Lisp **S-expressions**, where the head (value of first term) is the value of the node, the head of the tail (value of second term) is the left child, and the tail of the tail (list of third and future terms) is the right child.

In general a node in a tree will not have pointers to its parents, but this information can be included (expanding the data structure to also include a pointer to the parent) or stored separately. Alternatively, upward links can be included in the child node data, as in a **threaded binary tree**.

### 2.11.6 Generalizations

#### Digraphs

If edges (to child nodes) are thought of as references, then a tree is a special case of a digraph, and the tree data structure can be generalized to represent **directed graphs** by removing the constraints that a node may have at most one parent, and that no cycles are allowed. Edges are

still abstractly considered as pairs of nodes, however, the terms *parent* and *child* are usually replaced by different terminology (for example, *source* and *target*). Different implementation strategies exist: a digraph can be represented by the same local data structure as a tree (node with value and list of children), assuming that “list of children” is a list of references, or globally by such structures as **adjacency lists**.

In graph theory, a tree is a connected acyclic graph; unless stated otherwise, in graph theory trees and graphs are assumed undirected. There is no one-to-one correspondence between such trees and trees as data structure. We can take an arbitrary undirected tree, arbitrarily pick one of its vertices as the *root*, make all its edges directed by making them point away from the root node – producing an **arborescence** – and assign an order to all the nodes. The result corresponds to a tree data structure. Picking a different root or different ordering produces a different one.

Given a node in a tree, its children define an ordered forest (the union of subtrees given by all the children, or equivalently taking the subtree given by the node itself and erasing the root). Just as subtrees are natural for recursion (as in a depth-first search), forests are natural for **corecursion** (as in a breadth-first search).

Via **mutual recursion**, a forest can be defined as a list of trees (represented by root nodes), where a node (of a tree) consists of a value and a forest (its children):

f: [n[1], ..., n[k]] n: v f

## 2.11.7 Traversal methods

Main article: Tree traversal

Stepping through the items of a tree, by means of the connections between parents and children, is called **walking the tree**, and the action is a **walk** of the tree. Often, an operation might be performed when a pointer arrives at a particular node. A walk in which each parent node is traversed before its children is called a **pre-order** walk; a walk in which the children are traversed before their respective parents are traversed is called a **post-order** walk; a walk in which a node’s left subtree, then the node itself, and finally its right subtree are traversed is called an **in-order** traversal. (This last scenario, referring to exactly two subtrees, a left subtree and a right subtree, assumes specifically a **binary tree**.) A **level-order** walk effectively performs a **breadth-first search** over the entirety of a tree; nodes are traversed level by level, where the root node is visited first, followed by its direct child nodes and their siblings, followed by its grandchild nodes and their siblings, etc., until all nodes in the tree have been traversed.

## 2.11.8 Common operations

- Enumerating all the items
- Enumerating a section of a tree
- Searching for an item
- Adding a new item at a certain position on the tree
- Deleting an item
- **Pruning**: Removing a whole section of a tree
- **Grafting**: Adding a whole section to a tree
- Finding the root for any node

## 2.11.9 Common uses

- Representing hierarchical data
- Storing data in a way that makes it easily **searchable** (see **binary search tree** and **tree traversal**)
- Representing sorted lists of data
- As a workflow for **compositing** digital images for **visual effects**
- **Routing** algorithms

## 2.11.10 See also

- **Tree structure**
- **Tree (graph theory)**
- **Tree (set theory)**
- **Hierarchy (mathematics)**
- **Dialog tree**
- **Single inheritance**

## Other trees

- **Trie**
- **DSW algorithm**
- **Enfilade**
- **Left child-right sibling binary tree**
- **Hierarchical temporal memory**

### 2.11.11 Notes

- [1] Properly, a rooted, ordered directed graph.
- [2] Properly, a rooted, ordered directed graph.
- [3] This is different from the formal definition of subtree used in graph theory, which is a subgraph that forms a tree – it need not include all descendants. For example, the root node by itself is a subtree in the graph theory sense, but not in the data structure sense (unless there are no descendants).

### 2.11.12 References

- [1] Weisstein, Eric W., “Subtree”, *MathWorld*.
- Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4 . Section 2.3: Trees, pp. 308–423.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7 . Section 10.4: Representing rooted trees, pp. 214–217. Chapters 12–14 (Binary Search Trees, Red-Black Trees, Augmenting Data Structures), pp. 253–320.

### 2.11.13 External links

- Data Trees as a Means of Presenting Complex Data Analysis by Sally Knipe
- Description from the Dictionary of Algorithms and Data Structures
- STL-like C++ tree class
- Description of tree data structures from ideainfo.8m.com
- WormWeb.org: Interactive Visualization of the *C. elegans* Cell Tree – Visualize the entire cell lineage tree of the nematode *C. elegans* (javascript)
- *Binary Trees* by L. Allison

# Chapter 3

## Arrays

### 3.1 Array data structure

Not to be confused with Array data type.

In computer science, an **array data structure** or simply an **array** is a data structure consisting of a collection of *elements* (values or variables), each identified by at least one *array index* or *key*. An array is stored so that the position of each element can be computed from its index *tuple* by a mathematical formula.<sup>[1][2][3]</sup> The simplest type of data structure is a linear array, also called one-dimensional array.

For example, an array of 10 32-bit integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, ... 2036, so that the element with index  $i$  has the address  $2000 + 4 \times i$ .<sup>[4]</sup>

Because the mathematical concept of a matrix can be represented as a two-dimensional grid, two-dimensional arrays are also sometimes called matrices. In some cases the term “vector” is used in computing to refer to an array, although *tuples* rather than *vectors* are more correctly the mathematical equivalent. Arrays are often used to implement *tables*, especially *lookup tables*; the word *table* is sometimes used as a synonym of *array*.

Arrays are among the oldest and most important data structures, and are used by almost every program. They are also used to implement many other data structures, such as *lists* and *strings*. They effectively exploit the addressing logic of computers. In most modern computers and many *external storage* devices, the memory is a one-dimensional array of words, whose indices are their addresses. Processors, especially *vector processors*, are often optimized for array operations.

Arrays are useful mostly because the element indices can be computed at *run time*. Among other things, this feature allows a single iterative *statement* to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation. The set of valid index tuples and the addresses of the elements (and hence the element addressing formula) are usually,<sup>[3][5]</sup> but not always,<sup>[2]</sup> fixed while the array is

in use.

The term *array* is often used to mean array data type, a kind of data type provided by most *high-level programming languages* that consists of a collection of values or variables that can be selected by one or more indices computed at run-time. Array types are often implemented by array structures; however, in some languages they may be implemented by *hash tables*, *linked lists*, *search trees*, or other data structures.

The term is also used, especially in the description of algorithms, to mean *associative array* or “*abstract array*”, a *theoretical computer science model* (an *abstract data type* or *ADT*) intended to capture the essential properties of arrays.

#### 3.1.1 History

The first digital computers used machine-language programming to set up and access array structures for data tables, vector and matrix computations, and for many other purposes. Von Neumann wrote the first array-sorting program (merge sort) in 1945, during the building of the first stored-program computer.<sup>[6]</sup><sup>[p. 159]</sup> Array indexing was originally done by self-modifying code, and later using *index registers* and indirect addressing. Some mainframes designed in the 1960s, such as the *Burroughs B5000* and its successors, used *memory segmentation* to perform index-bounds checking in hardware.<sup>[7]</sup>

Assembly languages generally have no special support for arrays, other than what the machine itself provides. The earliest high-level programming languages, including *FORTRAN* (1957), *COBOL* (1960), and *ALGOL 60* (1960), had support for multi-dimensional arrays, and so has *C* (1972). In *C++* (1983), class templates exist for multi-dimensional arrays whose dimension is fixed at runtime<sup>[3][5]</sup> as well as for runtime-flexible arrays.<sup>[2]</sup>

#### 3.1.2 Applications

Arrays are used to implement mathematical *vectors* and *matrices*, as well as other kinds of rectangular tables. Many *databases*, small and large, consist of (or include)

one-dimensional arrays whose elements are records.

Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks, strings, and VLists.

One or more large arrays are sometimes used to emulate in-program **dynamic memory allocation**, particularly **memory pool** allocation. Historically, this has sometimes been the only way to allocate “dynamic memory” portably.

Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple IF statements. They are known in this context as **control tables** and are used in conjunction with a purpose built interpreter whose control flow is altered according to values contained in the array. The array may contain **subroutine pointers** (or relative subroutine numbers that can be acted upon by **SWITCH** statements) that direct the path of the execution.

### 3.1.3 Element identifier and addressing formulas

When data objects are stored in an array, individual objects are selected by an index that is usually a non-negative **scalar integer**. Indices are also called subscripts. An index *maps* the array value to a stored object.

There are three ways in which the elements of an array can be indexed:

- **0 (zero-based indexing)**: The first element of the array is indexed by subscript of 0.<sup>[8]</sup>
- **1 (one-based indexing)**: The first element of the array is indexed by subscript of 1.<sup>[9]</sup>
- **n (n-based indexing)**: The base index of an array can be freely chosen. Usually programming languages allowing *n-based indexing* also allow negative index values and other **scalar** data types like **enumerations**, or **characters** may be used as an array index.

Arrays can have multiple dimensions, thus it is not uncommon to access an array using multiple indices. For example a two-dimensional array A with three rows and four columns might provide access to the element at the 2nd row and 4th column by the expression A[1, 3] (in a **row major** language) or A[3, 1] (in a **column major** language) in the case of a zero-based indexing system. Thus two indices are used for a two-dimensional array, three for a three-dimensional array, and *n* for an *n*-dimensional array.

The number of indices needed to specify an element is called the dimension, dimensionality, or **rank** of the array.

In standard arrays, each index is restricted to a certain range of consecutive integers (or consecutive values of

some **enumerated type**), and the address of an element is computed by a “linear” formula on the indices.

### One-dimensional arrays

A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

As an example consider the C declaration `int anArrayName[10];`

Syntax : `datatype anArrayname[sizeofArray];`

In the given example the array can contain 10 elements of any value available to the `int` type. In C, the array element indices are 0-9 inclusive in this case. For example, the expressions `anArrayName[0]` and `anArrayName[9]` are the first and last elements respectively.

For a vector with linear addressing, the element with index *i* is located at the address  $B + c \times i$ , where *B* is a fixed **base address** and *c* a fixed constant, sometimes called the **address increment** or **stride**.

If the valid element indices begin at 0, the constant *B* is simply the address of the first element of the array. For this reason, the **C programming language** specifies that array indices always begin at 0; and many programmers will call that element “zeroth” rather than “first”.

However, one can choose the index of the first element by an appropriate choice of the base address *B*. For example, if the array has five elements, indexed 1 through 5, and the base address *B* is replaced by *B* + 30*c*, then the indices of those same elements will be 31 to 35. If the numbering does not start at 0, the constant *B* may not be the address of any element.

### Multidimensional arrays

For a two-dimensional array, the element with indices *i, j* would have address  $B + c \cdot i + d \cdot j$ , where the coefficients *c* and *d* are the **row** and **column address increments**, respectively.

More generally, in a *k*-dimensional array, the address of an element with indices  $i_1, i_2, \dots, i_k$  is

$$B + c_1 \cdot i_1 + c_2 \cdot i_2 + \dots + c_k \cdot i_k.$$

For example: `int a[3][2];`

This means that array *a* has 3 rows and 2 columns, and the array is of integer type. Here we can store 6 elements they are stored linearly but starting from first row linear then continuing with second row. The above array will be stored as *a*<sub>11</sub>, *a*<sub>12</sub>, *a*<sub>21</sub>, *a*<sub>22</sub>, *a*<sub>31</sub>, *a*<sub>32</sub>.

This formula requires only *k* multiplications and *k* additions, for any array that can fit in memory. Moreover, if

any coefficient is a fixed power of 2, the multiplication can be replaced by **bit shifting**.

The coefficients  $ck$  must be chosen so that every valid index tuple maps to the address of a distinct element.

If the minimum legal value for every index is 0, then  $B$  is the address of the element whose indices are all zero. As in the one-dimensional case, the element indices may be changed by changing the base address  $B$ . Thus, if a two-dimensional array has rows and columns indexed from 1 to 10 and 1 to 20, respectively, then replacing  $B$  by  $B + c_1 - 3c_2$  will cause them to be renumbered from 0 through 9 and 4 through 23, respectively. Taking advantage of this feature, some languages (like FORTRAN 77) specify that array indices begin at 1, as in mathematical tradition; while other languages (like Fortran 90, Pascal and Algol) let the user choose the minimum value for each index.

## Dope vectors

The addressing formula is completely defined by the dimension  $d$ , the base address  $B$ , and the increments  $c_1, c_2, \dots, c_d$ . It is often useful to pack these parameters into a record called the array's *descriptor* or *stride vector* or *dope vector*.<sup>[2][3]</sup> The size of each element, and the minimum and maximum values allowed for each index may also be included in the dope vector. The dope vector is a complete **handle** for the array, and is a convenient way to pass arrays as arguments to procedures. Many useful array **slicing** operations (such as selecting a sub-array, swapping indices, or reversing the direction of the indices) can be performed very efficiently by manipulating the dope vector.<sup>[2]</sup>

## Compact layouts

Often the coefficients are chosen so that the elements occupy a contiguous area of memory. However, that is not necessary. Even if arrays are always created with contiguous elements, some array slicing operations may create non-contiguous sub-arrays from them.

There are two systematic compact layouts for a two-dimensional array. For example, consider the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

In the **row-major order** layout (adopted by C for statically declared arrays), the elements in each row are stored in consecutive positions and all of the elements of a row have a lower address than any of the elements of a consecutive row:

In **column-major order** (traditionally used by Fortran), the elements in each column are consecutive in memory and

all of the elements of a column have a lower address than any of the elements of a consecutive column:

For arrays with three or more indices, “row major order” puts in consecutive positions any two elements whose index tuples differ only by one in the *last* index. “Column major order” is analogous with respect to the *first* index.

In systems which use **processor cache** or **virtual memory**, scanning an array is much faster if successive elements are stored in consecutive positions in memory, rather than sparsely scattered. Many algorithms that use multidimensional arrays will scan them in a predictable order. A programmer (or a sophisticated compiler) may use this information to choose between row- or column-major layout for each array. For example, when computing the product  $A \cdot B$  of two matrices, it would be best to have  $A$  stored in row-major order, and  $B$  in column-major order.

## Resizing

Main article: [Dynamic array](#)

Static arrays have a size that is fixed when they are created and consequently do not allow elements to be inserted or removed. However, by allocating a new array and copying the contents of the old array to it, it is possible to effectively implement a *dynamic* version of an array; see [dynamic array](#). If this operation is done infrequently, insertions at the end of the array require only amortized constant time.

Some array data structures do not reallocate storage, but do store a count of the number of elements of the array in use, called the *count* or *size*. This effectively makes the array a **dynamic array** with a fixed maximum size or capacity; **Pascal strings** are examples of this.

## Non-linear formulas

More complicated (non-linear) formulas are occasionally used. For a compact two-dimensional **triangular array**, for instance, the addressing formula is a polynomial of degree 2.

### 3.1.4 Efficiency

Both *store* and *select* take (deterministic worst case) constant time. Arrays take linear ( $O(n)$ ) space in the number of elements  $n$  that they hold.

In an array with element size  $k$  and on a machine with a cache line size of  $B$  bytes, iterating through an array of  $n$  elements requires the minimum of  $\lceil nk/B \rceil$  cache misses, because its elements occupy contiguous memory locations. This is roughly a factor of  $B/k$  better than the number of cache misses needed to access  $n$  elements at random memory locations. As a consequence, sequen-

tial iteration over an array is noticeably faster in practice than iteration over many other data structures, a property called **locality of reference** (this does *not* mean however, that using a **perfect hash** or **trivial hash** within the same (local) array, will not be even faster - and achievable in **constant time**). Libraries provide low-level optimized facilities for copying ranges of memory (such as `memcpy`) which can be used to move **contiguous** blocks of array elements significantly faster than can be achieved through individual element access. The speedup of such optimized routines varies by array element size, architecture, and implementation.

Memory-wise, arrays are compact data structures with no per-element overhead. There may be a per-array overhead, e.g. to store index bounds, but this is language-dependent. It can also happen that elements stored in an array require *less* memory than the same elements stored in individual variables, because several array elements can be stored in a single **word**; such arrays are often called **packed** arrays. An extreme (but commonly used) case is the **bit array**, where every bit represents a single element. A single **octet** can thus hold up to 256 different combinations of up to 8 different conditions, in the most compact form.

Array accesses with statically predictable access patterns are a major source of **data parallelism**.

### Comparison with other data structures

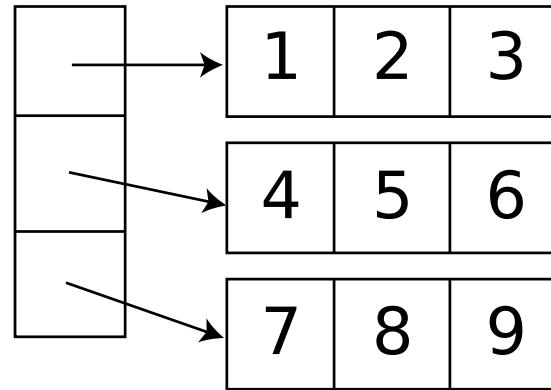
**Growable arrays** are similar to arrays but add the ability to insert and delete elements; adding and deleting at the end is particularly efficient. However, they reserve linear ( $\Theta(n)$ ) additional storage, whereas arrays do not reserve additional storage.

**Associative arrays** provide a mechanism for array-like functionality without huge storage overheads when the index values are sparse. For example, an array that contains values only at indexes 1 and 2 billion may benefit from using such a structure. Specialized associative arrays with integer keys include **Patricia tries**, **Judy arrays**, and **van Emde Boas trees**.

**Balanced trees** require  $O(\log n)$  time for indexed access, but also permit inserting or deleting elements in  $O(\log n)$  time,<sup>[14]</sup> whereas growable arrays require linear ( $\Theta(n)$ ) time to insert or delete elements at an arbitrary position.

**Linked lists** allow constant time removal and insertion in the middle but take linear time for indexed access. Their memory use is typically worse than arrays, but is still linear.

An **Iliffe vector** is an alternative to a multidimensional array structure. It uses a one-dimensional array of **references** to arrays of one dimension less. For two dimensions, in particular, this alternative structure would be a vector of pointers to vectors, one for each row. Thus an element in row  $i$  and column  $j$  of an array  $A$  would



A two-dimensional array stored as a one-dimensional array of one-dimensional arrays (rows).

be accessed by double indexing ( $A[i][j]$  in typical notation). This alternative structure allows **jagged arrays**, where each row may have a different size — or, in general, where the valid range of each index depends on the values of all preceding indices. It also saves one multiplication (by the column address increment) replacing it by a bit shift (to index the vector of row pointers) and one extra memory access (fetching the row address), which may be worthwhile in some architectures.

### 3.1.5 Dimension

The dimension of an array is the number of indices needed to select an element. Thus, if the array is seen as a function on a set of possible index combinations, it is the dimension of the space of which its domain is a discrete subset. Thus a one-dimensional array is a list of data, a two-dimensional array a rectangle of data, a three-dimensional array a block of data, etc.

This should not be confused with the dimension of the set of all matrices with a given domain, that is, the number of elements in the array. For example, an array with 5 rows and 4 columns is two-dimensional, but such matrices form a 20-dimensional space. Similarly, a three-dimensional vector can be represented by a one-dimensional array of size three.

### 3.1.6 See also

- Dynamic array
- Parallel array
- Variable-length array
- Bit array
- Array slicing
- Offset (computer science)
- Row-major order

- Stride of an array

### 3.1.7 References

- [1] Black, Paul E. (13 November 2008). “array”. *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. Retrieved 22 August 2010.
- [2] Bjoern Andres; Ullrich Koethe; Thorben Kroeger; Hamprecht (2010). “Runtime-Flexible Multi-dimensional Arrays and Views for C++98 and C++0x”. arXiv:1008.2909 [cs.DS].
- [3] Garcia, Ronald; Lumsdaine, Andrew (2005). “Multi-Array: a C++ library for generic programming with arrays”. *Software: Practice and Experience* 35 (2): 159–188. doi:10.1002/spe.630. ISSN 0038-0644.
- [4] David R. Richardson (2002), The Book on Data Structures. iUniverse, 112 pages. ISBN 0-595-24039-9, ISBN 978-0-595-24039-5.
- [5] T. Veldhuizen. Arrays in Blitz++. In Proc. of the 2nd Int. Conf. on Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), LNCS 1505, pages 223-220. Springer, 1998.
- [6] Donald Knuth, *The Art of Computer Programming*, vol. 3. Addison-Wesley
- [7] Levy, Henry M. (1984), *Capability-based Computer Systems*, Digital Press, p. 22, ISBN 9780932376220.
- [8] “Array Code Examples - PHP Array Functions - PHP code”. <http://www.configure-all.com/>: Computer Programming Web programming Tips. Retrieved 8 April 2011. In most computer languages array index (counting) starts from 0, not from 1. Index of the first element of the array is 0, index of the second element of the array is 1, and so on. In array of names below you can see indexes and values.
- [9] “Chapter 6 - Arrays, Types, and Constants”. *Modula-2 Tutorial*. <http://www.modula2.org/tutor/index.php>. Retrieved 8 April 2011. The names of the twelve variables are given by Automobiles[1], Automobiles[2], ... Automobiles[12]. The variable name is “Automobiles” and the array subscripts are the numbers 1 through 12. [i.e. in Modula-2, the index starts by one!]
- [10] Gerald Kruse. CS 240 Lecture Notes: Linked Lists Plus: Complexity Trade-offs. Juniata College. Spring 2008.
- [11] *Day 1 Keynote - Bjarne Stroustrup: C++11 Style at GoingNative 2012* on [channel9.msdn.com](http://channel9.msdn.com) from minute 45 or foil 44
- [12] *Number crunching: Why you should never, ever, EVER use linked-list in your code again* at [kjellkod.wordpress.com](http://kjellkod.wordpress.com)
- [13] Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (1999), *Resizable Arrays in Optimal Time and Space (Technical Report CS-99-09)* (PDF), Department of Computer Science, University of Waterloo
- [14] Counted B-Tree

### 3.1.8 External links

<https://www.tutorialcup.com/cplusplus/arrays.htm>

## 3.2 Row-major order

In computing, **row-major order** and **column-major order** describe methods for arranging multidimensional arrays in linear storage such as **memory**.

The difference is simply that in row-major order, consecutive elements of the rows of the array are contiguous in memory; in column-major order, consecutive elements of the columns are contiguous.

Array layout is critical for correctly passing arrays between programs written in different languages. It is also important for performance when traversing an array because accessing array elements that are contiguous in memory is usually faster than accessing elements which are not, due to **caching**.<sup>[1]</sup> In some media such as tape or **flash memory**, accessing sequentially is orders of magnitude faster than nonsequential access.

### 3.2.1 Explanation and example

Following conventional **matrix notation**, rows are numbered by the first index of a two-dimensional array and columns by the second index, i.e.,  $a[1,2]$  is the second element of the first row, counting downwards and rightwards. (Note this is the opposite of **Cartesian conventions**.)

The difference between row-major and column-major order is simply that the order of the dimensions is reversed. Equivalently, in row-major order the rightmost indices vary faster as one steps through consecutive memory locations, while in column-major order the leftmost indices vary faster.

This array

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$$

Would be stored as follows in the two orders:

### 3.2.2 Programming Languages

Programming languages which support multi-dimensional arrays have a native storage order for these arrays.

**Row-major order** is used in C/C++, Mathematica, PL/I, Pascal, Python, Speakeasy, SAS and others.

**Column-major order** is used in Fortran, OpenGL and OpenGL ES, MATLAB,<sup>[2]</sup> GNU Octave, S-Plus,<sup>[3]</sup> R,<sup>[4]</sup> Julia, Rasdaman, and Scilab.

### 3.2.3 Transposition

As exchanging the indices of an array is the essence of **array transposition**, an array stored as row-major but read as column-major (or vice versa) will appear transposed. As actually performing this **rearrangement in memory** is typically an expensive operation, some systems provide options to specify individual matrices as being stored transposed.

For example, the **Basic Linear Algebra Subprograms** functions are passed flags indicating which arrays are transposed.<sup>[5]</sup>

### 3.2.4 Address calculation in general

The concept trivially generalizes to arrays with more than two dimensions.

For a  $d$ -dimensional  $N_1 \times N_2 \times \dots \times N_d$  array with dimensions  $Nk$  ( $k=1\dots d$ ). A given element of this array is specified by a **tuple**  $(n_1, n_2, \dots, n_d)$  of  $d$  (zero-based) indices  $n_k \in [0, N_k - 1]$ .

In **row-major order**, the *last* dimension is contiguous, so that the memory-offset of this element is given by:

$$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\dots + N_{d-1} n_d) \dots))) = \sum_{k=1}^d \left( \prod_{\ell=1}^{k-1} N_\ell \right) n_k$$

In **column-major order**, the *first* dimension is contiguous, so that the memory-offset of this element is given by:

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\dots + N_2 n_1) \dots)))$$

### 3.2.5 See also

- **Matrix representation**
- **Vectorization (mathematics)**, the equivalent of turning a matrix into the corresponding column-major vector.

### 3.2.6 References

- [1] [https://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/fortran/lin/optaps/fortran/optaps\\_prg\\_arra\\_f.htm](https://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/fortran/lin/optaps/fortran/optaps_prg_arra_f.htm)
- [2] MATLAB documentation, MATLAB Data Storage (retrieved from Mathworks.co.uk, January 2014).
- [3] Spiegelhalter et al. (2003, p. 17): Spiegelhalter, David; Thomas, Andrew; Best, Nicky; Lunn, Dave (January 2003), “Formatting of data: S-Plus format”, *WinBUGS*

*User Manual* (Version 1.4 ed.), Robinson Way, Cambridge CB2 2SR, UK: MRC Biostatistics Unit, Institute of Public Health, [PDF document](#)

[4] *An Introduction to R*, Section 5.1: Arrays (retrieved March 2010).

[5] <http://www.netlib.org/blas/>

- Donald E. Knuth, *The Art of Computer Programming Volume 1: Fundamental Algorithms*, third edition, section 2.2.6 (Addison-Wesley: New York, 1997).

## 3.3 Dope vector

In computer programming, a **dope vector** is a data structure used to hold information about a **data object**,<sup>[1]</sup> e.g. an array, especially its **memory layout**.

A dope vector typically contains information about the type of array element, **rank of an array**, the **extents of an array**, and the **stride of an array** as well as a pointer to the block in memory containing the array elements.

It is often used in compilers to pass entire arrays between **procedures** in a **high level language** like **Fortran**.

The dope vector includes an identifier, a length, a parent address, and a next child address. The identifier was an assigned name and was mostly useless, but the length was the amount of allocated storage to this vector from the end of the dope vector that contained data of use to the internal processes of the computer. This length by many was called the offset, span of vector length. The parent and child references were absolute core references, or register settings to the parent or child depending on the type of computer.

Dope vectors were managed internally by the operating system and allowed the processor to allocate and deallocate storage in specific segments as needed.

Later dope vectors had a status bit that told the system if they were active; if it was not active it would be reallocated when needed. Using this technology the computer could perform a more granular memory management.

### 3.3.1 See also

Data descriptor

### 3.3.2 References

- [1] Pratt T. and M. Zelkowitz, *Programming Languages: Design and Implementation* (Third Edition), Prentice Hall, Upper Saddle River, NJ, (1996) pp 114

## 3.4 Iliffe vector

In computer programming, an **Iliffe vector**, also known as a **display**, is a data structure used to implement multi-dimensional arrays. An Iliffe vector for an  $n$ -dimensional array (where  $n \geq 2$ ) consists of a vector (or 1-dimensional array) of pointers to an  $(n - 1)$ -dimensional array. They are often used to avoid the need for expensive multiplication operations when performing address calculation on an array element. They can also be used to implement triangular arrays, or other kinds of irregularly shaped arrays. The data structure is named after John K. Iliffe.

Their disadvantages include the need for multiple chained pointer indirections to access an element, and the extra work required to determine the next row in an  $n$ -dimensional array to allow an optimising compiler to prefetch it. Both of these are a source of delays on systems where the CPU is significantly faster than main memory.

The Iliffe vector for a 2-dimensional array is simply a vector of pointers to vectors of data, i.e., the Iliffe vector represents the columns of an array where each column element is a pointer to a row vector.

Multidimensional arrays in languages such as Java, Python (multidimensional lists), Ruby, Visual Basic .NET, Perl, PHP, JavaScript, Objective-C, Swift, and Atlas Autocode are implemented as Iliffe vectors.

Iliffe vectors are contrasted with **dope vectors** in languages such as Fortran, which contain the stride factors and offset values for the subscripts in each dimension.

### 3.4.1 See also

- Jagged array

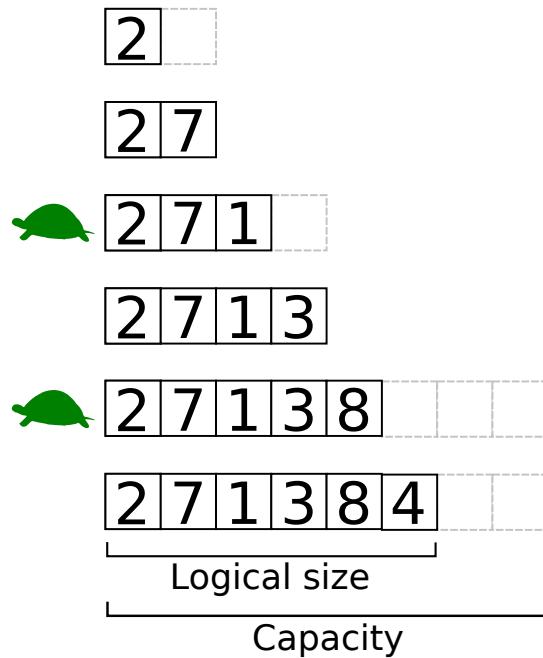
### 3.4.2 References

- John K. Iliffe (1961). “The Use of The Genie System in Numerical Calculations”. *Annual Review in Automatic Programming* 2: 25. doi:10.1016/S0066-4138(61)80002-5.

## 3.5 Dynamic array

In computer science, a **dynamic array**, **growable array**, **resizable array**, **dynamic table**, **mutable array**, or **array list** is a random access, variable-size list data structure that allows elements to be added or removed. It is supplied with standard libraries in many modern mainstream programming languages.

A dynamic array is not the same thing as a **dynamically allocated array**, which is an **array** whose size is fixed when



Several values are inserted at the end of a dynamic array using geometric expansion. Grey cells indicate space reserved for expansion. Most insertions are fast (constant time), while some are slow due to the need for reallocation ( $\Theta(n)$  time, labelled with turtles). The logical size and capacity of the final array are shown.

the array is allocated, although a dynamic array may use such a fixed-size array as a back end.<sup>[1]</sup>

### 3.5.1 Bounded-size dynamic arrays and capacity

The simplest dynamic array is constructed by allocating a fixed-size array and then dividing it into two parts: the first stores the elements of the dynamic array and the second is reserved, or unused. We can then add or remove elements at the end of the dynamic array in constant time by using the reserved space, until this space is completely consumed. The number of elements used by the dynamic array contents is its *logical size* or *size*, while the size of the underlying array is called the dynamic array's *capacity* or *physical size*, which is the maximum possible size without relocating data.<sup>[2]</sup>

In applications where the logical size is bounded, the fixed-size data structure suffices. This may be short-sighted, as more space may be needed later. A philosophical programmer may prefer to write the code to make every array capable of resizing from the outset, then return to using fixed-size arrays during **program optimization**. Resizing the underlying array is an expensive task, typically involving copying the entire contents of the array.

### 3.5.2 Geometric expansion and amortized cost

To avoid incurring the cost of resizing many times, dynamic arrays resize by a large amount, such as doubling in size, and use the reserved space for future expansion. The operation of adding an element to the end might work as follows:

```
function insertEnd(dynarray a, element e) if (a.size = a.capacity) // resize a to twice its current capacity:
a.capacity ← a.capacity * 2 // (copy the contents to the
new memory location here) a[a.size] ← e a.size ← a.size
+ 1
```

As  $n$  elements are inserted, the capacities form a **geometric progression**. Expanding the array by any constant proportion ensures that inserting  $n$  elements takes  $O(n)$  time overall, meaning that each insertion takes **amortized constant time**. The value of this proportion  $a$  leads to a time-space tradeoff: the average time per insertion operation is about  $a/(a-1)$ , while the number of wasted cells is bounded above by  $(a-1)n$ . The choice of  $a$  depends on the library or application: some textbooks use  $a = 2$ ,<sup>[3][4]</sup> but Java's `ArrayList` implementation uses  $a = 3/2$ <sup>[1]</sup> and the C implementation of Python's list data structure uses  $a = 9/8$ .<sup>[5]</sup>

Many dynamic arrays also deallocate some of the underlying storage if its size drops below a certain threshold, such as 30% of the capacity. This threshold must be strictly smaller than  $1/a$  in order to provide **hysteresis** (provide a stable band to avoiding repeatedly growing and shrinking) and support mixed sequences of insertions and removals with amortized constant cost.

Dynamic arrays are a common example when teaching amortized analysis.<sup>[3][4]</sup>

### 3.5.3 Performance

The dynamic array has performance similar to an array, with the addition of new operations to add and remove elements:

- Getting or setting the value at a particular index (constant time)
- Iterating over the elements in order (linear time, good cache performance)
- Inserting or deleting an element in the middle of the array (linear time)
- Inserting or deleting an element at the end of the array (constant amortized time)

Dynamic arrays benefit from many of the advantages of arrays, including good **locality of reference** and **data**

**cache utilization**, **compactness** (low memory use), and **random access**. They usually have only a small fixed additional overhead for storing information about the size and capacity. This makes dynamic arrays an attractive tool for building cache-friendly data structures. However, in languages like Python or Java that enforce reference semantics, the dynamic array generally will not store the actual data, but rather it will store references to the data that resides in other areas of memory. In this case, accessing items in the array sequentially will actually involve accessing multiple non-contiguous areas of memory, so the many advantages of the cache-friendliness of this data structure are lost.

Compared to **linked lists**, dynamic arrays have faster indexing (constant time versus linear time) and typically faster iteration due to improved locality of reference; however, dynamic arrays require linear time to insert or delete at an arbitrary location, since all following elements must be moved, while linked lists can do this in constant time. This disadvantage is mitigated by the **gap buffer** and **tiered vector** variants discussed under *Variants* below. Also, in a highly **fragmented** memory region, it may be expensive or impossible to find contiguous space for a large dynamic array, whereas linked lists do not require the whole data structure to be stored contiguously.

A **balanced tree** can store a list while providing all operations of both dynamic arrays and linked lists reasonably efficiently, but both insertion at the end and iteration over the list are slower than for a dynamic array, in theory and in practice, due to non-contiguous storage and tree traversal/manipulation overhead.

### 3.5.4 Variants

**Gap buffers** are similar to dynamic arrays but allow efficient insertion and deletion operations clustered near the same arbitrary location. Some **deque** implementations use **array deques**, which allow amortized constant time insertion/removal at both ends, instead of just one end.

Goodrich<sup>[10]</sup> presented a dynamic array algorithm called *Tiered Vectors* that provided  $O(n^{1/2})$  performance for order preserving insertions or deletions from the middle of the array.

**Hashed Array Tree** (HAT) is a dynamic array algorithm published by Sitarski in 1996.<sup>[11]</sup> Hashed Array Tree wastes order  $n^{1/2}$  amount of storage space, where  $n$  is the number of elements in the array. The algorithm has  $O(1)$  amortized performance when appending a series of objects to the end of a Hashed Array Tree.

In a 1999 paper,<sup>[9]</sup> Brodnik et al. describe a tiered dynamic array data structure, which wastes only  $n^{1/2}$  space for  $n$  elements at any point in time, and they prove a lower bound showing that any dynamic array must waste this much space if the operations are to remain amortized constant time. Additionally, they present a variant where

growing and shrinking the buffer has not only amortized but worst-case constant time.

Bagwell (2002)<sup>[12]</sup> presented the **VList** algorithm, which can be adapted to implement a dynamic array.

### 3.5.5 Language support

C++'s `std::vector` is an implementation of dynamic arrays, as are the `ArrayList`<sup>[13]</sup> classes supplied with the Java API and the .NET Framework. The generic `List<T>` class supplied with version 2.0 of the .NET Framework is also implemented with dynamic arrays. Smalltalk's `OrderedCollection` is a dynamic array with dynamic start and end-index, making the removal of the first element also  $O(1)$ . Python's list datatype implementation is a dynamic array. Delphi and D implement dynamic arrays at the language's core. Ada's `Ada.Containers.Vectors` generic package provides dynamic array implementation for a given subtype. Many scripting languages such as Perl and Ruby offer dynamic arrays as a built-in primitive data type. Several cross-platform frameworks provide dynamic array implementations for C, including `CFArray` and `CFMutableArray` in Core Foundation, and `GArray` and `GPtrArray` in GLib.

### 3.5.6 References

- [1] See, for example, the source code of `java.util.ArrayList` class from OpenJDK 6.
- [2] Lambert, Kenneth Alfred (2009), “Physical size and logical size”, *Fundamentals of Python: From First Programs Through Data Structures* (Cengage Learning): 510, ISBN 1423902181
- [3] Goodrich, Michael T.; Tamassia, Roberto (2002), “1.5.2 Analyzing an Extendable Array Implementation”, *Algorithm Design: Foundations, Analysis and Internet Examples*, Wiley, pp. 39–41.
- [4] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. “17.4 Dynamic tables”. *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. pp. 416–424. ISBN 0-262-03293-7.
- [5] List object implementation from [python.org](http://python.org), retrieved 2011-09-27.
- [6] Gerald Kruse. CS 240 Lecture Notes: Linked Lists Plus: Complexity Trade-offs. Juniata College. Spring 2008.
- [7] *Day 1 Keynote - Bjarne Stroustrup: C++11 Style at GoingNative 2012* on [channel9.msdn.com](http://channel9.msdn.com) from minute 45 or foil 44
- [8] *Number crunching: Why you should never, ever, EVER use linked-list in your code again* at [kjellkod.wordpress.com](http://kjellkod.wordpress.com)
- [9] Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (1999), *Resizable Arrays in Optimal Time and Space* (Technical Report CS-99-09) (PDF), Department of Computer Science, University of Waterloo

[10] Goodrich, Michael T.; Kloss II, John G. (1999), “Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences”, *Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **1663**: 205–216, doi:10.1007/3-540-48447-7\_21, ISBN 978-3-540-66279-2

[11] Sitarski, Edward (September 1996), “HATs: Hashed array trees”, *Dr. Dobb's Journal* **21** (11) |chapter= ignored (help)

[12] Bagwell, Phil (2002), *Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays*, EPFL

[13] Javadoc on `ArrayList`

### 3.5.7 External links

- NIST Dictionary of Algorithms and Data Structures: [Dynamic array](#)
- VPOOL - C language implementation of dynamic array.
- CollectionSpy — A Java profiler with explicit support for debugging `ArrayList`- and `Vector`-related issues.
- Open Data Structures - Chapter 2 - Array-Based Lists

## 3.6 Hashed array tree

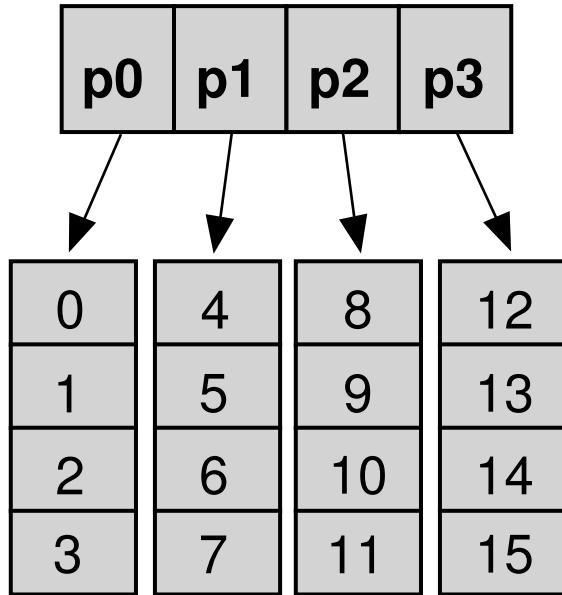
In computer science, a **hashed array tree (HAT)** is a **dynamic array** data-structure published by Edward Sitarski in 1996,<sup>[1]</sup> maintaining an array of separate memory fragments (or “leaves”) to store the data elements, unlike simple dynamic arrays which maintain their data in one contiguous memory area. Its primary objective is to reduce the amount of element copying due to automatic array resizing operations, and to improve memory usage patterns.

Whereas simple dynamic arrays based on geometric expansion waste linear ( $\Omega(n)$ ) space, where  $n$  is the number of elements in the array, hashed array trees waste only order  $O(\sqrt{n})$  storage space. An optimization of the algorithm allows to eliminate data copying completely, at a cost of increasing the wasted space.

It can perform **access** in constant ( $O(1)$ ) time, though slightly slower than simple dynamic arrays. The algorithm has  $O(1)$  amortized performance when appending a series of objects to the end of a hashed array tree. Contrary to its name, it does not use **hash** functions.

### 3.6.1 Definitions

As defined by Sitarski, a hashed array tree has a top-level directory containing a **power of two** number of leaf ar-



A full Hashed Array Tree with 16 elements

rays. All leaf arrays are the same size as the top-level directory. This structure superficially resembles a **hash table** with array-based collision chains, which is the basis for the name *hashed array tree*. A full hashed array tree can hold  $m^2$  elements, where  $m$  is the size of the top-level directory.<sup>[1]</sup> The use of powers of two enables faster physical addressing through bit operations instead of arithmetic operations of **quotient** and **remainder**<sup>[1]</sup> and ensures the  $O(1)$  amortized performance of append operation in the presence of occasional global array copy while expanding.

### 3.6.2 Expansions and size reductions

In a usual dynamic array **geometric expansion** scheme, the array is reallocated as a whole sequential chunk of memory with the new size a double of its current size (and the whole data is then moved to the new location). This ensures  $O(1)$  amortized operations at a cost of  $O(n)$  wasted space, as the enlarged array is filled to the half of its new capacity.

When a hashed array tree is full, its directory and leaves must be restructured to twice their prior size to accommodate additional append operations. The data held in old structure is then moved into the new locations. Only one new leaf is then allocated and added into the top array which thus becomes filled only to a quarter of its new capacity. All the extra leaves are not allocated yet, and will only be allocated when needed, thus wasting only  $O(\sqrt{n})$  of storage.

There are multiple alternatives for reducing size: when a Hashed Array Tree is one eighth full, it can be restructured to a smaller, half-full hashed array tree; another option is only freeing unused leaf arrays, without

resizing the leaves. Further optimizations include adding new leaves without resizing, growing the directory array as needed, possibly through geometric expansion. This would eliminate the need for data copying completely, at the cost of making the wasted space  $O(n)$ , with a small coefficient, and only performing restructuring when a set threshold overhead is reached.<sup>[1]</sup>

### 3.6.3 Related data structures

Brodnik et al.<sup>[2]</sup> presented a **dynamic array** algorithm with a similar space wastage profile to hashed array trees. Brodnik's implementation retains previously allocated leaf arrays, with a more complicated address calculation function as compared to hashed array trees.

#### See also

- **Dynamic array**
- **Unrolled linked list**
- **VList**
- **B-tree**

### 3.6.4 References

- [1] Sitarski, Edward (September 1996), “HATS: Hashed array trees”, *Dr. Dobb’s Journal* **21** (11) |chapter= ignored (help)
- [2] Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (Technical Report CS-99-09), *Resizable Arrays in Optimal Time and Space* (PDF), Department of Computer Science, University of Waterloo Check date values in: |date= (help)

## 3.7 Gap buffer

A **gap buffer** in computer science is a dynamic array that allows efficient insertion and deletion operations clustered near the same location. Gap buffers are especially common in **text editors**, where most changes to the text occur at or near the current location of the **cursor**. The text is stored in a large buffer in two contiguous segments, with a gap between them for inserting new text. Moving the cursor involves copying text from one side of the gap to the other (sometimes copying is delayed until the next operation that changes the text). Insertion adds new text at the end of the first segment; deletion deletes it.

Text in a gap buffer is represented as two **strings**, which take very little extra space and which can be searched and displayed very quickly, compared to more sophisticated **data structures** such as **linked lists**. However, operations at different locations in the text and ones that fill the gap

(requiring a new gap to be created) may require copying most of the text, which is especially inefficient for large files. The use of gap buffers is based on the assumption that such recopying occurs rarely enough that its cost can be amortized over the more common cheap operations. This makes the gap buffer a simpler alternative to the rope for use in text editors<sup>[1]</sup> such as Emacs.<sup>[2]</sup>

### 3.7.1 Example

Below are some examples of operations with buffer gaps. The gap is represented by the empty space between the square brackets. This representation is a bit misleading: in a typical implementation, the endpoints of the gap are tracked using pointers or array indices, and the contents of the gap are ignored; this allows, for example, deletions to be done by adjusting a pointer without changing the text in the buffer. It is a common programming practice to use a semi-open interval for the gap pointers, i.e. the start-of-gap points to the invalid character following the last character in the first buffer, and the end-of-gap points to the first valid character in the second buffer (or equivalently, the pointers are considered to point “between” characters).

Initial state:

This is the way [ ]out.

User inserts some new text:

This is the way the world started [ ]out.

User moves the cursor before “started”; system moves “started ” from the first buffer to the second buffer.

This is the way the world [ ]started out.

User adds text filling the gap; system creates new gap:

This is the way the world as we know it [ ]started out.

### 3.7.2 See also

- Dynamic array, the special case of a gap buffer where the gap is always at the end
- Zipper (data structure), conceptually a generalization of the gap buffer.
- Linked list
- Circular buffer
- Rope (computer science)
- Piece table - data structure used by Bravo and Microsoft Word

### 3.7.3 References

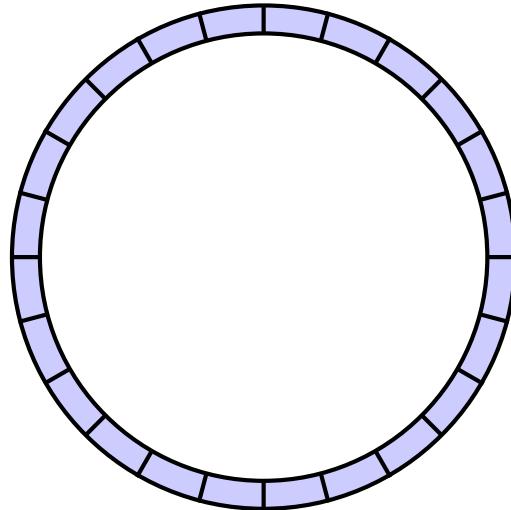
[1] Mark C. Chu-Carroll. "Gap Buffers, or, Don't Get Tied Up With Ropes?" *ScienceBlogs*, 2009-02-18. Accessed 2013-01-30.

[2] emacs gap buffer info Accessed 2013-01-30.

### 3.7.4 External links

- Overview and implementation in .NET/C#
- Brief overview and sample C++ code
- Implementation of a cyclic sorted gap buffer in .NET/C#
- Use of gap buffer in early editor. (First written somewhere between 1969 and 1971)
- emacs gap buffer info(Emacs gap buffer reference)
- Text Editing

## 3.8 Circular buffer



A *ring* showing, conceptually, a circular buffer. This visually shows that the buffer has no real end and it can loop around the buffer. However, since memory is never physically created as a ring, a linear representation is generally used as is done below.

A **circular buffer**, **cyclic buffer** or **ring buffer** is a data structure that uses a single, fixed-size **buffer** as if it were connected end-to-end. This structure lends itself easily to buffering data streams.

### 3.8.1 Uses

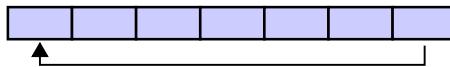
The useful property of a circular buffer is that it does not need to have its elements shuffled around when one is consumed. (If a non-circular buffer were used then it would be necessary to shift all elements when one is consumed.) In other words, the circular buffer is well-suited as a **FIFO** buffer while a standard, non-circular buffer is well suited as a **LIFO** buffer.

Circular buffering makes a good implementation strategy for a queue that has fixed maximum size. Should a maximum size be adopted for a queue, then a circular buffer is a completely ideal implementation; all queue operations are constant time. However, expanding a circular buffer requires shifting memory, which is comparatively costly. For arbitrarily expanding queues, a [linked list](#) approach may be preferred instead.

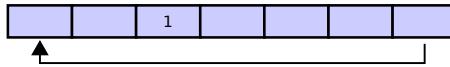
In some situations, overwriting circular buffer can be used, e.g. in multimedia. If the buffer is used as the bounded buffer in the [producer-consumer problem](#) then it is probably desired for the producer (e.g., an audio generator) to overwrite old data if the consumer (e.g., the sound card) is unable to momentarily keep up. Also, the [LZ77](#) family of lossless data compression algorithms operates on the assumption that strings seen more recently in a data stream are more likely to occur soon in the stream. Implementations store the most recent data in a circular buffer.

### 3.8.2 How it works

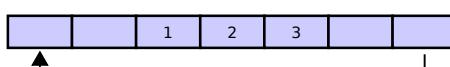
A circular buffer first starts empty and of some predefined length. For example, this is a 7-element buffer:



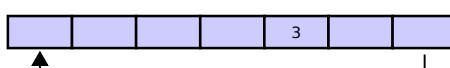
Assume that a 1 is written into the middle of the buffer (exact starting location does not matter in a circular buffer):



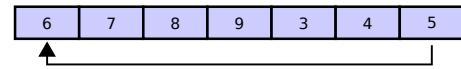
Then assume that two more elements are added — 2 & 3 — which get appended after the 1:



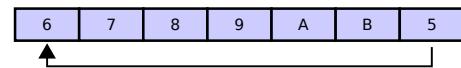
If two elements are then removed from the buffer, the oldest values inside the buffer are removed. The two elements removed, in this case, are 1 & 2, leaving the buffer with just a 3:



If the buffer has 7 elements then it is completely full:

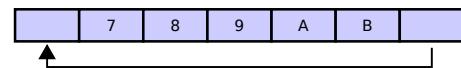


A consequence of the circular buffer is that when it is full and a subsequent write is performed, then it starts overwriting the oldest data. In this case, two more elements — A & B — are added and they *overwrite* the 3 & 4:



Alternatively, the routines that manage the buffer could prevent overwriting the data and return an error or raise an [exception](#). Whether or not data is overwritten is up to the semantics of the buffer routines or the application using the circular buffer.

Finally, if two elements are now removed then what would be returned is **not** 3 & 4 but 5 & 6 because A & B overwrote the 3 & the 4 yielding the buffer with:



### 3.8.3 Circular buffer mechanics

What is not shown in the example above is the mechanics of how the circular buffer is managed.

#### Start/end pointers (head/tail)

Generally, a circular buffer requires four pointers:

- one to the actual buffer in memory
- one to the buffer end in memory (or alternately: the size of the buffer)
- one to point to the start of valid data (or alternately: amount of data written to the buffer)
- one to point to the end of valid data (or alternately: amount of data read from the buffer)

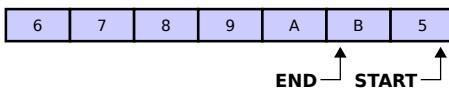
Alternatively, a fixed-length buffer with two integers to keep track of indices can be used in languages that do not have pointers.

Taking a couple of examples from above. (While there are numerous ways to label the pointers and exact semantics can vary, this is one way to do it.)

This image shows a partially full buffer:



This image shows a full buffer with two elements having been overwritten:

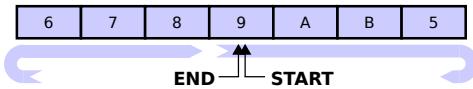


What to note about the second one is that after each element is overwritten then the start pointer is incremented as well.

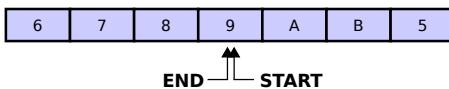
### 3.8.4 Difficulties

#### Full / Empty Buffer Distinction

A small disadvantage of relying on pointers or relative indices of the start and end of data is, that in the case the buffer is entirely full, both pointers point to the same element:



This is exactly the same situation as when the buffer is empty:



To solve this confusion there are a number of solutions:

- Always keep one slot open.
- Use a fill count to distinguish the two cases.
- Use an extra mirroring bit to distinguish the two cases.
- Use read and write counts to get the fill count from.
- Use absolute indices.
- Record last operation.
- Split buffer into two regions.

**Always keep one slot open** This design always keeps one slot unallocated. A full buffer has at most  $(\text{size} - 1)$  slots. If both pointers refer to the same slot, the buffer is empty. If the end (write) pointer refers to the slot preceding the one referred to by the start (read) pointer, the buffer is full.

The advantage is:

- The solution is simple and robust.

The disadvantages are:

- One slot is lost, so it is a bad compromise when the buffer size is small or the slot is big or is implemented in hardware.
- The test for full requires a modulo operation

**Use a fill count** This approach replaces the end pointer with a counter that tracks the number of readable items in the buffer. This unambiguously indicates when the buffer is empty or full and allows use of all buffer slots.

The performance impact should be negligible, since this approach adds the costs of maintaining the counter and computing the tail slot on writes but eliminates the need to maintain the end pointer and simplifies the fullness test.

The advantage is:

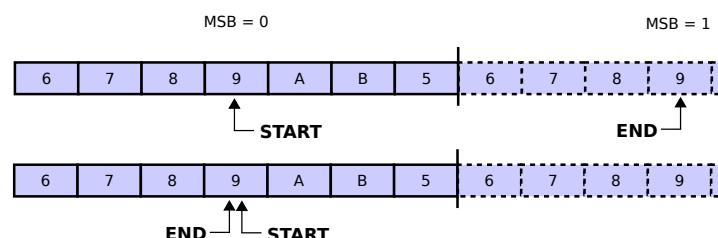
- The test for full/empty is simple

The disadvantages are:

- You need modulo for read and write
- Read and write operation must share the counter field, so it requires synchronization in multi-threaded situation.

Note: When using semaphores in a Producer-consumer model, the semaphores act as a fill count.

**Mirroring** Another solution is to remember the number of times each read and write pointers have wrapped and compare this to distinguish empty and full situations. In fact only the parity of the number of wraps is necessary, so you only need to keep an extra bit. You can see this as if the buffer adds a virtual mirror and the pointers point either to the normal or to the mirrored buffer.



It is easy to see above that when the pointers (including the extra msb bit) are equal, the buffer is empty, while if the pointers differ only by the extra msb bit, the buffer is full.

The advantages are:

- The test for full/empty is simple
- No modulo operation is needed
- The source and sink of data can implement independent policies for dealing with a full buffer and overrun while adhering to the rule that only the source of data modifies the write count and only the sink of data modifies the read count. This can result in elegant and robust circular buffer implementations even in multi-threaded environments.

The disadvantage is:

- You need one more bit for read and write pointer

**Read / Write Counts** Another solution is to keep counts of the number of items written to and read from the circular buffer. Both counts are stored in signed integer variables with numerical limits larger than the number of items that can be stored and are allowed to wrap freely.

The unsigned difference (write\_count - read\_count) always yields the number of items placed in the buffer and not yet retrieved. This can indicate that the buffer is empty, partially full, completely full (without waste of a storage location) or in a state of overrun.

The advantage is:

- The source and sink of data can implement independent policies for dealing with a full buffer and overrun while adhering to the rule that only the source of data modifies the write count and only the sink of data modifies the read count. This can result in elegant and robust circular buffer implementations even in multi-threaded environments.

The disadvantage is:

- You need two additional variables.

**Absolute indices** It is possible to optimize the previous solution by using indices instead of pointers: indices can store read/write counts instead of the offset from start of the buffer, the separate variables in the above solution are removed and relative indices are obtained on the fly by division **modulo** the buffer's length.

The advantage is:

- No extra variables are needed.

The disadvantages are:

- Every access needs an additional *modulo* operation.
- If counter wrap is possible, complex logic can be needed if the buffer's length is not a divisor of the counter's capacity.

On binary computers, both of these disadvantages disappear if the buffer's length is a power of two—at the cost of a constraint on possible buffers lengths.

**Record last operation** Another solution is to keep a flag indicating whether the most recent operation was a read or a write. If the two pointers are equal, then the flag will show whether the buffer is full or empty: if the most recent operation was a write, the buffer must be full, and conversely if it was a read, it must be empty.

The advantages are:

- Only a single bit needs to be stored (which may be particularly useful if the algorithm is implemented in hardware)
- The test for full/empty is simple

The disadvantage is:

- You need an extra variable
- Read and write operations must share the flag, so it will probably require synchronization in multi-threaded situation.

**Split buffer into two regions** This approach solves the wrap-around problem by splitting the buffer into a primary region and a secondary region. The secondary region always begins at the buffer start and is not activated until the primary region has reached the buffer's end. Additionally, if the primary region is emptied of data, the secondary region becomes the new primary.

The advantages are:

- The test for full/empty is simple.
- No modulo operation is needed.

The disadvantages are:

- An extra pointer variable is needed.
- Read and write operations are complicated.

### Multiple read pointers

A little bit more complex are multiple read pointers on the same circular buffer. This is useful if you have  $n$  threads, which are reading from the same buffer, but *one* thread writing to the buffer.

### Chunked Buffer

Much more complex are different chunks of data in the same circular buffer. The writer is not only writing elements to the buffer, it also assigns these elements to chunks .

The reader should not only be able to read from the buffer, it should also get informed about the chunk borders.

Example: The writer is reading data from small files, writing them into the same circular buffer. The reader is reading the data, but needs to know when and which file is starting at a given position.

### 3.8.5 Variants

Perhaps the most common version of the circular buffer uses 8-bit bytes as elements.

Some implementations of the circular buffer use fixed-length elements that are bigger than 8-bit bytes—16-bit integers for audio buffers, 53-byte ATM cells for telecom buffers, etc. Each item is contiguous and has the correct [data alignment](#), so software reading and writing these values can be faster than software that handles non-contiguous and non-aligned values.

[Ping-pong buffering](#) can be considered a very specialized circular buffer with exactly two large fixed-length elements.

The [Bip Buffer](#) is very similar to a circular buffer, except it always returns contiguous blocks (which can be variable length).<sup>[1]</sup>

### 3.8.6 External links

[1] Simon Cooke. “The Bip Buffer - The Circular Buffer with a Twist”. 2003.

- [CircularBuffer at the Portland Pattern Repository](#)
- [Boost: Templated Circular Buffer Container](#)
- <http://www.dspguide.com/ch28/2.htm>

## 3.9 Sparse array

In computer science, a **sparse array** is an array in which most of the elements have the same value (known as the default value—usually 0 or [null](#)). The occurrence of zero elements in a large array is inefficient for both computation and [storage](#). An array in which there is a large number of zero elements is referred to as being sparse.

In the case of sparse arrays, one can ask for a value from an “empty” array position. If one does this, then for an array of numbers, a value of zero should be returned, and for an array of objects, a value of [null](#) should be returned.

A [naive](#) implementation of an array may allocate space for the entire array, but in the case where there are few non-default values, this implementation is inefficient. Typically the algorithm used instead of an ordinary array is determined by other known features (or statistical features) of the array. For instance, if the sparsity is known in advance or if the elements are arranged according to some function (e.g., the elements occur in blocks).

A [heap memory allocator](#) in a program might choose to store regions of blank space in a linked list rather than storing all of the allocated regions in, say a [bit array](#).

### 3.9.1 Representation

Sparse Array can be represented as

`Sparse_Array[{pos1 -> val1, pos2 -> val2,...}]` or  
`Sparse_Array[{pos1, pos2,...} -> {val1, val2,...}]`

which yields a sparse array in which values *val<sub>i</sub>* appear at positions *pos<sub>i</sub>* .

### 3.9.2 Sparse Array as Linked List

An obvious question that might be asked is why we need a linked list to represent a sparse array if we can represent it easily using a normal array. The answer to this question lies in the fact that while representing a sparse array as a normal array, a lot of space is allocated for zero or [null](#) elements. For example, consider the following array declaration:

```
double arr[1000][1000];
```

When we define this array, enough space for 1,000,000 doubles is allocated. If each double requires 8 bytes of memory, this array will require 8 million [bytes](#) of memory. Because this is a sparse array, most of its elements will have a value of zero (or [null](#)). Hence, defining this array will soak up all this space which will result in wastage of memory (compared to an array in which memory has been allocated only for the nonzero elements). An effective way to overcome this problem is to represent the array using a linked list which requires less memory as only elements having non-zero value are stored. Also, when a linked list is used, the array elements can be accessed through fewer iterations than in a normal array.

A sparse array as a linked list contains nodes linked to each other. In a one-dimensional sparse array, each node consists of an “index” (position) of the non-zero element and the “value” at that position and a node pointer “next” (for linking to the next node), nodes are linked in order as per the index. In the case of a two-dimensional sparse array, each node contains a row index, a column index (which together give its position), a value at that position and a pointer to the next node.

### 3.9.3 See also

- Sparse matrix

### 3.9.4 External links

- Boost sparse vector class
- Rodolphe Buda, “Two Dimensional Aggregation Procedure: An Alternative to the Matrix Algebraic Algorithm”, *Computational Economics*, 31(4), May, pp.397–408, 2008.

## 3.10 Bit array

A **bit array** (also known as **bitmap**, **bitset**, **bit string**, or **bit vector**) is an **array data structure** that compactly stores **bits**. It can be used to implement a simple **set data structure**. A bit array is effective at exploiting bit-level parallelism in hardware to perform operations quickly. A typical bit array stores  $kw$  bits, where  $w$  is the number of bits in the unit of storage, such as a **byte** or **word**, and  $k$  is some nonnegative integer. If  $w$  does not divide the number of bits to be stored, some space is wasted due to internal fragmentation.

### 3.10.1 Definition

A bit array is a mapping from some domain (almost always a range of integers) to values in the set  $\{0, 1\}$ . The values can be interpreted as dark/light, absent/present, locked/unlocked, valid/invalid, et cetera. The point is that there are only two possible values, so they can be stored in one bit. As with other arrays, the access to a single bit can be managed by applying an index to the array. Assuming its size (or length) to be  $n$  bits, the array can be used to specify a subset of the domain (e.g.  $\{0, 1, 2, \dots, n-1\}$ ), where a 1-bit indicates the presence and a 0-bit the absence of a number in the set. This set data structure uses about  $n/w$  words of space, where  $w$  is the number of bits in each **machine word**. Whether the least significant bit (of the word) or the most significant bit indicates the smallest-index number is largely irrelevant, but the former tends to be preferred (on **little-endian** machines).

### 3.10.2 Basic operations

Although most machines are not able to address individual bits in memory, nor have instructions to manipulate single bits, each bit in a word can be singled out and manipulated using **bitwise operations**. In particular:

- OR can be used to set a bit to one:  $11101010 \text{ OR } 00000100 = 11101110$

- AND can be used to set a bit to zero:  $11101010 \text{ AND } 11111101 = 11101000$
- AND together with zero-testing can be used to determine if a bit is set:

$$\begin{aligned} 11101010 \text{ AND } 00000001 &= \\ 00000000 = 0 & \\ 11101010 \text{ AND } 00000010 &= \\ 00000010 \neq 0 & \end{aligned}$$

- XOR can be used to invert or toggle a bit:

$$\begin{aligned} 11101010 \text{ XOR } 00000100 &= \\ 11101110 & \\ 11101110 \text{ XOR } 00000100 &= \\ 11101010 & \end{aligned}$$

- NOT can be used to invert all bits.

$$\text{NOT } 10110010 = 01001101$$

To obtain the bit mask needed for these operations, we can use a **bit shift operator** to shift the number 1 to the left by the appropriate number of places, as well as **bitwise negation** if necessary.

Given two bit arrays of the same size representing sets, we can compute their **union**, **intersection**, and **set-theoretic difference** using  $n/w$  simple bit operations each ( $2n/w$  for difference), as well as the **complement** of either:

```
for i from 0 to n/w-1 complement_a[i] := not a[i] union[i] := a[i] or b[i] intersection[i] := a[i] and b[i] difference[i] := a[i] and (not b[i])
```

If we wish to iterate through the bits of a bit array, we can do this efficiently using a doubly nested loop that loops through each word, one at a time. Only  $n/w$  memory accesses are required:

```
for i from 0 to n/w-1 index := 0 // if needed word := a[i] for b from 0 to w-1 value := word and 1 ≠ 0 word := word shift right 1 // do something with value index := index + 1 // if needed
```

Both of these code samples exhibit **ideal locality of reference**, which will subsequently receive large performance boost from a data cache. If a cache line is  $k$  words, only about  $n/wk$  cache misses will occur.

### 3.10.3 More complex operations

As with **character strings** it is straightforward to define **length**, **substring**, **lexicographical compare**, **concatenation**, **reverse** operations. The implementation of some of these operations is sensitive to **endianness**.

### Population / Hamming weight

If we wish to find the number of 1 bits in a bit array, sometimes called the population count or Hamming weight, there are efficient branch-free algorithms that can compute the number of bits in a word using a series of simple bit operations. We simply run such an algorithm on each word and keep a running total. Counting zeros is similar. See the [Hamming weight](#) article for examples of an efficient implementation.

### Sorting

Similarly, sorting a bit array is trivial to do in  $O(n)$  time using [counting sort](#) — we count the number of ones  $k$ , fill the last  $k/w$  words with ones, set only the low  $k \bmod w$  bits of the next word, and set the rest to zero.

### Inversion

Vertical flipping of a one-bit-per-pixel image, or some FFT algorithms, require to flip the bits of individual words (so  $b_{31} b_{30} \dots b_0$  becomes  $b_0 \dots b_{30} b_{31}$ ). When this operation is not available on the processor, it's still possible to proceed by successive passes, in this example on 32 bits:

exchange two 16bit halfwords exchange bytes by pairs (0xddccbaa -> 0xccddabb) ... swap bits by pairs swap bits ( $b_{31} b_{30} \dots b_1 b_0 \rightarrow b_{30} b_{31} \dots b_0 b_1$ ) The last operation can be written  $((x \& 0x55555555) \ll 1) | (x \& 0xaaaaaaaa) \gg 1$ ).

### Find first one

The *find first set* or *find first one* operation identifies the index or position of the 1-bit with the smallest index in an array, and has widespread hardware support (for arrays not larger than a word) and efficient algorithms for its computation. When a [priority queue](#) is stored in a bit array, *find first one* can be used to identify the highest priority element in the queue. To expand a word-size *find first one* to longer arrays, one can find the first nonzero word and then run *find first one* on that word. The related operations *find first zero*, *count leading zeros*, *count leading ones*, *count trailing zeros*, *count trailing ones*, and *log base 2* (see *find first set*) can also be extended to a bit array in a straightforward manner.

### 3.10.4 Compression

A bit array is the densest storage for “random” bits, that is, where each bit is equally likely to be 0 or 1, and each one is independent. But most data is not random, so it

may be possible to store it more compactly. For example, the data of a typical fax image is not random and can be compressed. [Run-length encoding](#) is commonly used to compress these long streams. However, most compressed data formats are not so easy to access randomly; also by compressing bit arrays too aggressively we run the risk of losing the benefits due to bit-level parallelism ([vectorization](#)). Thus, instead of compressing bit arrays as streams of bits, we might compress them as streams of bytes or words (see [Bitmap index \(compression\)](#)).

### 3.10.5 Advantages and disadvantages

Bit arrays, despite their simplicity, have a number of marked advantages over other data structures for the same problems:

- They are extremely compact; few other data structures can store  $n$  independent pieces of data in  $n/w$  words.
- They allow small arrays of bits to be stored and manipulated in the register set for long periods of time with no memory accesses.
- Because of their ability to exploit bit-level parallelism, limit memory access, and maximally use the [data cache](#), they often outperform many other data structures on practical data sets, even those that are more asymptotically efficient.

However, bit arrays aren't the solution to everything. In particular:

- Without compression, they are wasteful set data structures for sparse sets (those with few elements compared to their range) in both time and space. For such applications, compressed bit arrays, [Judy arrays](#), tries, or even [Bloom filters](#) should be considered instead.
- Accessing individual elements can be expensive and difficult to express in some languages. If random access is more common than sequential and the array is relatively small, a byte array may be preferable on a machine with byte addressing. A word array, however, is probably not justified due to the huge space overhead and additional cache misses it causes, unless the machine only has word addressing.

### 3.10.6 Applications

Because of their compactness, bit arrays have a number of applications in areas where space or efficiency is at a premium. Most commonly, they are used to represent a simple group of boolean flags or an ordered sequence of boolean values.

Bit arrays are used for [priority queues](#), where the bit at index  $k$  is set if and only if  $k$  is in the queue; this data structure is used, for example, by the [Linux kernel](#), and benefits strongly from a find-first-zero operation in hardware.

Bit arrays can be used for the allocation of [memory pages](#), [inodes](#), disk sectors, etc. In such cases, the term *bitmap* may be used. However, this term is frequently used to refer to [raster images](#), which may use multiple bits per pixel.

Another application of bit arrays is the [Bloom filter](#), a probabilistic [set data structure](#) that can store large sets in a small space in exchange for a small probability of error. It is also possible to build probabilistic hash tables based on bit arrays that accept either false positives or false negatives.

Bit arrays and the operations on them are also important for constructing [succinct data structures](#), which use close to the minimum possible space. In this context, operations like finding the  $n$ th 1 bit or counting the number of 1 bits up to a certain position become important.

Bit arrays are also a useful abstraction for examining streams of [compressed data](#), which often contain elements that occupy portions of bytes or are not byte-aligned. For example, the compressed [Huffman coding](#) representation of a single 8-bit character can be anywhere from 1 to 255 bits long.

In [information retrieval](#), bit arrays are a good representation for the [posting lists](#) of very frequent terms. If we compute the gaps between adjacent values in a list of strictly increasing integers and encode them using [unary coding](#), the result is a bit array with a 1 bit in the  $n$ th position if and only if  $n$  is in the list. The implied probability of a gap of  $n$  is  $1/2^n$ . This is also the special case of [Golomb coding](#) where the parameter  $M$  is 1; this parameter is only normally selected when  $-\log(2-p)/\log(1-p) \leq 1$ , or roughly the term occurs in at least 38% of documents.

### 3.10.7 Language support

The [C](#) programming language's *bitfields*, pseudo-objects found in structs with size equal to some number of bits, are in fact small bit arrays; they are limited in that they cannot span words. Although they give a convenient syntax, the bits are still accessed using bitwise operators on most machines, and they can only be defined statically (like C's static arrays, their sizes are fixed at compile-time). It is also a common idiom for C programmers to use words as small bit arrays and access bits of them using bit operators. A widely available header file included in the [X11](#) system, `xtrapbits.h`, is "a portable way for systems to define bit field manipulation of arrays of bits." A more explanatory description of aforementioned approach can be found in the `comp.lang.c` faq.

In [C++](#), although individual bools typically occupy the

same space as a byte or an integer, the [STL](#) type `vector<bool>` is a [partial template specialization](#) in which bits are packed as a space efficiency optimization. Since bytes (and not bits) are the smallest addressable unit in C++, the `[]` operator does *not* return a reference to an element, but instead returns a [proxy reference](#). This might seem a minor point, but it means that `vector<bool>` is *not* a standard STL container, which is why the use of `vector<bool>` is generally discouraged. Another unique STL class, `bitset`,<sup>[1]</sup> creates a vector of bits fixed at a particular size at compile-time, and in its interface and syntax more resembles the idiomatic use of words as bit sets by C programmers. It also has some additional power, such as the ability to efficiently count the number of bits that are set. The [Boost C++ Libraries](#) provide a `dynamic_bitset` class<sup>[2]</sup> whose size is specified at run-time.

The [D](#) programming language provides bit arrays in both of its competing standard libraries. In Phobos, they are provided in `std.bitmanip`, and in Tango, they are provided in `tango.core.BitArray`. As in C++, the `[]` operator does not return a reference, since individual bits are not directly addressable on most hardware, but instead returns a `bool`.

In [Java](#), the class `BitSet` creates a bit array that is then manipulated with functions named after bitwise operators familiar to C programmers. Unlike the `bitset` in C++, the Java `BitSet` does not have a "size" state (it has an effectively infinite size, initialized with 0 bits); a bit can be set or tested at any index. In addition, there is a class `EnumSet`, which represents a Set of values of an [enumerated type](#) internally as a bit vector, as a safer alternative to `bitfields`.

The [.NET Framework](#) supplies a `BitArray` collection class. It stores boolean values, supports random access and bitwise operators, can be iterated over, and its `Length` property can be changed to grow or truncate it.

Although [Standard ML](#) has no support for bit arrays, Standard ML of New Jersey has an extension, the `BitArray` structure, in its [SML/NJ Library](#). It is not fixed in size and supports set operations and bit operations, including, unusually, shift operations.

[Haskell](#) likewise currently lacks standard support for bitwise operations, but both [GHC](#) and [Hugs](#) provide a `Data.Bits` module with assorted bitwise functions and operators, including shift and rotate operations and an "unboxed" array over boolean values may be used to model a Bit array, although this lacks support from the former module.

In [Perl](#), strings can be used as expandable bit arrays. They can be manipulated using the usual bitwise operators (`~` `|` `&` `^`),<sup>[3]</sup> and individual bits can be tested and set using the `vec` function.<sup>[4]</sup>

In [Ruby](#), you can access (but not set) a bit of an integer (Fixnum or Bignum) using the bracket operator (`[]`), as if it were an array of bits.

Apple's Core Foundation library contains `CFBitVector`

and `CFMutableBitVector` structures.

**PL/I** supports arrays of *bit strings* of arbitrary length, which may be either fixed-length or varying. The array elements may be *aligned*— each element begins on a byte or word boundary— or *unaligned*— elements immediately follow each other with no padding.

Hardware description languages such as **VHDL**, **Verilog**, and **SystemVerilog** natively support bit vectors as these are used to model storage elements like **flip-flops**, hardware busses and hardware signals in general. In hardware verification languages such as **OpenVera**, **e** and **SystemVerilog**, bit vectors are used to sample values from the hardware models, and to represent data that is transferred to hardware during simulations.

### 3.10.8 See also

- Bit field
- Bitboard Chess and similar games.
- Bitmap index
- Binary numeral system
- Bitstream
- Judy array

### 3.10.9 References

- [1] `std::bitset`
- [2] `boost::dynamic_bitset`
- [3] <http://perldoc.perl.org/perlop.html#Bitwise-String-Operators>
- [4] <http://perldoc.perl.org/functions/vec.html>

### 3.10.10 External links

- mathematical bases by Pr. D.E.Knuth
- `vector<bool>` Is Nonconforming, and Forces Optimization Choice
- `vector<bool>`: More Problems, Better Solutions

## 3.11 Bitboard

A **bitboard** is a data structure commonly used in computer systems that play board games.

A bitboard, often used for boardgames such as **chess**, **checkers**, **othello** and **word games**, is a specialization of the bit array data structure, where each bit represents a game position or state, designed for optimization of speed

and/or memory or disk use in mass calculations. Bits in the same bitboard relate to each other in the rules of the game often forming a game position when taken together. Other bitboards are commonly used as masks to transform or answer queries about positions. The “game” may be any game-like system where information is tightly packed in a structured form with “rules” affecting how the individual units or pieces relate.

### 3.11.1 Short description

Bitboards are used in many of the world’s highest-rated chess playing programs such as **Houdini**, **Stockfish**, and **Critter**. They help the programs analyze chess positions with few CPU instructions and hold a massive number of positions in memory efficiently.

Bitboards allow the computer to answer some questions about game state with one logical operation. For example, if a chess program wants to know if the white player has any pawns in the center of the board (center four squares) it can just compare a bitboard for the player’s pawns with one for the center of the board using a logical AND operation. If there are no center pawns then the result will be zero.

Query results can also be represented using bitboards. For example, the query “What are the squares between X and Y?” can be represented as a bitboard. These query results are generally pre-calculated, so that a program can simply retrieve a query result with one memory load.

However, as a result of the massive compression and encoding, bitboard programs are not easy for software developers to either write or debug.

### 3.11.2 History

The bitboard method for holding a board game appears to have been invented in the mid-1950s, by Arthur Samuel and was used in his checkers program. The method was published in 1959 as “Some Studies in Machine Learning Using the Game of Checkers” in the IBM Journal of Research and Development.

For the more complicated game of chess, it appears the method was independently rediscovered later by the Kaissa team in the Soviet Union in the late 1960s, published in 1970 “Programming a computer to play chess” in Russ. Math. Surv. , and again by the authors of the U.S. Northwestern University program “Chess” in the early 1970s, and documented in 1977 in “Chess Skill in Man and Machine”.

### 3.11.3 Description for all games or applications

Main article: Bit field

A bitboard or bit field is a format that stuffs a whole group of related boolean variables into the same machine word, typically representing positions on a board game. Each bit is a position, and when the bit is positive, a property of that position is true. In chess, for example, there would be a bitboard for black knights. There would be 64-bits where each bit represents a chess square. Another bitboard might be a constant representing the center four squares of the board. By comparing the two numbers with a bitwise logical AND instruction, we get a third bitboard which represents the black knights on the center four squares, if any. This format is generally more CPU and memory friendly than others.

### 3.11.4 General technical advantages and disadvantages

#### Processor use

**Pros** The advantage of the bitboard representation is that it takes advantage of the essential logical **bitwise** operations available on nearly all **CPUs** that complete in one cycle and are fully pipelined and cached etc. Nearly all CPUs have **AND**, **OR**, **NOR**, and **XOR**. Many CPUs have additional bit instructions, such as finding the “first” bit, that make bitboard operations even more efficient. If they do not have instructions well known algorithms can perform some “magic” transformations that do these quickly.

Furthermore, modern CPUs have **instruction pipelines** that queue instructions for execution. A processor with multiple execution units can perform more than one instruction per cycle if more than one instruction is available in the pipeline. Branching (the use of conditionals like **if**) makes it harder for the processor to fill its pipeline(s) because the CPU can't tell what it needs to do in advance. Too much branching makes the pipeline less effective and potentially reduces the number of instructions the processor can execute per cycle. Many bitboard operations require fewer conditionals and therefore increase pipelining and make effective use of multiple execution units on many CPUs.

CPUs have a bit width which they are designed toward and can carry out bitwise operations in one cycle in this width. So, on a 64-bit or more CPU, 64-bit operations can occur in one instruction. There may be support for higher or lower width instructions. Many 32-bit CPUs may have some 64-bit instructions and those may take more than one cycle or otherwise be handicapped compared to their 32-bit instructions.

If the bitboard is larger than the width of the instruction set, then a performance hit will be the result. So a program using 64-bit bitboards would run faster on a real 64-bit processor than on a 32-bit processor.

**Cons** Some queries are going to take longer than they would with perhaps arrays, so bitboards are generally used in conjunction with array boards in chess programs.

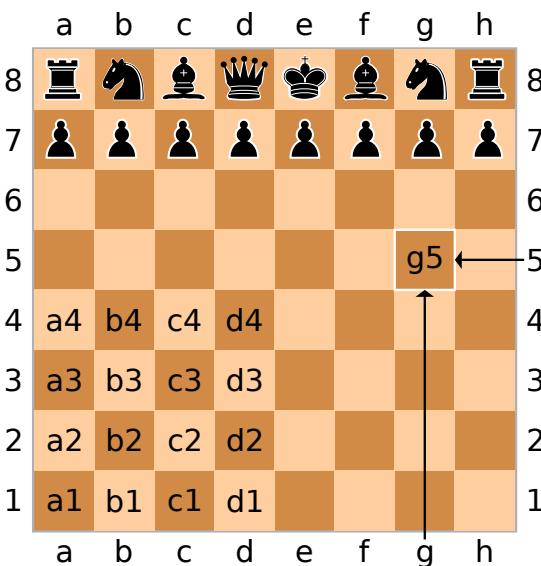
#### Memory use

**Pros** Bitboards are extremely compact. Since only a very small amount of memory is required to represent a position or a mask, more positions can find their way into registers, full speed cache, Level 2 cache, etc. In this way, compactness translates into better performance (on most machines). Also on some machines this might mean that more positions can be stored in main memory before going to disk.

**Cons** For some games writing a suitable bitboard engine requires a fair amount of source code that will be longer than the straight forward implementation. For limited devices (like cell phones) with a limited number of registers or processor instruction cache, this can cause a problem. For full-sized computers it may cause cache misses between level one and level two cache. This is a potential problem—not a major drawback. Most machines will have enough instruction cache so that this isn't an issue.

### 3.11.5 Chess bitboards

#### Standard



Algebraic notation

The first bit usually represents the square a1 (the lower left square), and the 64th bit represents the square h8 (the diagonally opposite square).

There are twelve types of pieces, and each type gets its own bitboard. Black pawns get a board, white pawns, etc. Together these twelve boards can represent a position. Some trivial information also needs to be tracked elsewhere; the programmer may use boolean variables for whether each side is in check, can castle, etc.

Constants are likely available, such as WHITE\_SQUARES, BLACK\_SQUARES, FILE\_A, RANK\_4 etc. More interesting ones might include CENTER, CORNERS, CASTLE\_SQUARES, etc.

Examples of variables would be WHITE\_ATTACKING, ATTACKED\_BY\_PAWN, WHITE\_PASSED\_PAWN, etc.

## Rotated

“Rotated” bitboards are usually used in programs that use bitboards. Rotated bitboards make certain operations more efficient. While engines are simply referred to as “rotated bitboard engines,” this is a misnomer as rotated boards are used in *addition* to normal boards making these hybrid standard/rotated bitboard engines.

These bitboards rotate the bitboard positions by 90 degrees, 45 degrees, and/or 315 degrees. A typical bitboard will have one byte per rank of the chess board. With this bitboard it’s easy to determine rook attacks across a rank, using a table indexed by the occupied square and the occupied positions in the rank (because rook attacks stop at the first occupied square). By rotating the bitboard 90 degrees, rook attacks across a file can be examined the same way. Adding bitboards rotated 45 degrees and 315 degrees produces bitboards in which the diagonals are easy to examine. The queen can be examined by combining rook and bishop attacks. Rotated bitboards appear to have been developed separately and (essentially) simultaneously by the developers of the *DarkThought* and *Crafty* programs.

## Magics

Magic move bitboard generation is a new and fast alternative to rotated move bitboard generators. These are also more versatile than rotated move bitboard generators because the generator can be used independently from any position. The basic idea is that you can use a multiply, right-shift hashing function to index a move database, which can be as small as 1.5K. A speedup is gained because no rotated bitboards need to be updated, and because the lookups are more cache-friendly.

## 3.11.6 Other bitboards

Many other games besides chess benefit from bitboards.

- In *Connect Four*, they allow for very efficient testing for four consecutive discs, by just two shift+and operations per direction.
- In the *Conway’s Game of Life*, they are a possible alternative to arrays.
- *Othello/Reversi* (see the *Reversi* article).
- In *word games*, they allow for very efficient generation of valid moves by simple logical operations.

## 3.11.7 See also

- Bit array
- Bit field
- Bit manipulation
- Bitwise operation
- Board representation (chess)
- Boolean algebra (logic)
- Instruction set
- Instruction pipeline
- Opcode
- Bytecode

## 3.11.8 External links

### Calculators

- 64 bits representation and manipulation

### Checkers

- Checkers Bitboard Tutorial by Jonathan Kreuzer

### Chess

#### Articles

- Programming area of the Beowulf project
- Heinz, Ernst A. How *DarkThought* plays chess. *ICCA Journal*, Vol. 20(3), pp. 166-176, Sept. 1997
- Laramee, Francois-Dominic. *Chess Programming Part 2: Data Structures*.
- Verhelst, Paul. *Chess Board Representations*

- Hyatt, Robert. Chess program board representations
- Hyatt, Robert. Rotated bitmaps, a new twist on an old idea
- Frayn, Colin. How to implement bitboards in a chess engine (chess programming theory)
- Pepicelli, Glen. *Bitfields, Bitboards, and Beyond* - (Example of bitboards in the Java Language and a discussion of why this optimization works with the Java Virtual Machine (www.OnJava.com publisher: O'Reilly 2005))
- Magic Move-Bitboard Generation in Computer Chess. Pradyumna Kannan

### Code examples

- The author of the Frenzee engine had posted some source examples.
- A 155 line java Connect-4 program demonstrating the use of bitboards.

### Implementations

#### Open source

- Beowulf Unix, Linux, Windows. Rotated bitboards.
- Crafty See the Crafty article. Written in straight C. Rotated bitboards in the old versions, now uses magic bitboards.
- GNU Chess See the GNU Chess Article.
- Stockfish UCI chess engine ranking second in Elo as of 2010
- Gray Matter C++, rotated bitboards.
- KnightCap GPL. ELO of 2300.
- Pepito C. Bitboard, by Carlos del Cacho. Windows and Linux binaries as well as source available.
- Simontacci Rotated bitboards.

#### Closed source

- DarkThought Home Page

#### Othello

- A complete discussion of Othello (Reversi) engines with some source code including an Othello bitboard in C and assembly.
- Edax (computing) See the Edax article. An Othello (Reversi) engine with source code based on bitboard.

### Word Games

- Overview of bit board usage in word games.

## 3.12 Parallel array

In computing, a group of **parallel arrays** is a data structure for representing arrays of records. It keeps a separate, homogeneous array for each field of the record, each having the same number of elements. Then, objects located at the same index in each array are implicitly the fields of a single record. Pointers from one object to another are replaced by array indices. This contrasts with the normal approach of storing all fields of each record together in memory. For example, one might declare an array of 100 names, each a string, and 100 ages, each an integer, associating each name with the age that has the same index.

An example in **C** using parallel arrays:

```
int ages[] = {0, 17, 2, 52, 25}; char *names[] = {"None", "Mike", "Billy", "Tom", "Stan"}; int parent[] = {0 /*None*/, 3 /*Tom*/, 1 /*Mike*/, 0 /*None*/, 3 /*Tom*/}; for(i = 1; i <= 4; i++) { printf("Name: %s, Age: %d, Parent: %s \n", names[i], ages[i], names[parent[i]]); }
```

in **Perl** (using a hash of arrays to hold references to each array):

```
my %data = (first_name => ['Joe', 'Bob', 'Frank', 'Hans'], last_name => ['Smith', 'Seger', 'Sinatra', 'Schultze'], height_in_cm => [169, 158, 201, 199]); for $i (0..$#{$data{first_name}}) { printf "Name: %s %s\n", $data{first_name}[$i], $data{last_name}[$i]; printf "Height in CM: %i\n", $data{height_in_cm}[$i]; }
```

Or, in **Python**:

```
firstName = ['Joe', 'Bob', 'Frank', 'Hans'] lastName = ['Smith', 'Seger', 'Sinatra', 'Schultze'] heightInCM = [169, 158, 201, 199] for i in xrange(len(firstName)): print "Name: %s %s" % (firstName[i], lastName[i]) print "Height in CM: %s" % heightInCM[i]
```

### 3.12.1 Pros and cons

Parallel arrays have a number of practical advantages over the normal approach:

- They can be used in languages which support only arrays of primitive types and not of records (or perhaps don't support records at all).
- Parallel arrays are simple to understand and use, and

are often used where declaring a record is more trouble than it's worth.

- They can save a substantial amount of space in some cases by avoiding alignment issues. For example, one of the fields of the record can be a single bit, and its array would only need to reserve one bit for each record, whereas in the normal approach many more bits would “pad” the field so that it consumes an entire byte or a word.
- If the number of items is small, array indices can occupy significantly less space than full pointers, particularly on architectures with large words.
- Sequentially examining a single field of each record in the array is very fast on modern machines, since this amounts to a linear traversal of a single array, exhibiting ideal locality of reference and cache behavior.

However, parallel arrays also have several strong disadvantages, which serves to explain why they are not generally preferred:

- They have significantly worse locality of reference when visiting the records non-sequentially and examining multiple fields of each record.
- They obscure the relationship between fields of a single record.
- They have little direct language support (the language and its syntax typically express no relationship between the arrays in the parallel array).
- They are expensive to grow or shrink, since each of several arrays must be reallocated. Multi-level arrays can ameliorate this problem, but impacts performance due to the additional indirection needed to find the desired elements.

The bad locality of reference can be alleviated in some cases: if a structure can be divided into groups of fields that are generally accessed together, an array can be constructed for each group, and its elements are records containing only these subsets of the larger structure's fields. This is a valuable way of speeding up access to very large structures with many members, while keeping the portions of the structure tied together. An alternative to tying them together using array indexes is to use references to tie the portions together, but this can be less efficient in time and space. Another alternative is to mock up a record structure in a single-dimensional array by declaring an array of  $n*m$  size and referring to the  $r$ -th field in record  $i$  as element as  $\text{array}(m*i+r)$ . Some compiler optimizations, particularly for vector processors, are able to perform this transformation automatically when arrays of structures are created in the program.

### 3.12.2 See also

- An example in the linked list article
- Column-oriented DBMS

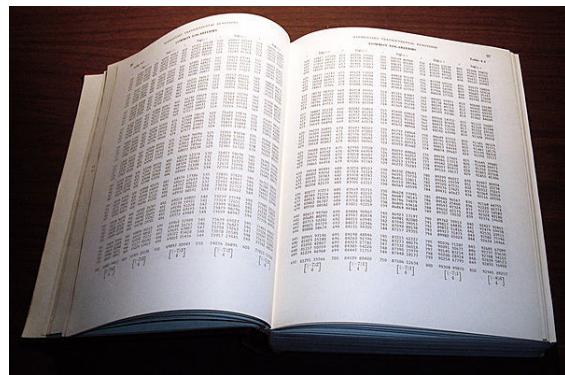
### 3.12.3 References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Page 209 of section 10.3: Implementing pointers and objects.
- Skeet, Jon (3 June 2014). “Anti-pattern: parallel collections”. Retrieved 28 October 2014.

## 3.13 Lookup table

In computer science, a **lookup table** is an array that replaces runtime computation with a simpler array indexing operation. The savings in terms of processing time can be significant, since retrieving a value from memory is often faster than undergoing an 'expensive' computation or input/output operation.<sup>[1]</sup> The tables may be precalculated and stored in static program storage, calculated (or “pre-fetched”) as part of a program’s initialization phase (memoization), or even stored in hardware in application-specific platforms. Lookup tables are also used extensively to validate input values by matching against a list of valid (or invalid) items in an array and, in some programming languages, may include pointer functions (or offsets to labels) to process the matching input.

### 3.13.1 History



Part of a 20th-century table of common logarithms in the reference book Abramowitz and Stegun.

Before the advent of computers, lookup tables of values were used to speed up hand calculations of complex functions, such as in trigonometry, logarithms, and statistical density functions<sup>[2]</sup>

In ancient (499 CE) India, **Aryabhata** created one of the first **sine tables**, which he encoded in a Sanskrit-letter-based number system. In 493 A.D., **Victorius of Aquitaine** wrote a 98-column multiplication table which gave (in **Roman numerals**) the product of every number from 2 to 50 times and the rows were “a list of numbers starting with one thousand, descending by hundreds to one hundred, then descending by tens to ten, then by ones to one, and then the fractions down to 1/144” [3]. Modern school children are often taught to memorize “times tables” to avoid calculations of the most commonly used numbers (up to  $9 \times 9$  or  $12 \times 12$ ).

Early in the history of computers, **input/output** operations were particularly slow – even in comparison to processor speeds of the time. It made sense to reduce expensive read operations by a form of manual **caching** by creating either static lookup tables (embedded in the program) or dynamic prefetched arrays to contain only the most commonly occurring data items. Despite the introduction of systemwide caching that now automates this process, application level lookup tables can still improve performance for data items that rarely, if ever, change.

### 3.13.2 Examples

#### Simple lookup in an array, an associative array or a linked list (unsorted list)

This is known as a **linear search** or **brute-force search**, each element being checked for equality in turn and the associated value, if any, used as a result of the search. This is often the slowest search method unless frequently occurring values occur early in the list. For a one-dimensional array or **linked list**, the lookup is usually to determine whether or not there is a match with an ‘input’ data value.

#### Binary search in an array or an associative array (sorted list)

An example of a “**divide and conquer algorithm**”, **binary search** involves each element being found by determining which half of the table a match may be found in and repeating until either success or failure. Only possible if the list is sorted but gives good performance even if the list is lengthy.

#### Trivial hash function

For a **trivial hash function** lookup, the unsigned **raw data** value is used *directly* as an index to a one-dimensional table to extract a result. For small ranges, this can be amongst the fastest lookup, even exceeding binary search speed with zero branches and executing in **constant time**.

**Counting bits in a series of bytes** One discrete problem that is expensive to solve on many computers, is that of counting the number of bits which are set to 1 in a (binary) number, sometimes called the **population function**. For example, the decimal number “37” is “00100101” in binary, so it contains three bits that are set to binary “1”.

A simple example of C code, designed to count the 1 bits in a **int**, might look like this:

```
int count_ones(unsigned int x) { int result = 0; while (x != 0) result++, x = x & (x-1); return result; }
```

This apparently simple algorithm can take potentially hundreds of cycles even on a modern architecture, because it makes many branches in the loop - and branching is slow. This can be ameliorated using **loop unrolling** and some other compiler optimizations. There is however a simple and much faster algorithmic solution - using a **trivial hash function** table lookup.

Simply construct a static table, **bits\_set**, with 256 entries giving the number of one bits set in each possible byte value (e.g.  $0x00 = 0$ ,  $0x01 = 1$ ,  $0x02 = 1$ , and so on). Then use this table to find the number of ones in each byte of the integer using a **trivial hash function** lookup on each byte in turn, and sum them. This requires no branches, and just four indexed memory accesses, considerably faster than the earlier code.

```
/* (this code assumes that 'int' is 32-bits wide) */ int
count_ones(unsigned int x) { return bits_set[x & 255] +
bits_set[(x >> 8) & 255] + bits_set[(x >> 16) & 255] +
bits_set[(x >> 24) & 255]; }
```

The above source can be improved easily, (avoiding AND’ing, and shifting) by ‘recasting’ ‘x’ as a 4 byte unsigned char array and, preferably, coded in-line as a single statement instead of being a function. Note that even this simple algorithm can be too slow now, because the original code might run faster from the cache of modern processors, and (large) lookup tables do not fit well in caches and can cause a slower access to memory (in addition, in the above example, it requires computing addresses within a table, to perform the four lookups needed).

#### Lookup tables in image processing

In data analysis applications, such as **image processing**, a **lookup table** (LUT) is used to transform the input data into a more desirable output format. For example, a grayscale picture of the planet Saturn will be transformed into a color image to emphasize the differences in its rings.

A classic example of reducing run-time computations using lookup tables is to obtain the result of a **trigonometry** calculation, such as the **sine** of a value. Calculating trigonometric functions can substantially slow a computing application. The same application can finish much

sooner when it first precalculates the sine of a number of values, for example for each whole number of degrees (The table can be defined as static variables at compile time, reducing repeated run time costs). When the program requires the sine of a value, it can use the lookup table to retrieve the closest sine value from a memory address, and may also take the step of interpolating to the sine of the desired value, instead of calculating by mathematical formula. Lookup tables are thus used by mathematics co-processors in computer systems. An error in a lookup table was responsible for Intel's infamous floating-point divide bug.

Functions of a single variable (such as sine and cosine) may be implemented by a simple array. Functions involving two or more variables require multidimensional array indexing techniques. The latter case may thus employ a two-dimensional array of **power[x][y]** to replace a function to calculate  $x^y$  for a limited range of x and y values. Functions that have more than one result may be implemented with lookup tables that are arrays of structures.

As mentioned, there are intermediate solutions that use tables in combination with a small amount of computation, often using **interpolation**. Pre-calculation combined with interpolation can produce higher accuracy for values that fall between two precomputed values. This technique requires slightly more time to be performed but can greatly enhance accuracy in applications that require the higher accuracy. Depending on the values being pre-computed, pre-computation with interpolation can also be used to shrink the lookup table size while maintaining accuracy.

In **image processing**, lookup tables are often called **LUTs** and give an output value for each of a range of index values. One common LUT, called the *colormap* or *palette*, is used to determine the colors and intensity values with which a particular image will be displayed. In **computed tomography**, “windowing” refers to a related concept for determining how to display the intensity of measured radiation.

While often effective, employing a lookup table may nevertheless result in a severe penalty if the computation that the LUT replaces is relatively simple. Memory retrieval time and the complexity of memory requirements can increase application operation time and system complexity relative to what would be required by straight formula computation. The possibility of **polluting the cache** may also become a problem. Table accesses for large tables will almost certainly cause a **cache miss**. This phenomenon is increasingly becoming an issue as processors outpace memory. A similar issue appears in **rematerialization**, a **compiler optimization**. In some environments, such as the **Java programming language**, table lookups can be even more expensive due to mandatory bounds-checking involving an additional comparison and branch for each lookup.

There are two fundamental limitations on when it is possible to construct a lookup table for a required operation. One is the amount of memory that is available: one cannot construct a lookup table larger than the space available for the table, although it is possible to construct disk-based lookup tables at the expense of lookup time. The other is the time required to compute the table values in the first instance; although this usually needs to be done only once, if it takes a prohibitively long time, it may make the use of a lookup table an inappropriate solution. As previously stated however, tables can be statically defined in many cases.

### Computing sines

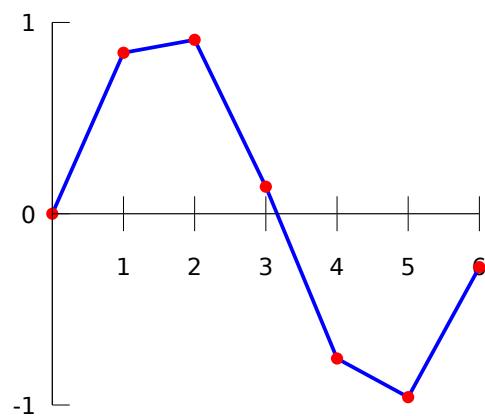
Most computers, which only perform basic arithmetic operations, cannot directly calculate the **sine** of a given value. Instead, they use the **CORDIC** algorithm or a complex formula such as the following **Taylor series** to compute the value of sine to a high degree of precision:

$$\sin(x) \approx x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} \text{ (for } x \text{ close to 0)}$$

However, this can be expensive to compute, especially on slow processors, and there are many applications, particularly in traditional **computer graphics**, that need to compute many thousands of sine values every second. A common solution is to initially compute the sine of many evenly distributed values, and then to find the sine of  $x$  we choose the sine of the value closest to  $x$ . This will be close to the correct value because sine is a **continuous function** with a bounded rate of change. For example:

```
real array sine_table[-1000..1000] for x from -1000 to 1000 sine_table[x] := sine(pi * x / 1000) function lookup_sine(x) return sine_table[round(1000 * x / pi)]
```

Unfortunately, the table requires quite a bit of space: if



Linear interpolation on a portion of the sine function

IEEE double-precision floating-point numbers are used, over 16,000 bytes would be required. We can use fewer

samples, but then our precision will significantly worsen. One good solution is **linear interpolation**, which draws a line between the two points in the table on either side of the value and locates the answer on that line. This is still quick to compute, and much more accurate for **smooth functions** such as the sine function. Here is our example using linear interpolation:

```
function lookup_sine(x) x1 := floor(x*1000/pi) y1 := sine_table[x1] y2 := sine_table[x1+1] return y1 + (y2-y1)*(x*1000/pi-x1)
```

Another solution that uses a quarter of the space but takes a bit longer to compute would be to take into account the relationships between sine and cosine along with their symmetry rules. In this case, the lookup table is calculated by using the sine function for the first quadrant (i.e.  $\sin(0..pi/2)$ ). When we need a value, we assign a variable to be the angle wrapped to the first quadrant. We then wrap the angle to the four quadrants (not needed if values are always between 0 and  $2*pi$ ) and return the correct value (i.e. first quadrant is a straight return, second quadrant is read from  $pi/2-x$ , third and fourth are negatives of the first and second respectively). For cosine, we only have to return the angle shifted by  $pi/2$  (i.e.  $x+pi/2$ ). For tangent, we divide the sine by the cosine (divide-by-zero handling may be needed depending on implementation):

```
function init_sine() for x from 0 to (360/4)+1 sine_table[x] := sine(2*pi * x / 360) function lookup_sine(x) x = wrap x from 0 to 360 y := mod(x, 90) if (x < 90) return sine_table[y] if (x < 180) return sine_table[90-y] if (x < 270) return -sine_table[y] return -sine_table[90-y] function lookup_cosine(x) return lookup_sine(x + 90) function lookup_tan(x) return (lookup_sine(x) / lookup_cosine(x))
```

When using interpolation, the size of the lookup table can be reduced by using **nonuniform sampling**, which means that where the function is close to straight, we use few sample points, while where it changes value quickly we use more sample points to keep the approximation close to the real curve. For more information, see **interpolation**.

### 3.13.3 Other usage of lookup tables

#### Caches

Main article: **Cache (computing)**

Storage caches (including disk caches for files, or processor caches for either code or data) work also like a lookup table. The table is built with very fast memory instead of being stored on slower external memory, and maintains two pieces of data for a subrange of bits composing an external memory (or disk) address (notably the lowest bits of any possible external address):

- one piece (the tag) contains the value of the remain-

ing bits of the address; if these bits match with those from the memory address to read or write, then the other piece contains the cached value for this address.

- the other piece maintains the data associated to that address.

A single (fast) lookup is performed to read the tag in the lookup table at the index specified by the lowest bits of the desired external storage address, and to determine if the memory address is hit by the cache. When a hit is found, no access to external memory is needed (except for write operations, where the cached value may need to be updated asynchronously to the slower memory after some time, or if the position in the cache must be replaced to cache another address).

#### Hardware LUTs

In **digital logic**, an  $n$ -bit lookup table can be implemented with a **multiplexer** whose select lines are the inputs of the LUT and whose inputs are constants. An  $n$ -bit LUT can encode any  $n$ -input Boolean function by modeling such functions as **truth tables**. This is an efficient way of encoding Boolean logic functions, and LUTs with 4-6 bits of input are in fact the key component of modern **field-programmable gate arrays (FPGAs)**.

#### 3.13.4 See also

- **TheEgoWorks LUTs** A LUT emulating the Hollywood epic teal-orange look.
- **Branch table**
- **Gal's accurate tables**
- **Memoization**
- **Memory bound function**
- **Shift register lookup table**
- **Palette and Colour Look-Up Table** – for the usage in computer graphics
- **3D LUT** – usage in film

#### 3.13.5 References

- [1] <http://pmcnamee.net/c++-memoization.html>
- [2] Campbell-Kelly, Martin; Croarken, Mary; Robson, Eleanor, eds. (October 2, 2003) [2003]. *The History of Mathematical Tables From Sumer to Spreadsheets* (1st ed.). New York, USA. ISBN 978-0-19-850841-0.
- [3] Maher, David. W. J. and John F. Makowski. "Literary Evidence for Roman Arithmetic With Fractions", 'Classical Philology' (2001) Vol. 96 No. 4 (2001) pp. 376–399. (See page p.383.)

### 3.13.6 External links

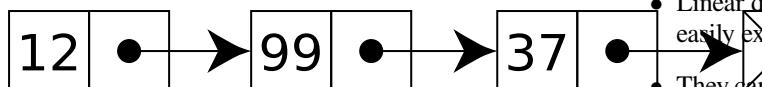
- Fast table lookup using input character as index for branch table
- Art of Assembly: Calculation via Table Lookups
- “Bit Twiddling Hacks” (includes lookup tables) By Sean Eron Anderson of Stanford university
- Memoization in C++ by Paul McNamee, Johns Hopkins University showing savings
- “The Quest for an Accelerated Population Count” by Henry S. Warren, Jr.

# Chapter 4

## Lists

### 4.1 Linked list

In computer science, a **linked list** is a data structure consisting of a group of **nodes** which together represent a sequence. Under the simplest form, each node is composed of data and a **reference** (in other words, a *link*) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence. Linear collection of data elements is called node pointers



A *linked list* whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including **lists** (the abstract data type), **stacks**, **queues**, **associative arrays**, and **S-expressions**, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

The principal benefit of a linked list over a conventional **array** is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while an array has to be declared in the source code, before compiling and running the program. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow **random access** to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list (assuming that the last node is not maintained as separate node reference in the list structure), or finding a node that contains a given datum, or locating the place where a new node should be

inserted — may require sequential scanning of most or all of the list elements. The advantages and disadvantages of using linked lists are given below.

#### 4.1.1 Advantages

- Linked lists are a dynamic data structure, allocating the needed memory while the program is running.
- Insertion and deletion node operations are easily implemented in a linked list.
- Linear data structures such as stacks and queues are easily executed with a linked list.
- They can reduce access time and may expand in real time without memory overhead.

#### 4.1.2 Disadvantages

- They have a tendency to use more memory due to **pointers** requiring extra storage space.
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently **sequential access**.
- Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list.
- Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards<sup>[1]</sup> and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.

#### 4.1.3 History

Linked lists were developed in 1955–1956 by Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation as the primary data structure for their **Information Processing Language**. IPL was used by the authors to develop several early artificial intelligence programs, including the **Logic Theory Machine**, the **General**

Problem Solver, and a computer chess program. Reports on their work appeared in IRE Transactions on Information Theory in 1956, and several conference proceedings from 1957 to 1959, including Proceedings of the Western Joint Computer Conference in 1957 and 1958, and Information Processing (Proceedings of the first UNESCO International Conference on Information Processing) in 1959. The now-classic diagram consisting of blocks representing list nodes with arrows pointing to successive list nodes appears in “Programming the Logic Theory Machine” by Newell and Shaw in Proc. WJCC, February 1957. Newell and Simon were recognized with the ACM Turing Award in 1975 for having “made basic contributions to artificial intelligence, the psychology of human cognition, and list processing”. The problem of machine translation for natural language processing led Victor Yngve at Massachusetts Institute of Technology (MIT) to use linked lists as data structures in his COMIT programming language for computer research in the field of linguistics. A report on this language entitled “A programming language for mechanical translation” appeared in Mechanical Translation in 1958.

LISP, standing for list processor, was created by John McCarthy in 1958 while he was at MIT and in 1960 he published its design in a paper in the Communications of the ACM, entitled “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. One of LISP’s major data structures is the linked list.

By the early 1960s, the utility of both linked lists and languages which use these structures as their primary data representation was well established. Bert Green of the MIT Lincoln Laboratory published a review article entitled “Computer languages for symbol manipulation” in IRE Transactions on Human Factors in Electronics in March 1961 which summarized the advantages of the linked list approach. A later review article, “A Comparison of list-processing computer languages” by Bobrow and Raphael, appeared in Communications of the ACM in April 1964.

Several operating systems developed by Technical Systems Consultants (originally of West Lafayette Indiana, and later of Chapel Hill, North Carolina) used singly linked lists as file structures. A directory entry pointed to the first sector of a file, and succeeding portions of the file were located by traversing pointers. Systems using this technique included Flex (for the Motorola 6800 CPU), mini-Flex (same CPU), and Flex9 (for the Motorola 6809 CPU). A variant developed by TSC for and marketed by Smoke Signal Broadcasting in California, used doubly linked lists in the same manner.

The TSS/360 operating system, developed by IBM for the System 360/370 machines, used a double linked list for their file system catalog. The directory structure was similar to Unix, where a directory could contain files and other directories and extend to any depth.

#### 4.1.4 Basic concepts and nomenclature

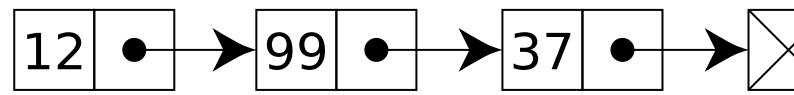
Each record of a linked list is often called an ‘element’ or ‘node’.

The field of each node that contains the address of the next node is usually called the ‘next link’ or ‘next pointer’. The remaining fields are known as the ‘data’, ‘information’, ‘value’, ‘cargo’, or ‘payload’ fields.

The ‘head’ of a list is its first node. The ‘tail’ of a list may refer either to the rest of the list after the head, or to the last node in the list. In **Lisp** and some derived languages, the next node may be called the ‘cdr’ (pronounced *could-er*) of the list, while the payload of the head node may be called the ‘car’.

##### Singly linked list

Singly linked lists contain nodes which have a data field as well as a ‘next’ field, which points to the next node in line of nodes. Operations that can be performed on singly linked lists include insertion, deletion and traversal.

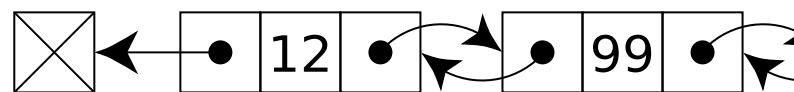


A singly linked list whose nodes contain two fields: an integer value and a link to the next node

##### Doubly linked list

Main article: **Doubly linked list**

In a ‘doubly linked list’, each node contains, besides the next-node link, a second link field pointing to the ‘previous’ node in the sequence. The two links may be called ‘forward(s)’ and ‘backwards’, or ‘next’ and ‘prev’(‘previous’).



A doubly linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

A technique known as **XOR-linking** allows a doubly linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level languages.

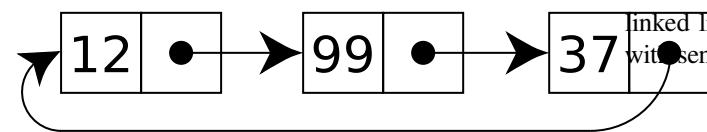
##### Multiply linked list

In a ‘multiply linked list’, each node contains two or more link fields, each field being used to connect the same set

of data records in a different order (e.g., by name, by department, by date of birth, etc.). While doubly linked lists can be seen as special cases of multiply linked list, the fact that the two orders are opposite to each other leads to simpler and more efficient algorithms, so they are usually treated as a separate case.

### Circular Linked list

In the last **node** of a list, the link field often contains a **null** reference, a special value used to indicate the lack of further nodes. A less common convention is to make it point to the first node of the list; in that case the list is said to be 'circular' or 'circularly linked'; otherwise it is said to be 'open' or 'linear'.



A circular linked list

In the case of a circular doubly linked list, the only change that occurs is that the end, or "tail", of the said list is linked back to the front, or "head", of the list and vice versa.

### Sentinel nodes

Main article: [Sentinel node](#)

In some implementations, an extra 'sentinel' or 'dummy' node may be added before the first data record or after the last one. This convention simplifies and accelerates some list-handling algorithms, by ensuring that all links can be safely dereferenced and that every list (even one that contains no data elements) always has a "first" and "last" node.

### Empty lists

An empty list is a list that contains no data records. This is usually the same as saying that it has zero nodes. If sentinel nodes are being used, the list is usually said to be empty when it has only sentinel nodes.

### Hash linking

The link fields need not be physically part of the nodes. If the data records are stored in an array and referenced by their indices, the link field may be stored in a separate array with the same indices as the data records.

### List handles

Since a reference to the first node gives access to the whole list, that reference is often called the 'address', 'pointer', or 'handle' of the list. Algorithms that manipulate linked lists usually get such handles to the input lists and return the handles to the resulting lists. In fact, in the context of such algorithms, the word "list" often means "list handle". In some situations, however, it may be convenient to refer to a list by a handle that consists of two links, pointing to its first and last nodes.

### Combining alternatives

The alternatives listed above may be arbitrarily combined in almost every way, so one may have circular doubly linked lists without sentinels, circular singly linked lists with sentinels, etc.

### 4.1.5 Tradeoffs

As with most choices in computer programming and design, no method is well suited to all circumstances. A linked list data structure might work well in one case, but cause problems in another. This is a list of some of the common tradeoffs involving linked list structures.

### Linked lists vs. dynamic arrays

A **dynamic array** is a data structure that allocates all elements contiguously in memory, and keeps a count of the current number of elements. If the space reserved for the dynamic array is exceeded, it is reallocated and (possibly) copied, an expensive operation.

Linked lists have several advantages over dynamic arrays. Insertion or deletion of an element at a specific point of a list, assuming that we have indexed a pointer to the node (before the one to be removed, or before the insertion point) already, is a constant-time operation (otherwise without this reference it is  $O(n)$ ), whereas insertion in a dynamic array at random locations will require moving half of the elements on average, and all the elements in the worst case. While one can "delete" an element from an array in constant time by somehow marking its slot as "vacant", this causes **fragmentation** that impedes the performance of iteration.

Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while a dynamic array will eventually fill up its underlying array data structure and will have to reallocate — an expensive operation, one that may not even be possible if memory is fragmented, although the cost of reallocation can be averaged over insertions, and the cost of an insertion due to reallocation would still be **amortized**  $O(1)$ . This helps with appending elements at the array's

end, but inserting into (or removing from) middle positions still carries prohibitive costs due to data moving to maintain contiguity. An array from which many elements are removed may also have to be resized in order to avoid wasting too much space.

On the other hand, dynamic arrays (as well as fixed-size [array data structures](#)) allow constant-time random access, while linked lists allow only [sequential access](#) to elements. Singly linked lists, in fact, can be easily traversed in only one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as [heapsort](#). Sequential access on arrays and dynamic arrays is also faster than on linked lists on many machines, because they have optimal [locality of reference](#) and thus make good use of data caching.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as [characters](#) or [boolean values](#), because the storage overhead for the links may exceed by a factor of two or more the size of the data. In contrast, a dynamic array requires only the space for the data itself (and a very small amount of control data).<sup>[note 1]</sup> It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using [memory pools](#).

Some hybrid solutions try to combine the advantages of the two representations. [Unrolled linked lists](#) store several elements in each list node, increasing cache performance while decreasing memory overhead for references. [CDR coding](#) does both these as well, by replacing references with the actual data referenced, which extends off the end of the referencing record.

A good example that highlights the pros and cons of using dynamic arrays vs. linked lists is by implementing a program that resolves the [Josephus problem](#). The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle  $n$  times. Once you reach the  $n$ th person, take them out of the circle and have the members close the circle. Then count around the circle the same  $n$  times and repeat the process, until only one person is left. That person wins the election. This shows the strengths and weaknesses of a linked list vs. a dynamic array, because if you view the people as connected nodes in a circular linked list then it shows how easily the linked list is able to delete nodes (as it only has to rearrange the links to the different nodes). However, the linked list will be poor at finding the next person to remove and will need to search through the list until it finds that person. A dynamic array, on the other hand, will be poor at deleting nodes (or elements) as it cannot remove one node without individually shifting all the elements up the list by one. However, it is exceptionally easy to find the  $n$ th person in the circle by directly referencing them by their position in the array.

The [list ranking](#) problem concerns the efficient conver-

sion of a linked list representation into an array. Although trivial for a conventional computer, solving this problem by a [parallel algorithm](#) is complicated and has been the subject of much research.

A [balanced tree](#) has similar memory access patterns and space overhead to a linked list while permitting much more efficient indexing, taking  $O(\log n)$  time instead of  $O(n)$  for a random access. However, insertion and deletion operations are more expensive due to the overhead of tree manipulations to maintain balance. Schemes exist for trees to automatically maintain themselves in a balanced state: [AVL trees](#) or [red-black trees](#).

### Singly linked linear lists vs. other lists

While doubly-linked and circular lists have advantages over singly-linked linear lists, linear lists offer some advantages that make them preferable in some situations.

A singly linked linear list is a [recursive data structure](#), because it contains a pointer to a *smaller* object of the same type. For that reason, many operations on singly linked linear lists (such as [merging](#) two lists, or enumerating the elements in reverse order) often have very simple recursive algorithms, much simpler than any solution using [iterative commands](#). While those recursive solutions can be adapted for doubly linked and circularly linked lists, the procedures generally need extra arguments and more complicated base cases.

Linear singly linked lists also allow [tail-sharing](#), the use of a common final portion of sub-list as the terminal portion of two different lists. In particular, if a new node is added at the beginning of a list, the former list remains available as the tail of the new one — a simple example of a [persistent data structure](#). Again, this is not true with the other variants: a node may never belong to two different circular or doubly linked lists.

In particular, end-sentinel nodes can be shared among singly linked non-circular lists. The same end-sentinel node may be used for *every* such list. In [Lisp](#), for example, every proper list ends with a link to a special node, denoted by `nil` or `()`, whose [CAR](#) and [CDR](#) links point to itself. Thus a Lisp procedure can safely take the [CAR](#) or [CDR](#) of *any* list.

The advantages of the fancy variants are often limited to the complexity of the algorithms, not in their efficiency. A circular list, in particular, can usually be emulated by a linear list together with two variables that point to the first and last nodes, at no extra cost.

### Doubly linked vs. singly linked

Double-linked lists require more space per node (unless one uses [XOR-linking](#)), and their elementary operations are more expensive; but they are often easier to manipulate because they allow fast and easy sequential access

to the list in both directions. In a doubly linked list, one can insert or delete a node in a constant number of operations given only that node's address. To do the same in a singly linked list, one must have the *address of the pointer* to that node, which is either the handle for the whole list (in case of the first node) or the link field in the *previous* node. Some algorithms require access in both directions. On the other hand, doubly linked lists do not allow tail-sharing and cannot be used as **persistent** data structures.

### Circularly linked vs. linearly linked

A circularly linked list may be a natural option to represent arrays that are naturally circular, e.g. the corners of a polygon, a pool of buffers that are used and released in FIFO (“first in, first out”) order, or a set of processes that should be **time-shared in round-robin order**. In these applications, a pointer to any node serves as a handle to the whole list.

With a circular list, a pointer to the last node gives easy access also to the first node, by following one link. Thus, in applications that require access to both ends of the list (e.g., in the implementation of a queue), a circular structure allows one to handle the structure by a single pointer, instead of two.

A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. The operation consists in swapping the contents of the link fields of those two nodes. Applying the same operation to any two nodes in two distinct lists joins the two list into one. This property greatly simplifies some algorithms and data structures, such as the **quad-edge** and **face-edge**.

The simplest representation for an empty *circular* list (when such a thing makes sense) is a null pointer, indicating that the list has no nodes. Without this choice, many algorithms have to test for this special case, and handle it separately. By contrast, the use of null to denote an empty *linear* list is more natural and often creates fewer special cases.

### Using sentinel nodes

Sentinel node may simplify certain list operations, by ensuring that the next or previous nodes exist for every element, and that even empty lists have at least one node. One may also use a sentinel node at the end of the list, with an appropriate data field, to eliminate some end-of-list tests. For example, when scanning the list looking for a node with a given value  $x$ , setting the sentinel's data field to  $x$  makes it unnecessary to test for end-of-list inside the loop. Another example is the merging two sorted lists: if their sentinels have data fields set to  $+\infty$ , the choice of the next output node does not need special handling for empty lists.

However, sentinel nodes use up extra space (especially in applications that use many short lists), and they may complicate other operations (such as the creation of a new empty list).

However, if the circular list is used merely to simulate a linear list, one may avoid some of this complexity by adding a single sentinel node to every list, between the last and the first data nodes. With this convention, an empty list consists of the sentinel node alone, pointing to itself via the next-node link. The list handle should then be a pointer to the last data node, before the sentinel, if the list is not empty; or to the sentinel itself, if the list is empty.

The same trick can be used to simplify the handling of a doubly linked linear list, by turning it into a circular doubly linked list with a single sentinel node. However, in this case, the handle should be a single pointer to the dummy node itself.<sup>[6]</sup>

### 4.1.6 Linked list operations

When manipulating linked lists in-place, care must be taken to not use values that you have invalidated in previous assignments. This makes algorithms for inserting or deleting linked list nodes somewhat subtle. This section gives **pseudocode** for adding or removing nodes from singly, doubly, and circularly linked lists in-place. Throughout we will use *null* to refer to an end-of-list marker or **sentinel**, which may be implemented in a number of ways.

#### Linearly linked lists

**Singly linked lists** Our node data structure will have two fields. We also keep a variable *firstNode* which always points to the first node in the list, or is *null* for an empty list.

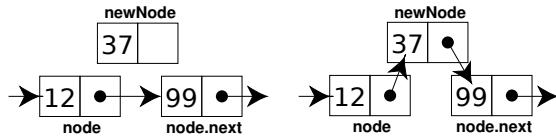
```
record Node { data; // The data being stored in the node
 Node next // A reference to the next node, null for last node
} record List { Node firstNode // points to first node of list;
 null for empty list }
```

Traversal of a singly linked list is simple, beginning at the first node and following each *next* link until we come to the end:

```
node := list.firstNode while node not null (do something
 with node.data) node := node.next
```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done directly; instead, one must keep track of the previous node and insert a node after it.

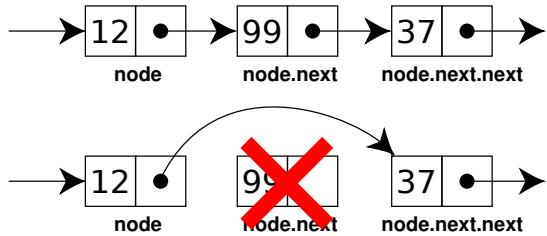
```
function insertAfter(Node node, Node newNode) // insert
 newNode after node
 newNode.next := node.next
 node.next := newNode
```



Inserting at the beginning of the list requires a separate function. This requires updating `firstNode`.

```
function insertBeginning(List list, Node newNode) // insert node before current first node
newNode.next := list.firstNode
list.firstNode := newNode
```

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.



```
function removeAfter(Node node) // remove node past this one
obsoleteNode := node.next
node.next := node.next.next
destroy obsoleteNode
function removeBeginning(List list) // remove first node
obsoleteNode := list.firstNode
list.firstNode := list.firstNode.next // point past deleted node
destroy obsoleteNode
```

Notice that `removeBeginning()` sets `list.firstNode` to null when removing the last node in the list.

Since we can't iterate backwards, efficient `insertBefore` or `removeBefore` operations are not possible.

Appending one linked list to another can be inefficient unless a reference to the tail is kept as part of the List structure, because we must traverse the entire first list in order to find the tail, and then append the second list to this. Thus, if two linearly linked lists are each of length  $n$ , list appending has **asymptotic time complexity** of  $O(n)$ . In the Lisp family of languages, list appending is provided by the `append` procedure.

Many of the special cases of linked list operations can be eliminated by including a dummy element at the front of the list. This ensures that there are no special cases for the beginning of the list and renders both `insertBeginning()` and `removeBeginning()` unnecessary. In this case, the first useful data in the list will be found at `list.firstNode.next`.

### Circularly linked list

In a circularly linked list, all nodes are linked in a continuous circle, without using `null`. For lists with a front and a back (such as a queue), one stores a reference to the last node in the list. The `next` node after the last node is the first node. Elements can be added to the back of the list and removed from the front in constant time.

Circularly linked lists can be either singly or doubly linked.

Both types of circularly linked lists benefit from the ability to traverse the full list beginning at any given node. This often allows us to avoid storing `firstNode` and `lastNode`, although if the list may be empty we need a special representation for the empty list, such as a `lastNode` variable which points to some node in the list or is `null` if it's empty; we use such a `lastNode` here. This representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case.

**Algorithms** Assuming that `someNode` is some node in a non-empty circular singly linked list, this code iterates through that list starting with `someNode`:

```
function iterate(someNode) if someNode ≠ null
node := someNode do something with node.value
node := node.next while node ≠ someNode
```

Notice that the test "`while node ≠ someNode`" must be at the end of the loop. If the test was moved to the beginning of the loop, the procedure would fail whenever the list had only one node.

This function inserts a node "newNode" into a circular linked list after a given node "node". If "node" is null, it assumes that the list is empty.

```
function insertAfter(Node node, Node newNode) if node = null
newNode.next := newNode else
newNode.next := node.next
node.next := newNode
```

Suppose that "L" is a variable pointing to the last node of a circular linked list (or null if the list is empty). To append "newNode" to the *end* of the list, one may do

```
insertAfter(L, newNode) L := newNode
```

To insert "newNode" at the *beginning* of the list, one may do

```
insertAfter(L, newNode) if L = null L := newNode
```

#### 4.1.7 Linked lists using arrays of node

Languages that do not support any type of reference can still create links by replacing pointers with array indices. The approach is to keep an **array of records**, where each record has integer fields indicating the index of the next (and possibly previous) node in the array. Not all nodes in the array need be used. If records are also not supported, **parallel arrays** can often be used instead.

As an example, consider the following linked list record that uses arrays instead of pointers:

```
record Entry { integer next; // index of next entry in array
 integer prev; // previous entry (if double-linked)
 string name; real balance; }
```

A linked list can be build by creating an array of these structures, and an integer variable to store the index of the first element.

```
integer listHead Entry Records[1000]
```

Links between elements are formed by placing the array index of the next (or previous) cell into the Next or Prev field within a given element. For example:

In the above example, ListHead would be set to 2, the location of the first entry in the list. Notice that entry 3 and 5 through 7 are not part of the list. These cells are available for any additions to the list. By creating a ListFree integer variable, a **free list** could be created to keep track of what cells are available. If all entries are in use, the size of the array would have to be increased or some elements would have to be deleted before new entries could be stored in the list.

The following code would traverse the list and display names and account balance:

```
i := listHead
while i >= 0 // loop through the list
print i, Records[i].name, Records[i].balance // print entry
i := Records[i].next
```

When faced with a choice, the advantages of this approach include:

- The linked list is relocatable, meaning it can be moved about in memory at will, and it can also be quickly and directly **serialized** for storage on disk or transfer over a network.
- Especially for a small list, array indexes can occupy significantly less space than a full pointer on many architectures.
- **Locality of reference** can be improved by keeping the nodes together in memory and by periodically rearranging them, although this can also be done in a general store.
- Naïve **dynamic memory allocators** can produce an excessive amount of overhead storage for each node allocated; almost no allocation overhead is incurred per node in this approach.
- Seizing an entry from a pre-allocated array is faster than using dynamic memory allocation for each node, since dynamic memory allocation typically requires a search for a free memory block of the desired size.

This approach has one main disadvantage, however: it creates and manages a private memory space for its nodes. This leads to the following issues:

- It increases complexity of the implementation.
- Growing a large array when it is full may be difficult or impossible, whereas finding space for a new linked list node in a large, general memory pool may be easier.
- Adding elements to a dynamic array will occasionally (when it is full) unexpectedly take linear ( $O(n)$ ) instead of constant time (although it's still an amortized constant).
- Using a general memory pool leaves more memory for other data if the list is smaller than expected or if many nodes are freed.

For these reasons, this approach is mainly used for languages that do not support dynamic memory allocation. These disadvantages are also mitigated if the maximum size of the list is known at the time the array is created.

#### 4.1.8 Language support

Many **programming languages** such as **Lisp** and **Scheme** have singly linked lists built in. In many **functional languages**, these lists are constructed from nodes, each called a **cons** or **cons cell**. The cons has two fields: the **car**, a reference to the data for that node, and the **cdr**, a reference to the next node. Although cons cells can be used to build other data structures, this is their primary purpose.

In languages that support **abstract data types** or templates, linked list **ADTs** or templates are available for building linked lists. In other languages, linked lists are typically built using **references** together with **records**.

#### 4.1.9 Internal and external storage

When constructing a linked list, one is faced with the choice of whether to store the data of the list directly in the linked list nodes, called **internal storage**, or merely to store a reference to the data, called **external storage**. Internal storage has the advantage of making access to the data more efficient, requiring less storage overall, having better **locality of reference**, and simplifying memory management for the list (its data is allocated and deallocated at the same time as the list nodes).

External storage, on the other hand, has the advantage of being more generic, in that the same data structure and machine code can be used for a linked list no matter what the size of the data is. It also makes it easy to place the same data in multiple linked lists. Although with internal storage the same data can be placed in multiple lists by including multiple **next** references in the node data structure, it would then be necessary to create separate routines to add or delete cells based on each field. It is possible to create additional linked lists of elements that use internal storage by using external storage, and having the

cells of the additional linked lists store references to the nodes of the linked list containing the data.

In general, if a set of data structures needs to be included in linked lists, external storage is the best approach. If a set of data structures need to be included in only one linked list, then internal storage is slightly better, unless a generic linked list package using external storage is available. Likewise, if different sets of data that can be stored in the same data structure are to be included in a single linked list, then internal storage would be fine.

Another approach that can be used with some languages involves having different data structures, but all have the initial fields, including the *next* (and *prev* if double linked list) references in the same location. After defining separate structures for each type of data, a generic structure can be defined that contains the minimum amount of data shared by all the other structures and contained at the top (beginning) of the structures. Then generic routines can be created that use the minimal structure to perform linked list type operations, but separate routines can then handle the specific data. This approach is often used in message parsing routines, where several types of messages are received, but all start with the same set of fields, usually including a field for message type. The generic routines are used to add new messages to a queue when they are received, and remove them from the queue in order to process the message. The message type field is then used to call the correct routine to process the specific type of message.

### Example of internal and external storage

Suppose you wanted to create a linked list of families and their members. Using internal storage, the structure might look like the following:

```
record member { // member of a family member next;
string firstName; integer age; } record family { // the
family itself family next; string lastName; string address;
member members //head of list of members of this family
}
```

To print a complete list of families and their members using internal storage, we could write:

```
aFamily := Families // start at head of families list while
aFamily ≠ null // loop through list of families print
information about family aMember := aFamily.members //get
head of list of this family's members while aMember
≠ null // loop through list of members print information
about member aMember := aMember.next aFamily := aFamily.next
```

Using external storage, we would create the following structures:

```
record node { // generic link structure node next; pointer
data // generic pointer for data at node } record member
{ // structure for family member string firstName; integer
age } record family { // structure for family string last-
```

```
Name; string address; node members // head of list of
members of this family }
```

To print a complete list of families and their members using external storage, we could write:

```
famNode := Families // start at head of families list
while famNode ≠ null // loop through list of families
aFamily := (family) famNode.data // extract family from
node print information about family memNode := aFamily.members //get
list of family members while memNode ≠ null // loop
through list of members aMember := (member)memNode.data // extract
member from node print information about member memNode := memNode.next
famNode := famNode.next
```

Notice that when using external storage, an extra step is needed to extract the record from the node and cast it into the proper data type. This is because both the list of families and the list of members within the family are stored in two linked lists using the same data structure (*node*), and this language does not have parametric types.

As long as the number of families that a member can belong to is known at compile time, internal storage works fine. If, however, a member needed to be included in an arbitrary number of families, with the specific number known only at run time, external storage would be necessary.

### Speeding up search

Finding a specific element in a linked list, even if it is sorted, normally requires  $O(n)$  time ([linear search](#)). This is one of the primary disadvantages of linked lists over other data structures. In addition to the variants discussed above, below are two simple ways to improve search time.

In an unordered list, one simple heuristic for decreasing average search time is the *move-to-front heuristic*, which simply moves an element to the beginning of the list once it is found. This scheme, handy for creating simple caches, ensures that the most recently used items are also the quickest to find again.

Another common approach is to "index" a linked list using a more efficient external data structure. For example, one can build a [red-black tree](#) or [hash table](#) whose elements are references to the linked list nodes. Multiple such indexes can be built on a single list. The disadvantage is that these indexes may need to be updated each time a node is added or removed (or at least, before that index is used again).

### Random access lists

A [random access list](#) is a list with support for fast random access to read or modify any element in the list.<sup>[7]</sup> One possible implementation is a [skew binary random access list](#) using the [skew binary number system](#), which involves

a list of trees with special properties; this allows worst-case constant time head/cons operations, and worst-case logarithmic time random access to an element by index.<sup>[7]</sup> Random access lists can be implemented as persistent data structures.<sup>[7]</sup>

Random access lists can be viewed as immutable linked lists in that they likewise support the same  $O(1)$  head and tail operations.<sup>[7]</sup>

A simple extension to random access lists is the **min-list**, which provides an additional operation that yields the minimum element in the entire list in constant time (without mutation complexities).<sup>[7]</sup>

### 4.1.10 Related data structures

Both **stacks** and **queues** are often implemented using linked lists, and simply restrict the type of operations which are supported.

The **skip list** is a linked list augmented with layers of pointers for quickly jumping over large numbers of elements, and then descending to the next layer. This process continues down to the bottom layer, which is the actual list.

A **binary tree** can be seen as a type of linked list where the elements are themselves linked lists of the same nature. The result is that each node may include a reference to the first node of one or two other linked lists, which, together with their contents, form the subtrees below that node.

An **unrolled linked list** is a linked list in which each node contains an array of data values. This leads to improved cache performance, since more list elements are contiguous in memory, and reduced memory overhead, because less metadata needs to be stored for each element of the list.

A **hash table** may use linked lists to store the chains of items that hash to the same position in the hash table.

A **heap** shares some of the ordering properties of a linked list, but is almost always implemented using an array. Instead of references from node to node, the next and previous data indexes are calculated using the current data's index.

A **self-organizing list** rearranges its nodes based on some heuristic which reduces search times for data retrieval by keeping commonly accessed nodes at the head of the list.

### 4.1.11 Notes

[1] The amount of control data required for a dynamic array is usually of the form  $K + B * n$ , where  $K$  is a per-array constant,  $B$  is a per-dimension constant, and  $n$  is the number of dimensions.  $K$  and  $B$  are typically on the order of 10 bytes.

### 4.1.12 Footnotes

- [1] Skiena, Steven S. (2009). *The Algorithm Design Manual* (2nd ed.). Springer. p. 76. ISBN 9781848000704. We can do nothing without this list predecessor, and so must spend linear time searching for it on a singly-linked list.
- [2] Gerald Kruse. CS 240 Lecture Notes: Linked Lists Plus: Complexity Trade-offs. Juniata College. Spring 2008.
- [3] *Day 1 Keynote - Bjarne Stroustrup: C++11 Style* at *GoingNative 2012* on *channel9.msdn.com* from minute 45 or foil 44
- [4] *Number crunching: Why you should never, ever, EVER use linked-list in your code again* at *kjellkod.wordpress.com*
- [5] Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (1999). *Resizable Arrays in Optimal Time and Space* (Technical Report CS-99-09) (PDF). Department of Computer Science, University of Waterloo
- [6] Ford, William; Topp, William (2002). *Data Structures with C++ using STL* (Second ed.). Prentice-Hall. pp. 466–467. ISBN 0-13-085850-1.
- [7] Okasaki, Chris (1995). *Purely Functional Random-Access Lists* (PS). In *Functional Programming Languages and Computer Architecture* (ACM Press). pp. 86–95. Retrieved May 7, 2015.

### 4.1.13 References

- Juan, Angel (2006). “Ch20 –Data Structures; ID06 – PROGRAMMING with JAVA (slide part of the book 'Big Java', by CayS. Horstmann)” (PDF). p. 3.
- Black, Paul E. (2004-08-16). Pieterse, Vreda; Black, Paul E., eds. “linked list”. *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. Retrieved 2004-12-14.
- Antonakos, James L.; Mansfield, Kenneth C., Jr. (1999). *Practical Data Structures Using C/C++*. Prentice-Hall. pp. 165–190. ISBN 0-13-280843-9.
- Collins, William J. (2005) [2002]. *Data Structures and the Java Collections Framework*. New York: McGraw Hill. pp. 239–303. ISBN 0-07-282379-8.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2003). *Introduction to Algorithms*. MIT Press. pp. 205–213, 501–505. ISBN 0-262-03293-7.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). “10.2: Linked lists”. *Introduction to Algorithms* (2nd ed.). MIT Press. pp. 204–209. ISBN 0-262-03293-7.

- Green, Bert F., Jr. (1961). “Computer Languages for Symbol Manipulation”. *IRE Transactions on Human Factors in Electronics* (2): 3–8. doi:10.1109/THFE2.1961.4503292.
- McCarthy, John (1960). “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. *Communications of the ACM* 3 (4): 184. doi:10.1145/367177.367199.
- Knuth, Donald (1997). “2.2.3-2.2.5”. *Fundamental Algorithms* (3rd ed.). Addison-Wesley. pp. 254–298. ISBN 0-201-89683-4.
- Newell, Allen; Shaw, F. C. (1957). “Programming the Logic Theory Machine”. *Proceedings of the Western Joint Computer Conference*: 230–240.
- Parlante, Nick (2001). “Linked list basics” (PDF). Stanford University. Retrieved 2009-09-21.
- Sedgewick, Robert (1998). *Algorithms in C*. Addison Wesley. pp. 90–109. ISBN 0-201-31452-5.
- Shaffer, Clifford A. (1998). *A Practical Introduction to Data Structures and Algorithm Analysis*. New Jersey: Prentice Hall. pp. 77–102. ISBN 0-13-660911-2.
- Wilkes, Maurice Vincent (1964). “An Experiment with a Self-compiling Compiler for a Simple List-Processing Language”. *Annual Review in Automatic Programming* (Pergamon Press) 4 (1): 1. doi:10.1016/0066-4138(64)90013-8.
- Wilkes, Maurice Vincent (1964). “Lists and Why They are Useful”. *Proceeds of the ACM National Conference, Philadelphia 1964* (ACM) (P-64): F1–1.
- Shanmugasundaram, Kulesh (2005-04-04). “Linux Kernel Linked List Explained”. Retrieved 2009-09-21.

#### 4.1.14 External links

- Description from the Dictionary of Algorithms and Data Structures
- Introduction to Linked Lists, Stanford University Computer Science Library
- Linked List Problems, Stanford University Computer Science Library
- Open Data Structures - Chapter 3 - Linked Lists
- Patent for the idea of having nodes which are in several linked lists simultaneously (note that this technique was widely used for many decades before the patent was granted)

## 4.2 XOR linked list

An **XOR linked list** is a **data structure** used in computer programming. It takes advantage of the bitwise XOR operation to decrease storage requirements for **doubly linked lists**.

### 4.2.1 Description

An ordinary doubly linked list stores addresses of the previous and next list items in each list node, requiring two address fields:

... A B C D E ...  $\rightarrow$  next  $\rightarrow$  next  $\rightarrow$  next  $\rightarrow$   $\leftarrow$  prev  $\leftarrow$  prev  $\leftarrow$  prev  $\leftarrow$

An XOR linked list compresses the same information into *one* address field by storing the bitwise XOR (here denoted by  $\oplus$ ) of the address for *previous* and the address for *next* in one field:

... A B C D E ...  $\leftrightarrow$  A $\oplus$ C  $\leftrightarrow$  B $\oplus$ D  $\leftrightarrow$  C $\oplus$ E  $\leftrightarrow$

More formally:

$\text{link}(B) = \text{addr}(A) \oplus \text{addr}(C)$ ,  $\text{link}(C) = \text{addr}(B) \oplus \text{addr}(D)$ , ...

When you traverse the list from left to right: supposing you are at C, you can take the address of the previous item, B, and XOR it with the value in the link field (B $\oplus$ D). You will then have the address for D and you can continue traversing the list. The same pattern applies in the other direction.

i.e.  $\text{addr}(D) = \text{link}(C) \oplus \text{addr}(B)$  where  $\text{link}(C) = \text{addr}(B) \oplus \text{addr}(D)$  so  $\text{addr}(D) = \text{addr}(B) \oplus \text{addr}(D) \oplus \text{addr}(B)$   $\text{addr}(D) = \text{addr}(B) \oplus \text{addr}(B) \oplus \text{addr}(D)$  since  $X \oplus X = 0 \Rightarrow \text{addr}(D) = 0 \oplus \text{addr}(D)$  since  $X \oplus 0 = X \Rightarrow \text{addr}(D) = \text{addr}(D)$  The XOR operation cancels  $\text{addr}(B)$  appearing twice in the equation and all we are left with is the  $\text{addr}(D)$ .

To start traversing the list in either direction from some point, you need the address of two consecutive items, not just one. If the addresses of the two consecutive items are reversed, you will end up traversing the list in the opposite direction.

### Theory of operation

The key is the first operation, and the properties of XOR:

- $X \oplus X = 0$
- $X \oplus 0 = X$
- $X \oplus Y = Y \oplus X$
- $(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$

The R2 register always contains the XOR of the address of current item C with the address of the predecessor

item P:  $C \oplus P$ . The Link fields in the records contain the XOR of the left and right successor addresses, say  $L \oplus R$ . XOR of  $R2$  ( $C \oplus P$ ) with the current link field ( $L \oplus R$ ) yields  $C \oplus P \oplus L \oplus R$ .

- If the predecessor was  $L$ , the  $P (=L)$  and  $L$  cancel out leaving  $C \oplus R$ .
- If the predecessor had been  $R$ , the  $P (=R)$  and  $R$  cancel, leaving  $C \oplus L$ .

In each case, the result is the XOR of the current address with the next address. XOR of this with the current address in  $R1$  leaves the next address.  $R2$  is left with the requisite XOR pair of the (now) current address and the predecessor.

### 4.2.2 Features

- Given only one list item, one cannot immediately obtain the addresses of the other elements of the list.
- Two XOR operations suffice to do the traversal from one item to the next, the same instructions sufficing in both cases. Consider a list with items  $\{ \dots B \ C \ D \dots \}$  and with  $R1$  and  $R2$  being **registers** containing, respectively, the address of the current (say  $C$ ) list item and a work register containing the XOR of the current address with the previous address (say  $C \oplus D$ ). Cast as **System/360** instructions:

$X \ R2, \text{Link } R2 \leftarrow C \oplus D \oplus B \oplus D$  (i.e.  $B \oplus C$ , “Link” being the link field in the current record, containing  $B \oplus D$ )  $X \ R1, \ R2 \ R1 \leftarrow C \oplus B \oplus C$  (i.e.  $B$ , voilà: the next record)

- End of list is signified by imagining a list item at address zero placed adjacent to an end point, as in  $\{0 \ A \ B \ C \dots\}$ . The link field at  $A$  would be  $0 \oplus B$ . An additional instruction is needed in the above sequence after the two XOR operations to detect a zero result in developing the address of the current item,
- A list end point can be made reflective by making the link pointer be zero. A zero pointer is a *mirror*. (The XOR of the left and right neighbor addresses, being the same, is zero.)

### 4.2.3 Drawbacks

- General-purpose debugging tools cannot follow the XOR chain, making debugging more difficult; <sup>[1]</sup>
- The price for the decrease in memory usage is an increase in code complexity, making maintenance more expensive;
- Most **garbage collection** schemes do not work with data structures that do not contain literal pointers;

- XOR of pointers is not defined in some contexts (e.g., the **C** language), although many languages provide some kind of **type conversion** between pointers and integers;
- The pointers will be unreadable if one isn't traversing the list — for example, if the pointer to a list item was contained in another data structure;
- While traversing the list you need to remember the address of the previously accessed node in order to calculate the next node's address.
- XOR linked lists do not provide some of the important advantages of doubly-linked lists, such as the ability to delete a node from the list knowing only its address or the ability to insert a new node before or after an existing node when knowing only the address of the existing node.

Computer systems have increasingly cheap and plentiful memory, and storage overhead is not generally an overriding issue outside specialized **embedded systems**. Where it is still desirable to reduce the overhead of a linked list, **unrolling** provides a more practical approach (as well as other advantages, such as increasing cache performance and speeding **random access**).

### 4.2.4 Variations

The underlying principle of the XOR linked list can be applied to any reversible binary operation. Replacing XOR by addition or subtraction gives slightly different, but largely equivalent, formulations:

#### Addition linked list

$\dots A \ B \ C \ D \ E \dots \leftrightarrow A+C \leftrightarrow B+D \leftrightarrow C+E \leftrightarrow$

This kind of list has exactly the same properties as the XOR linked list, except that a zero link field is not a “mirror”. The address of the next node in the list is given by subtracting the previous node's address from the current node's link field.

#### Subtraction linked list

$\dots A \ B \ C \ D \ E \dots \leftrightarrow C-A \leftrightarrow D-B \leftrightarrow E-C \leftrightarrow$

This kind of list differs from the “traditional” XOR linked list in that the instruction sequences needed to traverse the list forwards is different from the sequence needed to traverse the list in reverse. The address of the next node, going forwards, is given by *adding* the link field to the previous node's address; the address of the preceding node is given by *subtracting* the link field from the next node's address.

The subtraction linked list is also special in that the entire list can be relocated in memory without needing any patching of pointer values, since adding a constant offset to each address in the list will not require any changes to the values stored in the link fields. (See also [Serialization](#).) This is an advantage over both XOR linked lists and traditional linked lists.

#### 4.2.5 See also

- [XOR swap algorithm](#)

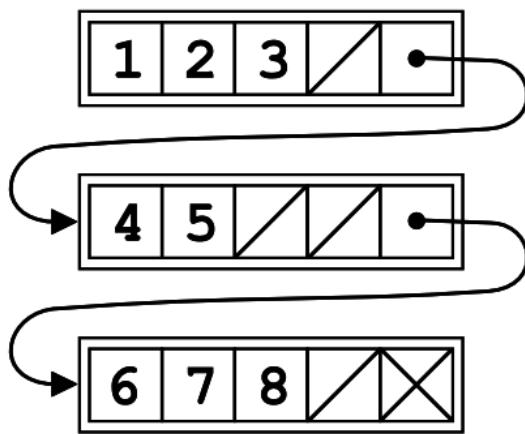
#### 4.2.6 References

- [1] <http://www.iecc.com/gclist/GC-faq.html#GC,%20C,%20and%20C++>

#### 4.2.7 External links

- [Example implementation in C++.](#)

### 4.3 Unrolled linked list



*Unrolled linked list*

*On this model, maximum number of elements is 4 for each node.*

In computer programming, an **unrolled linked list** is a variation on the [linked list](#) which stores multiple elements in each node. It can dramatically increase [cache](#) performance, while decreasing the memory overhead associated with storing list metadata such as references. It is related to the B-tree.

#### 4.3.1 Overview

A typical unrolled linked list node looks like this:

```
record node { node next // reference to next node in list
int numElements // number of elements in this node, up
to maxElements array elements // an array of numElements
elements, // with space allocated for maxElements
elements }
```

Each node holds up to a certain maximum number of elements, typically just large enough so that the node fills a single [cache line](#) or a small multiple thereof. A position in the list is indicated by both a reference to the node and a position in the elements array. It is also possible to include a *previous* pointer for an unrolled [doubly linked list](#).

To insert a new element, we simply find the node the element should be in and insert the element into the elements array, incrementing numElements. If the array is already full, we first insert a new node either preceding or following the current one and move half of the elements in the current node into it.

To remove an element, we simply find the node it is in and delete it from the elements array, decrementing numElements. If this reduces the node to less than half-full, then we move elements from the next node to fill it back up above half. If this leaves the next node less than half full, then we move all its remaining elements into the current node, then bypass and delete it.

#### 4.3.2 Performance

One of the primary benefits of unrolled linked lists is decreased storage requirements. All nodes (except at most one) are at least half-full. If many random inserts and deletes are done, the average node will be about three-quarters full, and if inserts and deletes are only done at the beginning and end, almost all nodes will be full. Assume that:

- $m$  = maxElements, the maximum number of elements in each elements array;
- $v$  = the overhead per node for references and element counts;
- $s$  = the size of a single element.

Then, the space used for  $n$  elements varies between  $(v/m + s)n$  and  $(2v/m + s)n$ . For comparison, ordinary linked lists require  $(v+s)n$  space, although  $v$  may be smaller, and [arrays](#), one of the most compact data structures, require  $sn$  space. Unrolled linked lists effectively spread the overhead  $v$  over a number of elements of the list. Thus, we see the most significant space gain when overhead is large, maxElements is large, or elements are small.

If the elements are particularly small, such as bits, the overhead can be as much as 64 times larger than the data on many machines. Moreover, many popular memory

allocators will keep a small amount of metadata for each node allocated, increasing the effective overhead  $v$ . Both of these make unrolled linked lists more attractive.

Because unrolled linked list nodes each store a count next to the *next* field, retrieving the  $k$ th element of an unrolled linked list (indexing) can be done in  $n/m + 1$  cache misses, up to a factor of  $m$  better than ordinary linked lists. Additionally, if the size of each element is small compared to the cache line size, the list can be traversed in order with fewer cache misses than ordinary linked lists. In either case, operation time still increases linearly with the size of the list.

### 4.3.3 See also

- CDR coding, another technique for decreasing overhead and improving cache locality in linked lists similar to unrolled linked lists.
- the **VList**, another array/singly-linked list hybrid designed for fast lookup
- the **skip list**, a similar variation on the linked list, offers fast lookup and hurts the advantages of linked lists (quick insert/deletion) less than an unrolled linked list
- the **B-tree** and **T-tree**, data structures that are similar to unrolled linked lists in the sense that each of them could be viewed as an “unrolled binary tree”
- **XOR linked list**, a doubly linked list that uses one XORed pointer per node instead of two ordinary pointers.
- **Hashed array tree**, where pointers to the chunks of data are held in a higher-level, separate array.

### 4.3.4 References

- Shao, Z.; Reppy, J. H.; Appel, A. W. (1994), “Unrolling lists”, *Conference record of the 1994 ACM Conference on Lisp and Functional Programming*: 185–191, doi:10.1145/182409.182453, ISBN 0897916433

### 4.3.5 External links

- Implementation written in C++
- Implementation written in C
- Another implementation written in Java
- Open Data Structures—Section 3.3—SELList: A Space-Efficient Linked List

## 4.4 VList

Not to be confused with **Vlist**, a municipality in the Netherlands.

In computer science, the **VList** is a persistent data structure designed by Phil Bagwell in 2002 that combines the fast indexing of arrays with the easy extension of cons-based (or singly linked) linked lists.<sup>[1]</sup>

Like arrays, VLists have constant-time lookup on average and are highly compact, requiring only  $O(\log n)$  storage for pointers, allowing them to take advantage of locality of reference. Like singly linked or cons-based lists, they are persistent, and elements can be added to or removed from the front in constant time. Length can also be found in  $O(\log n)$  time.

### 4.4.1 Operations

The primary operations of a VList are:

- Locate the  $k$ th element ( $O(1)$  average,  $O(\log n)$  worst-case)
- Add an element to the front of the VList ( $O(1)$  average, with an occasional allocation)
- Obtain a new array beginning at the second element of an old array ( $O(1)$ )
- Compute the length of the list ( $O(\log n)$ )

### 4.4.2 Advantages and Disadvantages

The primary advantage VLists have over arrays is that different updated versions of the VList automatically share structure. Because VLists are immutable, they are most useful in functional programming languages, where their efficiency allows a purely functional implementation of data structures traditionally thought to require mutable arrays, such as hash tables.

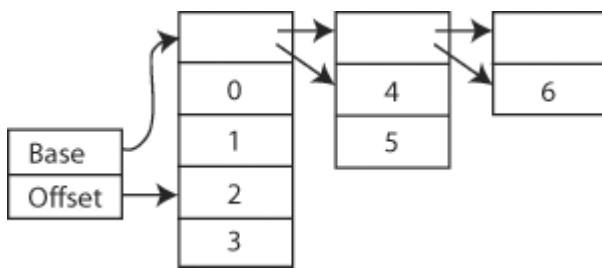
However, VLists also have a number of disadvantages over their competitors:

- While immutability is a benefit, it is also a drawback, making it inefficient to modify elements in the middle of the array.
- Access near the end of the list can be as expensive as  $O(\log n)$ ; it is only constant on average over all elements. This is still, however, much better than performing the same operation on cons-based lists.
- Wasted space in the first block is proportional to  $n$ . This is similar to linked lists, but there are data structures with less overhead. When used as a fully persistent data structure, the overhead may be consid-

erably higher and this data structure may not be appropriate.

### 4.4.3 Structure

The underlying structure of a VList can be seen as a singly linked list of arrays whose sizes decrease geometrically; in its simplest form, the first contains the first half of the elements in the list, the next the first half of the remainder, and so on. Each of these blocks stores some information such as its size and a pointer to the next.



An array-list. The reference shown refers to the VList (2,3,4,5,6).

The average constant-time indexing operation comes directly from this structure; given a random valid index, we simply observe the size of the blocks and follow pointers until we reach the one it should be in. The chance is 1/2 that it falls in the first block and we need not follow any pointers; the chance is 1/4 we have to follow only one, and so on, so that the expected number of pointers we have to follow is:

$$\sum_{i=1}^{\lceil \log_2 n \rceil} \frac{i-1}{2^i} < \sum_{i=1}^{\infty} \frac{i-1}{2^i} = 1.$$

Any particular reference to a VList is actually a `<base, offset>` pair indicating the position of its first element in the data structure described above. The `base` part indicates which of the arrays its first element falls in, while the `offset` part indicates its index in that array. This makes it easy to “remove” an element from the front of the list; we simply increase the offset, or increase the base and set the offset to zero if the offset goes out of range. If a particular reference is the last to leave a block, the block will be **garbage-collected** if such facilities are available, or otherwise must be freed explicitly.

Because the lists are constructed incrementally, the first array in the array list may not contain twice as many values as the next one, although the rest do; this does not significantly impact indexing performance. We nevertheless allocate this much space for the first array, so that if we add more elements to the front of the list in the future we can simply add them to this list and update the size. If the array fills up, we create a new array, twice as large again as this one, and link it to the old first array.

The trickier case, however, is adding a new item to the front of a list, call it A, which starts somewhere in the

middle of the array-list data structure. This is the operation that allows VLists to be persistent. To accomplish this, we create a new array, and we link it to the array containing the first element of A. The new array must also store the offset of the first element of A in that array. Then, we can proceed to add any number of items we like to our new array, and any references into this new array will point to VLists which share a tail of values with the old array. Note that with this operation it is possible to create VLists which degenerate into simple linked lists, thus obliterating the performance claims made at the beginning of this article.

### 4.4.4 Variants

VList may be modified to support the implementation of a **growable array**. In the application of a growable array, **immutability** is no longer required. Instead of growing at the beginning of the list, the ordering interpretation is reversed to allow growing at the end of the array.

### 4.4.5 See also

- Purely functional
- Unrolled linked list

### 4.4.6 References

- [1] Bagwell, Phil (2002), *Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays* (PDF), EPFL

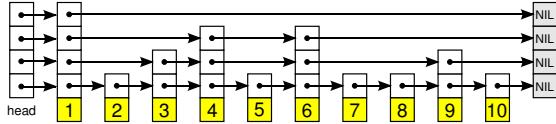
### 4.4.7 External links

- C# implementation of VLists
- Scheme implementation of VLists and VList-based hash lists for GNU Guile
- Scheme (Typed Racket) implementation of VLists for Racket

## 4.5 Skip list

In computer science, a **skip list** is a data structure that allows fast search within an ordered sequence of elements. Fast search is made possible by maintaining a **linked hierarchy** of subsequences, each skipping over fewer elements. Searching starts in the sparsest subsequence until two consecutive elements have been found, one smaller and one larger than the element searched for. Via the linked hierarchy these two elements link to elements of the next sparsest subsequence where searching is continued until finally we are searching in the full sequence. The

elements that are skipped over may be chosen probabilistically<sup>[2]</sup> or deterministically,<sup>[3]</sup> with the former being more common.



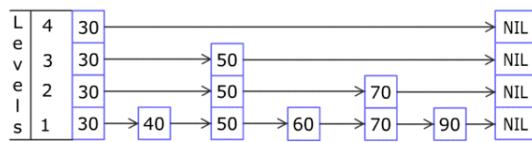
A schematic picture of the skip list data structure. Each box with an arrow represents a pointer and a row is a *linked list* giving a sparse subsequence; the numbered boxes at the bottom represent the ordered data sequence. Searching proceeds downwards from the sparsest subsequence at the top until consecutive elements bracketing the search element are found.

### 4.5.1 Description

A skip list is built in layers. The bottom layer is an ordinary ordered *linked list*. Each higher layer acts as an “express lane” for the lists below, where an element in layer  $i$  appears in layer  $i+1$  with some fixed probability  $p$  (two commonly used values for  $p$  are  $1/2$  or  $1/4$ ). On average, each element appears in  $1/(1-p)$  lists, and the tallest element (usually a special head element at the front of the skip list) in  $\log_{1/p} n$  lists.

A search for a target element begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, it has been found. If the current element is greater than the target, or the search reaches the end of the linked list, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower list. The expected number of steps in each linked list is at most  $1/p$ , which can be seen by tracing the search path backwards from the target until reaching an element that appears in the next higher list or reaching the beginning of the current list. Therefore, the total *expected* cost of a search is  $(\log_{1/p} n)/p$ , which is  $\mathcal{O}(\log n)$  when  $p$  is a constant. By choosing different values of  $p$ , it is possible to trade search costs against storage costs.

### Implementation details



Inserting elements to skip list

The elements used for a skip list can contain more than one pointer since they can participate in more than one

list.

Insertions and deletions are implemented much like the corresponding linked-list operations, except that “tall” elements must be inserted into or deleted from more than one linked list.

$\mathcal{O}(n)$  operations, which force us to visit every node in ascending order (such as printing the entire list), provide the opportunity to perform a behind-the-scenes derandomization of the level structure of the skip-list in an optimal way, bringing the skip list to  $\mathcal{O}(\log n)$  search time. (Choose the level of the  $i$ 'th finite node to be 1 plus the number of times we can repeatedly divide  $i$  by 2 before it becomes odd. Also,  $i=0$  for the negative infinity header as we have the usual special case of choosing the highest possible level for negative and/or positive infinite nodes.) However this also allows someone to know where all of the higher-than-level 1 nodes are and delete them.

Alternatively, we could make the level structure quasi-random in the following way:

```
make all nodes level 1 j ← 1 while the number of nodes
at level j > 1 do for each i'th node at level j do if i is odd
if i is not the last node at level j randomly choose whether
to promote it to level j+1 else do not promote end if else
if i is even and node i-1 was not promoted promote it to
level j+1 end if repeat j ← j + 1 repeat
```

Like the derandomized version, quasi-randomization is only done when there is some other reason to be running a  $\mathcal{O}(n)$  operation (which visits every node).

The advantage of this quasi-randomness is that it doesn't give away nearly as much level-structure related information to an *adversarial user* as the de-randomized one. This is desirable because an adversarial user who is able to tell which nodes are not at the lowest level can pessimize performance by simply deleting higher-level nodes. The search performance is still guaranteed to be logarithmic.

It would be tempting to make the following “optimization”: In the part which says “Next, for each  $i$ 'th...”, forget about doing a coin-flip for each even-odd pair. Just flip a coin once to decide whether to promote only the even ones or only the odd ones. Instead of  $\mathcal{O}(n \log n)$  coin flips, there would only be  $\mathcal{O}(\log n)$  of them. Unfortunately, this gives the adversarial user a 50/50 chance of being correct upon guessing that all of the even numbered nodes (among the ones at level 1 or higher) are higher than level one. This is despite the property that he has a very low probability of guessing that a particular node is at level  $N$  for some integer  $N$ .

A skip list does not provide the same absolute worst-case performance guarantees as more traditional *balanced tree* data structures, because it is always possible (though with very low probability) that the coin-flips used to build the skip list will produce a badly balanced structure. However, they work well in practice, and the randomized balancing scheme has been argued to be easier to implement than the deterministic balancing schemes used in

balanced binary search trees. Skip lists are also useful in **parallel computing**, where insertions can be done in different parts of the skip list in parallel without any global rebalancing of the data structure. Such parallelism can be especially advantageous for resource discovery in an ad-hoc **wireless network** because a randomized skip list can be made robust to the loss of any single node.<sup>[4]</sup>

There has been some evidence that skip lists have worse real-world performance and space requirements than **B trees** due to **memory locality** and other issues.<sup>[5]</sup>

### Indexable skiplist

As described above, a skiplist is capable of fast  $\mathcal{O}(\log n)$  insertion and removal of values from a sorted sequence, but it has only slow  $\mathcal{O}(n)$  lookups of values at a given position in the sequence (i.e. return the 500th value); however, with a minor modification the speed of **random access** indexed lookups can be improved to  $\mathcal{O}(\log n)$ .

For every link, also store the width of the link. The width is defined as the number of bottom layer links being traversed by each of the higher layer “express lane” links.

For example, here are the widths of the links in the example at the top of the page:

```
1 10 o---> o-----
-----> o Top level 1 3 2 5 o---> o-----> o-----
o-----> o Level 3 1 2 1 2 5 o---> o--->
-----> o---> o-----> o Level
2 1 1 1 1 1 1 1 1 1 1 o---> o---> o---> o---> o--->
---> o---> o---> o---> o---> o Bottom level Head
1st 2nd 3rd 4th 5th 6th 7th 8th 9th 10th NIL Node Node
Node Node Node Node Node Node Node Node Node
```

Notice that the width of a higher level link is the sum of the component links below it (i.e. the width 10 link spans the links of widths 3, 2 and 5 immediately below it). Consequently, the sum of all widths is the same on every level ( $10 + 1 = 1 + 3 + 2 + 5 = 1 + 2 + 1 + 2 + 5$ ).

To index the skiplist and find the  $i$ 'th value, traverse the skiplist while counting down the widths of each traversed link. Descend a level whenever the upcoming width would be too large.

For example, to find the node in the fifth position (Node 5), traverse a link of width 1 at the top level. Now four more steps are needed but the next width on this level is ten which is too large, so drop one level. Traverse one link of width 3. Since another step of width 2 would be too far, drop down to the bottom level. Now traverse the final link of width 1 to reach the target running total of 5 ( $1+3+1$ ).

```
function lookupByPositionIndex(i) node ← head i ← i
+ 1 # don't count the head as a step
for level from top
to bottom do while i ≥ node.width[level] do # if next
step is not too far i ← i - node.width[level] # subtract the
current width node ← node.next[level] # traverse forward
```

*at the current level repeat repeat return node.value end function*

This method of implementing indexing is detailed in Section 3.4 **Linear List Operations** in “A skip list cookbook” by William Pugh.

### 4.5.2 History

Skip lists were first described in 1989 by William Pugh.<sup>[6]</sup>

To quote the author:

*Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.*

### 4.5.3 Usages

List of applications and frameworks that use skip lists:

- **Cyrus IMAP server** offers a “skiplist” backend DB implementation ([source file](#))
- **Lucene** uses skip lists to search delta-encoded posting lists in logarithmic time.
- **QMap** (up to Qt 4) template class of **Qt** that provides a dictionary.
- **Redis**, an ANSI-C open-source persistent key/value store for Posix systems, uses skip lists in its implementation of ordered sets.<sup>[7]</sup>
- **nessDB**, a very fast key-value embedded Database Storage Engine (Using log-structured-merge (LSM) trees), uses skip lists for its memtable.
- **skipdb** is an open-source database format using ordered key/value pairs.
- **ConcurrentSkipListSet** and **ConcurrentSkipListMap** in the Java 1.6 API.
- **leveldb**, a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values
- **Skip lists** are used for efficient statistical computations of running medians (also known as moving medians).

Skip lists are also used in distributed applications (where the nodes represent physical computers, and pointers represent network connections) and for implementing highly scalable concurrent **priority queues** with less lock contention,<sup>[8]</sup> or even without locking,<sup>[9][10][11]</sup> as well as

lockless concurrent dictionaries.<sup>[12]</sup> There are also several US patents for using skip lists to implement (lockless) priority queues and concurrent dictionaries.<sup>[13]</sup>

#### 4.5.4 See also

- Bloom filter
- Skip graph

#### 4.5.5 References

- [1] <http://www.cs.uwaterloo.ca/research/tr/1993/28/root2side.pdf>
- [2] Pugh, W. (1990). “Skip lists: A probabilistic alternative to balanced trees” (PDF). *Communications of the ACM* **33** (6): 668. doi:10.1145/78973.78977.
- [3] Munro, J. Ian; Papadakis, Thomas; Sedgewick, Robert (1992). “Deterministic skip lists”. *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA '92)*. Orlando, Florida, USA: Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. pp. 367–375. alternative link
- [4] Shah, Gauri Ph.D.; James Aspnes (December 2003). “Distributed Data Structures for Peer-to-Peer Systems” (PDF). Retrieved 2008-09-23.
- [5] [http://resnet.uoregon.edu/~{}gurney\\_j/jmpc/skiplist.html](http://resnet.uoregon.edu/~{}gurney_j/jmpc/skiplist.html)
- [6] William Pugh (April 1989). “Concurrent Maintenance of Skip Lists”, Tech. Report CS-TR-2222, Dept. of Computer Science, U. Maryland.
- [7] “Redis ordered set implementation”.
- [8] Shavit, N.; Lotan, I. (2000). “SkipList-based concurrent priority queues”. *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000* (PDF). p. 263. doi:10.1109/IPDPS.2000.845994. ISBN 0-7695-0574-0.
- [9] Sundell, H.; Tsigas, P. (2003). “Fast and lock-free concurrent priority queues for multi-thread systems”. *Proceedings International Parallel and Distributed Processing Symposium*. p. 11. doi:10.1109/IPDPS.2003.1213189. ISBN 0-7695-1926-1.
- [10] Fomitchev, M.; Ruppert, E. (2004). “Lock-free linked lists and skip lists”. *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing - PODC '04*. p. 50. doi:10.1145/1011767.1011776. ISBN 1581138024.
- [11] Bajpai, R.; Dhara, K. K.; Krishnaswamy, V. (2008). “QPID: A Distributed Priority Queue with Item Locality”. *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. p. 215. doi:10.1109/ISPA.2008.90. ISBN 978-0-7695-3471-8.
- [12] Sundell, H. K.; Tsigas, P. (2004). “Scalable and lock-free concurrent dictionaries”. *Proceedings of the 2004 ACM symposium on Applied computing - SAC '04* (PDF). p. 1438. doi:10.1145/967900.968188. ISBN 1581138121.
- [13] US patent 7937378

#### 4.5.6 External links

- “Skip list” entry in the Dictionary of Algorithms and Data Structures
- Skip Lists: A Linked List with Self-Balancing BST-Like Properties on MSDN in C# 2.0
- SkipDB, a BerkeleyDB-style database implemented using skip lists.
- Skip Lists lecture (MIT OpenCourseWare: Introduction to Algorithms)
- Open Data Structures - Chapter 4 - Skiplists
- Skip trees, an alternative data structure to skip lists in a concurrent approach
- Skip tree graphs, a distributed version of skip trees
- More on skip tree graphs, a distributed version of skip trees

#### Demo applets

- Skip List Applet by Kubo Kovac
- Thomas Wenger’s demo applet on skiplists

#### Implementations

- A generic Skip List in C++ by Antonio Gulli
- Algorithm::SkipList, implementation in Perl on CPAN
- John Shipman’s implementation in Python
- Raymond Hettinger’s implementation in Python
- A Lua port of John Shipman’s Python version
- Java Implementation with index based access
- ConcurrentSkipListSet documentation for Java 6 (and sourcecode)

## 4.6 Self-organizing list

A **self-organizing list** is a list that reorders its elements based on some **self-organizing heuristic** to improve **average access time**. The aim of a self-organizing list is to improve efficiency of linear search by moving more frequently accessed items towards the head of the list. A self-organizing list achieves near constant time for element access in the best case. A self-organizing list uses a reorganizing algorithm to adapt to various query distributions at runtime.

### 4.6.1 History

The concept of self-organizing lists has its roots in the idea of activity organization [1] of records in files stored on disks or tapes. One frequently cited discussion of self-organizing files and lists is Knuth [2]. John McCabe has the first algorithmic complexity analyses of the Move-to-Front (MTF) strategy where an item is moved to the front of the list after it is accessed. [3] He analyses the average time needed for randomly ordered list to get in optimal order. The optimal ordering of a list is the one in which items are ordered in the list by the probability with which they will be needed, with the most accessed item first. The optimal ordering may not be known in advance, and may also change over time. McCabe introduced the transposition strategy in which an accessed item is exchanged with the item in front of it in the list. He made the conjecture that transposition worked at least as well in the average case as MTF in approaching the optimal ordering of records in the limit. This conjecture was later proved by Rivest. [4] McCabe also noted that with either the transposition or MTF heuristic, the optimal ordering of records would be approached even if the heuristic was only applied every  $N$ th access, and that a value of  $N$  might be chosen that would reflect the relative cost of relocating records with the value of approaching the optimal ordering more quickly. Further improvements were made, and algorithms suggested by researchers including: Rivest, Tenenbaum and Nemes, Knuth, and Bentley and McGeoch (e.g. Worst-case analyses for self-organizing sequential search heuristics).

### 4.6.2 Introduction

The simplest implementation of a self-organizing list is as a **linked list** and thus while being efficient in random node inserting and memory allocation, suffers from inefficient accesses to random nodes. A self-organizing list reduces the inefficiency by dynamically rearranging the nodes in the list based on access frequency.

### Inefficiency of linked list traversals

If a particular node is to be searched for in the list, each node in the list must be sequentially compared till the desired node is reached. In a linked list, retrieving the  $n$ th element is an  $O(n)$  operation. This is highly inefficient when compared to an array for example, where accessing the  $n^{\text{th}}$  element is an  $O(1)$  operation.

### Efficiency of self-organizing lists

A self organizing list rearranges the nodes keeping the most frequently accessed ones at the head of the list. Generally, in a particular query, the chances of accessing a node which has been accessed many times before are higher than the chances of accessing a node which historically has not been so frequently accessed. As a result, keeping the commonly accessed nodes at the head of the list results in reducing the number of comparisons required in an average case to reach the desired node. This leads to better efficiency and generally reduced query times.

### 4.6.3 Implementation of a self-organizing list

The implementation and methods of a self-organizing list are identical to the those for a standard **linked list**. The linked list and the self-organizing list differ only in terms of the organization of the nodes; the interface remains the same.

### 4.6.4 Analysis of Running Times for Access/ Search in a List

#### Average Case

It can be shown that in the average case, the time required to a search on a self-organizing list of size  $n$  is

$$T_{avg} = 1 * p(1) + 2 * p(2) + 3 * p(3) + \dots + n * p(n).$$

where  $p(i)$  is the probability of accessing the  $i$ th element in the list, thus also called the access probability. If the access probability of each element is the same (i.e.  $p(1) = p(2) = p(3) = \dots = p(n) = 1/n$ ) then the ordering of the elements is irrelevant and the average time complexity is given by

$$T(n) = 1/n + 2/n + 3/n + \dots + n/n = (1+2+3+\dots+n)/n = (n+1)/2$$

and  $T(n)$  does not depend on the individual access probabilities of the elements in the list in this case. However in the case of searches on lists with non uniform record

access probabilities (i.e. those lists in which the probability of accessing one element is different from another), the average time complexity can be reduced drastically by proper positioning of the elements contained in the list. This is done by pairing smaller  $i$  with larger access probabilities so as to reduce the overall average time complexity.

This may be demonstrated as follows:

Given List: A(0.1), B(0.1), C(0.3), D(0.1), E(0.4)

Without rearranging, average search time required is:

$$T(n) = 1*0.1 + 2*0.1 + 3*0.3 + 4*0.1 + 5*0.4 = 3.6$$

Now suppose the nodes are rearranged so that those nodes with highest probability of access are placed closest to the front so that the rearranged list is now:

E(0.4), C(0.3), D(0.1), A(0.1), B(0.1)

Here, average search time is:

$$T(n) = 1*0.4 + 2*0.3 + 3*0.1 + 4*0.1 + 5*0.1 = 2.2$$

Thus the average time required for searching in an organized list is (in this case) around 40% less than the time required to search a randomly arranged list.

This is the concept of the self-organized list in that the average speed of data retrieval is increased by rearranging the nodes according to access frequency.

### Worst Case

In the worst case, the element to be located is at the very end of the list be it a normal list or a self-organized one and thus  $n$  comparisons must be made to reach it. Therefore the worst case running time of a linear search on the list is  $O(n)$  independent of the type of list used. Note that the expression for the average search time in the previous section is a probabilistic one. Keeping the commonly accessed elements at the head of the list simply reduces the probability of the worst case occurring but does not eliminate it completely. Even in a self-organizing list, if a lowest access probability element (obviously located at the end of the list) is to be accessed, the entire list must be traversed completely to retrieve it. This is the worst case search.

### Best Case

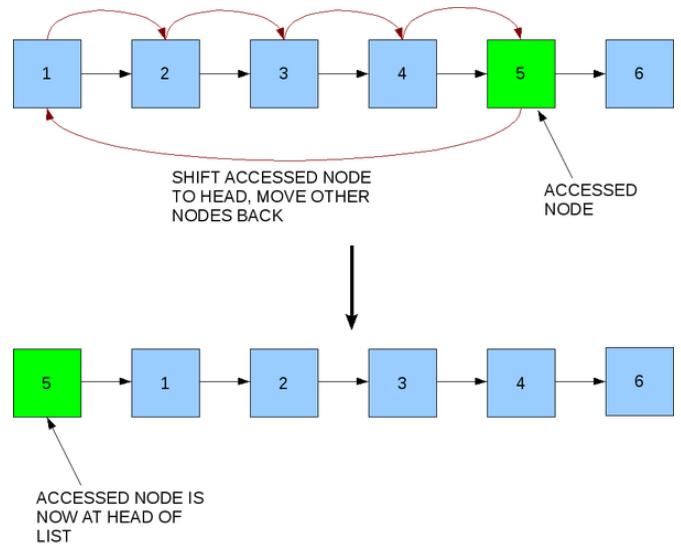
In the best case, the node to be searched is one which has been commonly accessed and has thus been identified by the list and kept at the head. This will result in a near constant time operation. In big-oh notation, in the best case, accessing an element is an  $O(1)$  operation.

### 4.6.5 Techniques for Rearranging Nodes

While ordering the elements in the list, the access probabilities of the elements are not generally known in advance. This has led to the development of various heuristics to approximate optimal behavior. The basic heuristics used to reorder the elements in the list are:

#### Move to Front Method (MTF)

This technique moves the element which is accessed to the head of the list. This has the advantage of being easily implemented and requiring no extra memory. This heuristic also adapts quickly to rapid changes in the query distribution. On the other hand, this method may prioritize infrequently accessed nodes—for example, if an uncommon node is accessed even once, it is moved to the head of the list and given maximum priority even if it is not going to be accessed frequently in the future. These 'over rewarded' nodes destroy the optimal ordering of the list and lead to slower access times for commonly accessed elements. Another disadvantage is that this method may become too flexible leading to access patterns that change too rapidly. This means that due to the very short memories of access patterns even an optimal arrangement of the list can be disturbed immediately by accessing an infrequent node in the list.

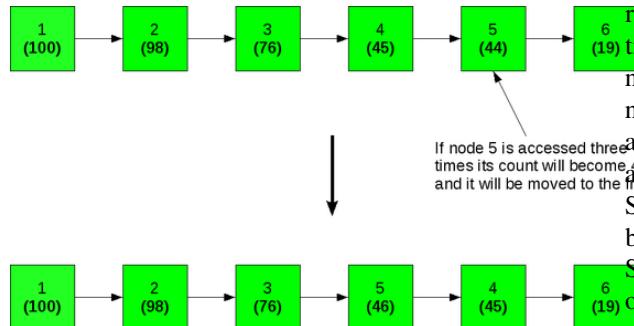


If the 5th node is selected, it is moved to the front  
At the  $t$ -th item selection: if item  $i$  is selected: move item  $i$  to head of the list

#### Count Method

In this technique, the number of times each node was searched for is counted i.e. every node keeps a separate counter variable which is incremented every time it is called. The nodes are then rearranged according to decreasing count. Thus, the nodes of highest count i.e. most frequently accessed are kept at the head of the list. The

primary advantage of this technique is that it generally is more realistic in representing the actual access pattern. However, there is an added memory requirement, that of maintaining a counter variable for each node in the list. Also, this technique does not adapt quickly to rapid changes in the access patterns. For example: if the count of the head element say A is 100 and for any node after it say B is 40, then even if B becomes the new most commonly accessed element, it must still be accessed at least  $(100 - 40 = 60)$  times before it can become the head element and thus make the list ordering optimal.

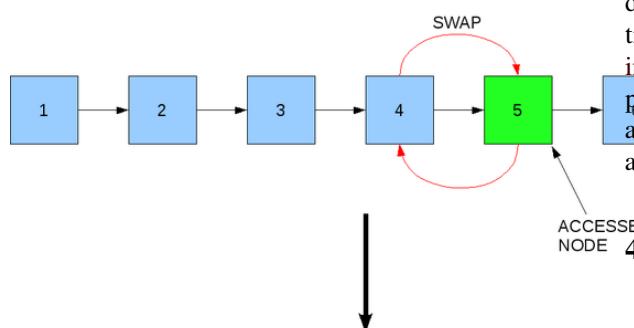


If the 5th node in the list is searched for twice, it will be swapped with the 4th

**init:** count(i) = 0 for each item i At t-th item selection: **if** item i is searched: count(i) = count(i) + 1 rearrange items based on count

### Transpose Method

This technique involves swapping an accessed node with its predecessor. Therefore, if any node is accessed, it is swapped with the node in front unless it is the head node, thereby increasing its priority. This algorithm is again easy to implement and space efficient and is more likely to keep frequently accessed nodes at the front of the list. However, the transpose method is more cautious. i.e. it will take many accesses to move the element to the head of the list. This method also does not allow for rapid response to changes in the query distributions on the nodes in the list.



If the 5th node in the list is selected, it will be swapped with the 4th

At the t-th item selection: **if** item i is selected: **if** i is not the head of list: swap item i with item (i - 1)

### Other Methods

Research has been focused on fusing the above algorithms to achieve better efficiency.<sup>[5]</sup> Bitner's Algorithm uses MTF initially and then uses transpose method for finer rearrangements. Some algorithms are randomized and try to prevent the over-rewarding of infrequently accessed nodes that may occur in the MTF algorithm. Other techniques involve reorganizing the nodes based on the above algorithms after every n accesses on the list as a whole or after n accesses in a row on a particular node and so on. Some algorithms rearrange the nodes which are accessed based on their proximity to the head node, for example: Swap-With-Parent or Move-To-Parent algorithms. Another class of algorithms are used when the search pattern exhibits a property called locality of reference whereby in a given interval of time, only a smaller subset of the list is probabilistically most likely to be accessed. This is also referred to as dependent access where the probability of the access of a particular element depends on the probability of access of its neighboring elements. Such models are common in real world applications such as database or file systems and memory management and caching. A common framework for algorithms dealing with such dependent environments is to rearrange the list not only based on the record accessed but also on the records near it. This effectively involves reorganizing a sublist of the list to which the record belongs.

### 4.6.6 Applications of self-organizing lists

Language translators like compilers and interpreters use self-organizing lists to maintain **symbol tables** during compilation or interpretation of program source code. Currently research is underway to incorporate the self-organizing list data structure in **embedded systems** to reduce bus transition activity which leads to power dissipation in those circuits. These lists are also used in **artificial intelligence** and **neural networks** as well as self-adjusting programs. The algorithms used in self-organizing lists are also used as **caching algorithms** as in the case of LFU algorithm.

### 4.6.7 References

- [1] Hayes, R.M. (1963), *Information Storage and Retrieval: Tools, Elements, Theories*, New York: Wiley
- [2] Knuth, Donald (1998), *Sorting and Searching*, The Art of Computer Programming, Volume 3 (Second ed.), Addison-Wesley, p. 402, ISBN 0-201-89685-0

- [3] McCabe, John (1965), “On Serial Files with Relocatable Records”, *Operations Research (INFORMS)* **13** (4): 609–618, doi:10.1287/opre.13.4.609
- [4] Rivest, Ronald (1976), “On self-organizing sequential search heuristics”, *Communications of the ACM* **19** (2): 63–67, doi:10.1145/359997.360000
- [5] <http://www.springerlink.com/content/978-3-540-34597-8/#section=508698&page=1&locus=3> Lists on Lists: A Framework for Self Organizing-Lists in Environments with Locality of Reference
  - Vlajic, N (2003), *Self-Organizing Lists* (PDF)
  - *Self Organization* (PDF), 2004
  - NIST DADS entry
  - A Drozdek, Data Structures and Algorithms in Java Third edition
  - Amer, Abdelrehman; B. John Oommen (2006), Lists on Lists: A Framework for Self Organizing-Lists in Environments with Locality of Reference *Lists on Lists: A Framework for Self-organizing Lists in Environments with Locality of Reference* (PDF)

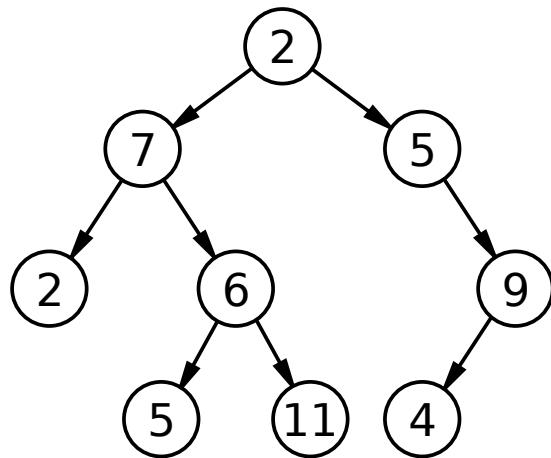
# Chapter 5

## Binary trees

### 5.1 Binary tree

Not to be confused with **B-tree**.

In computer science, a **binary tree** is a tree data struc-



A labeled binary tree of size 9 and height 3, with a root node whose value is 2. The above tree is unbalanced and not sorted.

ture in which each node has at most two **children**, which are referred to as the *left* child and the *right* child. A recursive definition using just **set theory** notions is that a (non-empty) binary tree is a **triple**  $(L, S, R)$ , where  $L$  and  $R$  are binary trees or the empty set and  $S$  is a **singleton set**.<sup>[1]</sup> Some authors allow the binary tree to be the empty set as well.<sup>[2]</sup>

From a **graph theory** perspective, binary (and K-ary) trees as defined here are actually **arborescences**.<sup>[3]</sup> A binary tree may thus be also called a **bifurcating arborescence**<sup>[3]</sup>—a term which actually appears in some very old programming books,<sup>[4]</sup> before the modern computer science terminology prevailed. It is also possible to interpret a binary tree as an **undirected**, rather than a **directed graph**, in which case a binary tree is an **ordered, rooted tree**.<sup>[5]</sup> Some authors use **rooted binary tree** instead of **binary tree** to emphasize the fact that the tree is rooted, but as defined above, a binary tree is always rooted.<sup>[6]</sup> A binary tree is a special case of an ordered **K-ary tree**, where  $k$  is 2. There is however a subtle difference between the binary tree data structure as defined here and

the notions from graph theory or as K-ary tree is usually defined. As defined here, a binary tree node having a left child but no right child is not the same as a node having a right child but no left child, whereas an ordered/plane tree (or arborescence) from graph theory cannot tell these cases apart, and neither does  $k$ -ary as usually understood as using a list of children.<sup>[7]</sup> An actual generalization of binary tree would have to discern, for example, a case like having a first and third, but no second child; the **trie** data structure is actually the more appropriate generalization in this respect.<sup>[8]</sup>

In computing, binary trees are seldom used solely for their structure. Much more typical is to define a labeling function on the nodes, which associates some value to each node.<sup>[9]</sup> Binary trees labelled this way are used to implement **binary search trees** and **binary heaps**, and are used for efficient **searching** and **sorting**. The designation of non-root nodes as left or right child even when there is only one child present matters in some of these applications, in particular it is significant in binary search trees.<sup>[10]</sup> In mathematics, what is termed **binary tree** can vary significantly from author to author. Some use the definition commonly used in computer science,<sup>[11]</sup> but others define it as every non-leaf having exactly two children and don't necessarily order (as left/right) the children either.<sup>[12]</sup>

#### 5.1.1 Definitions

##### Recursive definition

Another way of defining a **full** binary tree is a **recursive definition**. A full binary tree is either:<sup>[13]</sup>

- A single vertex.
- A graph formed by taking two (full) binary trees, adding a vertex, and adding an edge directed from the new vertex to the root of each binary tree.

This also does not establish the order of children, but does fix a specific root node.

To actually define a binary tree in general, we must allow for the possibility that only one of children may be

empty. An artifact, which in some textbooks is called an *extended binary tree* is needed for that purpose. An extended binary tree is thus recursively defined as:<sup>[13]</sup>

- the empty set is an extended binary tree
- if  $T_1$  and  $T_2$  are extended binary trees, then denote by  $T_1 \bullet T_2$  the extended binary tree obtained by adding a root  $r$  connected to the left to  $T_1$  and to the right to  $T_2$  by adding edges when these sub-trees are non-empty.

Another way of imagining this construction (and understanding the terminology) is to consider instead of the empty set a different type of node—for instance square nodes if the regular ones are circles.<sup>[14]</sup>

### Using graph theory concepts

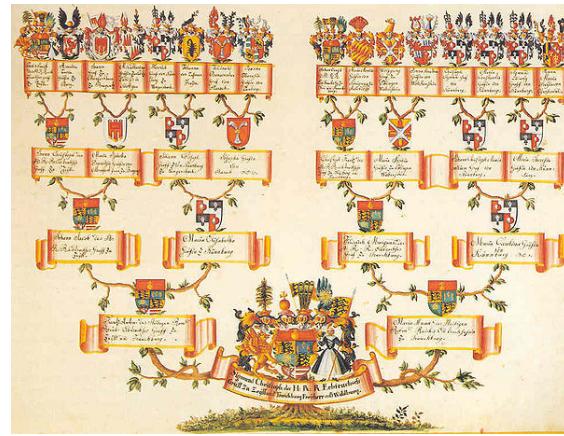
A binary tree is a **rooted tree** that is also an **ordered tree** (a.k.a. **plane tree**) in which every node has at most two children. A rooted tree naturally imparts a notion of levels (distance from the root), thus for every node a notion of children may be defined as the nodes connected to it at a level below. Ordering of these children (e.g., by drawing them on a plane) makes possible to distinguish left child from right child.<sup>[15]</sup> But this still doesn't distinguish between a node with left but not a right child from a one with right but no left child.

The necessary distinction can be made by first partitioning the edges, i.e., defining the binary tree as triplet  $(V, E_1, E_2)$ , where  $(V, E_1 \cup E_2)$  is a rooted tree (equivalently arborescence) and  $E_1 \cap E_2$  is empty, and also requiring that for all  $j \in \{1, 2\}$  every node has at most one  $E_j$  child.<sup>[16]</sup> A more informal way of making the distinction is to say, quoting the *Encyclopedia of Mathematics*, that “every node has a left child, a right child, neither, or both” and to specify that these “are all different” binary trees.<sup>[11]</sup>

#### 5.1.2 Types of binary trees

Tree terminology is not well-standardized and so varies in the literature.

- A **rooted** binary tree has a **root node** and every node has at most two children.
- A **full** binary tree (sometimes referred to as a **proper** or **plane** binary tree)<sup>[17][18]</sup> is a tree in which every node in the tree has either 0 or 2 children.
- A **perfect** binary tree is a binary tree in which all leaves have the same *depth* or same *level*.<sup>[19]</sup> (This is ambiguously also called a **complete** or **full** binary tree.) An example of a perfect binary tree is



An ancestry chart which maps to a perfect depth-4 binary tree.

the ancestry chart of a person to a given depth, as each person has exactly two biological parents (one mother and one father).

- In a **complete** binary tree every level, *except possibly the last*, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and  $2^h$  nodes at the last level  $h$ .<sup>[20]</sup> A binary tree is called an **almost complete** binary tree or **nearly complete** binary tree if the last level is not completely filled. This type of binary tree is used as a specialized data structure called a **binary heap**.<sup>[20]</sup>
- In the **infinite complete** binary tree, every node has two children (and so the set of levels is **countably infinite**). The set of all nodes is countably infinite, but the set of all infinite paths from the root is uncountable, having the **cardinality of the continuum**. These paths correspond by an order-preserving bijection to the points of the **Cantor set**, or (using the example of a Stern–Brocot tree) to the set of positive **irrational numbers**.
- A **balanced** binary tree has the minimum possible **maximum height** (a.k.a. **depth**) for the leaf nodes, because for any given number of leaf nodes the leaf nodes are placed at the greatest height possible.

#### **h** Balanced Unbalanced, $h = (n + 1)/2 - 1$

0: ABCDE ABCDE

/ \ / \

1: ABCD E ABCD E

/ \ / \

2: AB CD ABC D

/ \ / \ / \

3: A B C D AB C

/ \

4: A B

One common balanced tree structure is a binary tree structure in which the left and right subtrees of every node differ in height by no more than 1.<sup>[21]</sup> One may also consider binary trees where no leaf is much farther away from the root than any other leaf. (Different balancing schemes allow different definitions of “much farther”.<sup>[22]</sup>)

- A **degenerate** (or **pathological**) tree is where each parent node has only one associated child node. This means that performance-wise, the tree will behave like a **linked list** data structure.

### 5.1.3 Properties of binary trees

- The number of nodes  $n$  in a full binary tree, is at least  $n = 2(h + 1) - 1$  and at most  $n = 2^{h+1} - 1$ , where  $h$  is the height of the tree. A tree consisting of only a root node has a height of 0.
- The number of leaf nodes  $l$  in a perfect binary tree, is  $l = (n + 1)/2$  because the number of non-leaf (a.k.a. internal) nodes  $n - l = \sum_{k=0}^{\log_2(l)-1} 2^k = 2^{\log_2(l)} - 1 = l - 1$ .
- This means that a perfect binary tree with  $l$  leaves has  $n = 2l - 1$  nodes.
- In a **balanced** full binary tree,  $h = \lceil \log_2(l) \rceil + 1 = \lceil \log_2((n+1)/2) \rceil + 1 = \lceil \log_2(n+1) \rceil$  (see ceiling function).
- In a **perfect** full binary tree,  $l = 2^h$  thus  $n = 2^{h+1} - 1$ .
- The maximum possible number of null links (i.e., absent children of the nodes) in a **complete** binary tree of  $n$  nodes is  $(n+1)$ , where only 1 node exists in bottom-most level to the far left.
- The number of internal nodes in a **complete** binary tree of  $n$  nodes is  $\lfloor n/2 \rfloor$ .
- For any non-empty binary tree with  $n_0$  leaf nodes and  $n_2$  nodes of degree 2,  $n_0 = n_2 + 1$ .<sup>[23]</sup>

### 5.1.4 Combinatorics

In combinatorics one considers the problem of counting the number of full binary trees of a given size. Here the trees have no values attached to their nodes (this would just multiply the number of possible trees by an easily determined factor), and trees are distinguished only by their structure; however the left and right child of any node are distinguished (if they are different trees, then interchanging them will produce a tree distinct from the original one). The size of the tree is taken to be the number  $n$  of internal nodes (those with two children); the other nodes

are leaf nodes and there are  $n + 1$  of them. The number of such binary trees of size  $n$  is equal to the number of ways of fully parenthesizing a string of  $n + 1$  symbols (representing leaves) separated by  $n$  binary operators (representing internal nodes), so as to determine the argument subexpressions of each operator. For instance for  $n = 3$  one has to parenthesize a string like  $X * X * X * X$ , which is possible in five ways:

$$((X*X)*X)*X, \quad (X*(X*X))*X, \quad (X*X)*(X*X), \quad X*((X*X)*X), \quad X*(X*(X*X))$$

The correspondence to binary trees should be obvious, and the addition of redundant parentheses (around an already parenthesized expression or around the full expression) is disallowed (or at least not counted as producing a new possibility).

There is a unique binary tree of size 0 (consisting of a single leaf), and any other binary tree is characterized by the pair of its left and right children; if these have sizes  $i$  and  $j$  respectively, the full tree has size  $i + j + 1$ . Therefore the number  $C_n$  of binary trees of size  $n$  has the following recursive description  $C_0 = 1$ , and  $C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$  for any positive integer  $n$ . It follows that  $C_n$  is the **Catalan number** of index  $n$ .

The above parenthesized strings should not be confused with the set of words of length  $2n$  in the **Dyck language**, which consist only of parentheses in such a way that they are properly balanced. The number of such strings satisfies the same recursive description (each Dyck word of length  $2n$  is determined by the Dyck subword enclosed by the initial '(' and its matching ')' together with the Dyck subword remaining after that closing parenthesis, whose lengths  $2i$  and  $2j$  satisfy  $i + j + 1 = n$ ); this number is therefore also the Catalan number  $C_n$ . So there are also five Dyck words of length 10:

$$()(), ()(), ()(), ()(), ()()$$

These Dyck words do not correspond in an obvious way to binary trees. A bijective correspondence can nevertheless be defined as follows: enclose the Dyck word in an extra pair of parentheses, so that the result can be interpreted as a **Lisp** list expression (with the empty list () as only occurring atom); then the **dotted-pair** expression for that proper list is a fully parenthesized expression (with NIL as symbol and '.' as operator) describing the corresponding binary tree (which is in fact the internal representation of the proper list).

The ability to represent binary trees as strings of symbols and parentheses implies that binary trees can represent the elements of a **free magma** on a singleton set.

### 5.1.5 Methods for storing binary trees

Binary trees can be constructed from programming language primitives in several ways.

### Nodes and references

In a language with **records** and **references**, binary trees are typically constructed by having a tree node structure which contains some data and references to its left child and its right child. Sometimes it also contains a reference to its unique parent. If a node has fewer than two children, some of the child pointers may be set to a special null value, or to a special **sentinel node**.

This method of storing binary trees wastes a fair bit of memory, as the pointers will be null (or point to the sentinel) more than half the time; a more conservative representation alternative is **threaded binary tree**.<sup>[24]</sup>

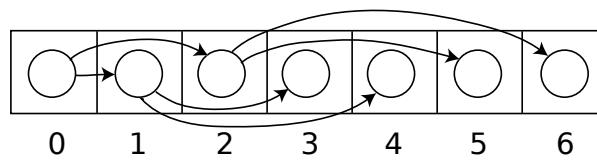
In languages with **tagged unions** such as **ML**, a tree node is often a tagged union of two types of nodes, one of which is a 3-tuple of data, left child, and right child, and the other of which is a “leaf” node, which contains no data and functions much like the null value in a language with pointers. For example, the following line of code in **OCaml** (an ML dialect) defines a binary tree that stores a character in each node.<sup>[25]</sup>

```
type chr_tree = Empty | Node of char * chr_tree * chr_tree
```

### Arrays

Binary trees can also be stored in breadth-first order as an **implicit data structure** in **arrays**, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index  $i$ , its children are found at indices  $2i + 1$  (for the left child) and  $2i + 2$  (for the right), while its parent (if any) is found at index  $\lfloor \frac{i-1}{2} \rfloor$  (assuming the root has index zero). This method benefits from more compact storage and better **locality of reference**, particularly during a preorder traversal. However, it is expensive to grow and wastes space proportional to  $2^h - n$  for a tree of depth  $h$  with  $n$  nodes.

This method of storage is often used for **binary heaps**. No space is wasted because nodes are added in breadth-first order.



### 5.1.6 Encodings

#### Succinct encodings

A succinct data structure is one which occupies close to minimum possible space, as established by **information**

**theoretical** lower bounds. The number of different binary trees on  $n$  nodes is  $C_n$ , the  $n$  th **Catalan number** (assuming we view trees with identical *structure* as identical). For large  $n$ , this is about  $4^n$ ; thus we need at least about  $\log_2 4^n = 2n$  bits to encode it. A succinct binary tree therefore would occupy  $2n + o(n)$  bits.

One simple representation which meets this bound is to visit the nodes of the tree in preorder, outputting “1” for an internal node and “0” for a leaf. If the tree contains data, we can simply simultaneously store it in a consecutive array in preorder. This function accomplishes this:

```
function EncodeSuccinct(node n, bitstring structure, array data) { if n = nil then append 0 to structure; else append 1 to structure; append n.data to data; EncodeSuccinct(n.left, structure, data); EncodeSuccinct(n.right, structure, data); }
```

The string *structure* has only  $2n + 1$  bits in the end, where  $n$  is the number of (internal) nodes; we don't even have to store its length. To show that no information is lost, we can convert the output back to the original tree like this:

```
function DecodeSuccinct(bitstring structure, array data) { remove first bit of structure and put it in b if b = 1 then create a new node n remove first element of data and put it in n.data n.left = DecodeSuccinct(structure, data) n.right = DecodeSuccinct(structure, data) return n else return nil }
```

More sophisticated succinct representations allow not only compact storage of trees but even useful operations on those trees directly while they're still in their succinct form.

#### Encoding general trees as binary trees

There is a one-to-one mapping between general ordered trees and binary trees, which in particular is used by **Lisp** to represent general ordered trees as binary trees. To convert a general ordered tree to binary tree, we only need to represent the general tree in left child-right sibling way. The result of this representation will be automatically binary tree, if viewed from a different perspective. Each node  $N$  in the ordered tree corresponds to a node  $N'$  in the binary tree; the *left child* of  $N'$  is the node corresponding to the first child of  $N$ , and the *right child* of  $N'$  is the node corresponding to  $N$ 's next sibling --- that is, the next node in order among the children of the parent of  $N$ . This binary tree representation of a general order tree is sometimes also referred to as a **left child-right sibling binary tree** (LCRS tree), or a **doubly chained tree**, or a **Filial-Heir chain**.

One way of thinking about this is that each node's children are in a **linked list**, chained together with their *right* fields, and the node only has a pointer to the beginning or head of this list, through its *left* field.

For example, in the tree on the left, A has the 6 children {B,C,D,E,F,G}. It can be converted into the binary tree

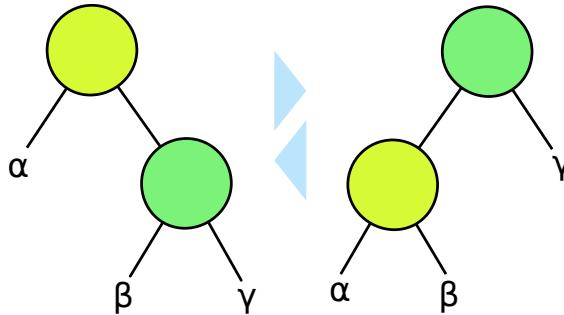
on the right.

The binary tree can be thought of as the original tree tilted sideways, with the black left edges representing *first child* and the blue right edges representing *next sibling*. The leaves of the tree on the left would be written in Lisp as:

`((N O) I J) C D ((P) (Q)) F (M))`

which would be implemented in memory as the binary tree on the right, without any letters on those nodes that have a left child.

### 5.1.7 Common operations



*Tree rotations are very common internal operations on self-balancing binary trees.*

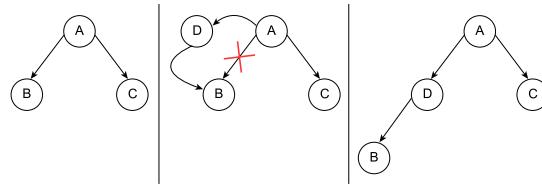
There are a variety of different operations that can be performed on binary trees. Some are **mutator** operations, while others simply return useful information about the tree.

#### Insertion

Nodes can be inserted into binary trees in between two other nodes or added after a **leaf node**. In binary trees, a node that is inserted is specified as to which child it is.

**External nodes** Suppose that the external node being added onto is node A. To add a new node after node A, A assigns the new node as one of its children and the new node assigns node A as its parent.

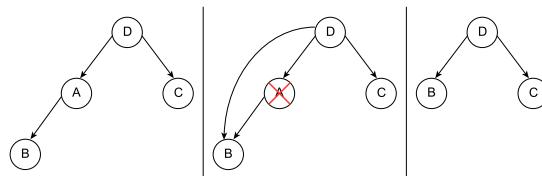
**Internal nodes** Insertion on **internal nodes** is slightly more complex than on external nodes. Say that the internal node is node A and that node B is the child of A. (If the insertion is to insert a right child, then B is the right child of A, and similarly with a left child insertion.) A assigns its child to the new node and the new node assigns its parent to A. Then the new node assigns its child to B and B assigns its parent as the new node.



*The process of inserting a node into a binary tree*

#### Deletion

Deletion is the process whereby a node is removed from the tree. Only certain nodes in a binary tree can be removed unambiguously.<sup>[26]</sup>



*The process of deleting an internal node in a binary tree*

**Node with zero or one children** Suppose that the node to delete is node A. If a node has no children (external node), deletion is accomplished by setting the child of A's parent to **null**. If it has one child, set the parent of A's child to A's parent and set the child of A's parent to A's child.

**Node with two children** In a binary tree, a node with two children cannot be deleted unambiguously.<sup>[26]</sup> However, in certain binary trees (including **binary search trees**) these nodes *can* be deleted, though with a rearrangement of the tree structure.

#### Traversal

Main article: [Tree traversal](#)

Pre-order, in-order, and post-order traversal visit each node in a tree by recursively visiting each node in the left and right subtrees of the root.

**Depth-first order** In depth-first order, we always attempt to visit the node farthest from the root node that we can, but with the caveat that it must be a child of a node we have already visited. Unlike a depth-first search on graphs, there is no need to remember all the nodes we have visited, because a tree cannot contain cycles. Pre-order is a special case of this. See [depth-first search](#) for more information.

**Breadth-first order** Contrasting with depth-first order is breadth-first order, which always attempts to visit the node closest to the root that it has not already visited. See [breadth-first search](#) for more information. Also called a *level-order traversal*.

In a complete binary tree, a node's breadth-index ( $i - (2^d - 1)$ ) can be used as traversal instructions from the root. Reading bitwise from left to right, starting at bit  $d - 1$ , where  $d$  is the node's distance from the root ( $d = \text{floor}(\log_2(i+1))$ ) and the node in question is not the root itself ( $d > 0$ ). When the breadth-index is masked at bit  $d - 1$ , the bit values 0 and 1 mean to step either left or right, respectively. The process continues by successively checking the next bit to the right until there are no more. The rightmost bit indicates the final traversal from the desired node's parent to the node itself. There is a time-space trade-off between iterating a complete binary tree this way versus each node having pointer/s to its sibling/s.

## 5.1.8 See also

- 2–3 tree
- 2–3–4 tree
- AA tree
- Ahnentafel
- AVL tree
- B-tree
- Binary space partitioning
- Huffman tree
- K-ary tree
- Kraft's inequality
- Optimal binary search tree
- Random binary tree
- Recursion (computer science)
- Red–black tree
- Rope (computer science)
- Self-balancing binary search tree
- Splay tree
- Strahler number
- Tree of primitive Pythagorean triples#Alternative methods of generating the tree
- Unrooted binary tree

## 5.1.9 References

### Citations

- [1] Rowan Garnier; John Taylor (2009). *Discrete Mathematics: Proofs, Structures and Applications, Third Edition*. CRC Press. p. 620. ISBN 978-1-4398-1280-8.
- [2] Steven S Skiena (2009). *The Algorithm Design Manual*. Springer Science & Business Media. p. 77. ISBN 978-1-84800-070-4.
- [3] Knuth (1997). *The Art Of Computer Programming, Volume 1, 3/E*. Pearson Education. p. 363. ISBN 0-201-89683-4.
- [4] Iván Flores (1971). *Computer programming system/360*. Prentice-Hall. p. 39.
- [5] Kenneth Rosen (2011). *Discrete Mathematics and Its Applications, 7th edition*. McGraw-Hill Science. p. 749. ISBN 978-0-07-338309-5.
- [6] David R. Mazur (2010). *Combinatorics: A Guided Tour*. Mathematical Association of America. p. 246. ISBN 978-0-88385-762-5.
- [7] Alfred V. Aho; John E. Hopcroft; Jeffrey D. Ullman (1983). *Data Structures and Algorithms*. Pearson Education. section 3.4: “Binary trees”. ISBN 978-81-7758-826-2.
- [8] J.A. Storer (2002). *An Introduction to Data Structures and Algorithms*. Springer Science & Business Media. p. 127. ISBN 978-1-4612-6601-3.
- [9] David Makinson (2009). *Sets, Logic and Maths for Computing*. Springer Science & Business Media. p. 199. ISBN 978-1-84628-845-6.
- [10] Jonathan L. Gross (2007). *Combinatorial Methods with Computer Applications*. CRC Press. p. 248. ISBN 978-1-58488-743-0.
- [11] Hazewinkel, Michiel, ed. (2001), “Binary tree”, *Encyclopedia of Mathematics*, Springer, ISBN 978-1-55608-010-4 also in print as Michiel Hazewinkel (1997). *Encyclopaedia of Mathematics. Supplement I*. Springer Science & Business Media. p. 124. ISBN 978-0-7923-4709-5.
- [12] L.R. Foulds (1992). *Graph Theory Applications*. Springer Science & Business Media. p. 32. ISBN 978-0-387-97599-3.
- [13] Kenneth Rosen (2011). *Discrete Mathematics and Its Applications 7th edition*. McGraw-Hill Science. pp. 352–353. ISBN 978-0-07-338309-5.
- [14] Te Chiang Hu; Man-tak Shing (2002). *Combinatorial Algorithms*. Courier Dover Publications. p. 162. ISBN 978-0-486-41962-6.
- [15] Lih-Hsing Hsu; Cheng-Kuan Lin (2008). *Graph Theory and Interconnection Networks*. CRC Press. p. 66. ISBN 978-1-4200-4482-9.

- [16] J. Flum; M. Grohe (2006). *Parameterized Complexity Theory*. Springer. p. 245. ISBN 978-3-540-29953-0.
- [17] “full binary tree”. NIST.
- [18] Richard Stanley, *Enumerative Combinatorics*, volume 2, p.36
- [19] “perfect binary tree”. NIST.
- [20] “complete binary tree”. NIST.
- [21] Aaron M. Tenenbaum, et al. *Data Structures Using C*, Prentice Hall, 1990 ISBN 0-13-199746-7
- [22] Paul E. Black (ed.), entry for *data structure* in *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology. 15 December 2004. *Online version Accessed 2010-12-19*.
- [23] Mehta, Dinesh; Sartaj Sahni (2004). *Handbook of Data Structures and Applications*. Chapman and Hall. ISBN 1-58488-435-5.
- [24] D. Samanta (2004). *Classic Data Structures*. PHI Learning Pvt. Ltd. pp. 264–265. ISBN 978-81-203-1874-8.
- [25] Michael L. Scott (2009). *Programming Language Pragmatics* (3rd ed.). Morgan Kaufmann. p. 347. ISBN 978-0-08-092299-7.
- [26] Dung X. Nguyen (2003). “Binary Tree Structure”. rice.edu. Retrieved December 28, 2010.

## Bibliography

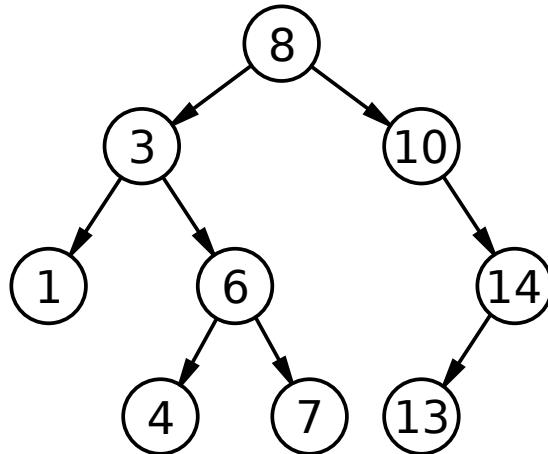
- Donald Knuth. *The art of computer programming vol 1. Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.3, especially subsections 2.3.1–2.3.2 (pp. 318–348).

### 5.1.10 External links

- binary trees entry in the [FindStat](#) database
- Gamedev.net introduction on binary trees
- Binary Tree Proof by Induction
- Balanced binary search tree on array How to create bottom-up an Ahnentafel list, or a balanced binary search tree on array

## 5.2 Binary search tree

In computer science, **binary search trees (BST)**, sometimes called **ordered** or **sorted binary trees**, are a particular type of **containers**: **data structures** that store “items” (such as numbers, names, etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either **dynamic sets** of items, or lookup



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

tables that allow finding an item by its *key* (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of **binary search**: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes **time proportional to the logarithm** of the number of items stored in the tree. This is much better than the **linear time** required to find items by key in an (unsorted) array, but slower than the corresponding operations on **hash tables**.

### 5.2.1 Definition

A binary search tree is a **rooted binary tree**, whose internal nodes each store a *key* (and optionally, an associated *value*) and each have two distinguished sub-trees, commonly denoted *left* and *right*. The tree additionally satisfies the **binary search tree property**, which states that the key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree.<sup>[1]</sup> (The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another. Leaves are commonly represented by a special leaf or nil symbol, a NULL pointer, etc.)

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their *keys* rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the related **sorting algorithms** and **search algorithms** such as **in-order traversal** can be very

efficient; they are also easy to code.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as [sets](#), [multisets](#), and [associative arrays](#). Some of their disadvantages are as follows:

- The shape of the binary search tree totally depends on the order of insertions, and it can be degenerated.
- When inserting or searching for an element in binary search tree, the key of each visited node has to be compared with the key of the element to be inserted or found, i.e., it takes a long time to search an element in a binary search tree.
- The keys in the binary search tree may be long and the run time may increase.
- After a long intermixed sequence of random insertion and deletion, the expected height of the tree approaches square root of the number of keys,  $\sqrt{n}$ , which grows much faster than  $\log n$ .

## 5.2.2 Operations

Binary search trees support three main operations: insertion of keys, deletion of keys, and lookup (checking whether a key is present). Each requires a *comparator*, a subroutine that computes the total order (linear order) on any two keys. This comparator can be explicitly or implicitly defined, depending on the language in which the binary search tree was implemented. A common comparator is the less-than function, for example,  $a < b$ , where  $a$  and  $b$  are keys of two nodes  $a$  and  $b$  in a binary search tree.

### Searching

Searching a binary search tree for a specific key can be a recursive or an iterative process.

We begin by examining the [root node](#). If the tree is [null](#), the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is [null](#). If the searched key is not found before a [null](#) subtree is reached, then the item must not be present in the tree. This is easily expressed as a recursive algorithm:

```
function Find-recursive(key, node): // call initially with
node = root if node = Null or node.key = key then return
node else if key < node.key then return Find-recursive(key,
node.left) else return Find-recursive(key,
node.right)
```

The same algorithm can be implemented iteratively:

```
function Find(key, root): current-node := root while
current-node is not Null do if current-node.key = key
then return current-node else if key < current-node.key
then current-node ← current-node.left else current-node
← current-node.right return Null
```

Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's *height* (see [tree terminology](#)). On average, binary search trees with  $n$  nodes have  $O(\log n)$  height. However, in the worst case, binary search trees can have  $O(n)$  height, when the unbalanced tree resembles a [linked list](#) (degenerate tree).

### Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in a binary tree in [C++](#):

```
void insert(Node*& root, int data) { if (!root) root = new
Node(data); else if (data < root->data) insert(root->left,
data); else if (data > root->data) insert(root->right, data);
}
```

The above *destructive* procedural variant modifies the tree in place. It uses only constant heap space (and the iterative version uses constant stack space as well), but the prior version of the tree is lost. Alternatively, as in the following [Python](#) example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a [persistent data structure](#):

```
def binary_tree_insert(node, key, value): if node is
None: return TreeNode(None, key, value, None) if
key == node.key: return TreeNode(node.left, key,
value, node.right) if key < node.key: return
TreeNode(binary_tree_insert(node.left, key, value),
node.key, node.value, node.right) else: return
TreeNode(node.left, node.key, node.value,
binary_tree_insert(node.right, key, value))
```

The part that is rebuilt uses  $O(\log n)$  space in the average case and  $O(n)$  in the worst case (see [big-O notation](#)).

In either version, this operation requires time proportional to the height of the tree in the worst case, which is  $O(\log n)$ .

$n$ ) time in the average case over all trees, but  $O(n)$  time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its key.

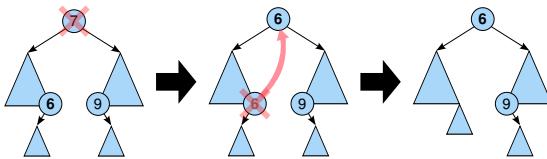
There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

## Deletion

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted  $N$ . Do not delete  $N$ . Instead, choose either its **in-order** successor node or its in-order predecessor node,  $R$ . Copy the value of  $R$  to  $N$ , then recursively call delete on  $R$  until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of first 2 cases.

Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



*Deleting a node with two children from a binary search tree. First the rightmost node in the left subtree, the inorder predecessor 6, is identified. Its value is copied into the node being deleted. The inorder predecessor can then be easily deleted because it has at most one child. The same method works symmetrically using the inorder successor labelled 9.*

Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can

lead to an unbalanced tree, so some implementations select one or the other at different times.

**Runtime analysis:** Although this operation does not always traverse the tree down to a leaf, this is always a possibility; thus in the worst case it requires time proportional to the height of the tree. It does not require more even when the node has two children, since it still follows a single path and does not visit any node twice.

```
def find_min(self): # Gets minimum node in a subtree
 current_node = self
 while current_node.left_child:
 current_node = current_node.left_child
 return current_node
def replace_node_in_parent(self, new_value=None):
 if self.parent:
 if self == self.parent.left_child:
 self.parent.left_child = new_value
 else:
 self.parent.right_child = new_value
 if new_value:
 new_value.parent = self.parent
def binary_tree_delete(self, key):
 if key < self.key:
 self.left_child.binary_tree_delete(key)
 elif key > self.key:
 self.right_child.binary_tree_delete(key)
 else: # delete the key here
 if self.left_child and self.right_child:
 # if both children are present
 successor = self.right_child.find_min()
 self.key = successor.key
 successor.binary_tree_delete(successor.key)
 elif self.left_child:
 # if the node has only a *left* child
 self.replace_node_in_parent(self.left_child)
 elif self.right_child:
 # if the node has only a *right* child
 self.replace_node_in_parent(self.right_child)
 else: # this node has no children
 self.replace_node_in_parent(None)
```

## Traversal

Main article: [Tree traversal](#)

Once the binary search tree has been created, its elements can be retrieved **in-order** by recursively traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as it's recursively accessed. As with all binary trees, one may conduct a **pre-order traversal** or a **post-order traversal**, but neither are likely to be useful for binary search trees. An in-order traversal of a binary search tree will always result in a sorted list of node items (numbers, strings or other comparable items).

The code for in-order traversal in Python is given below. It will call **callback** for every node in the tree.

```
def traverse_binary_tree(node, callback):
 if node is None:
 return
 traverse_binary_tree(node.leftChild, callback)
 callback(node.value)
 traverse_binary_tree(node.rightChild, callback)
```

Traversal requires  $O(n)$  time, since it must visit every node. This algorithm is also  $O(n)$ , so it is **asymptotically**

optimal.

## Sort

Main article: Tree sort

A binary search tree can be used to implement a simple **sorting algorithm**. Similar to **heapsort**, we insert all the values we wish to sort into a new ordered data structure—in this case a binary search tree—and then traverse it in order.

The worst-case time of `build_binary_tree` is  $O(n^2)$  — if you feed it a sorted list of values, it chains them into a **linked list** with no left subtrees. For example, `build_binary_tree([1, 2, 3, 4, 5])` yields the tree (1 (2 (3 (4 (5))))).

There are several schemes for overcoming this flaw with simple binary trees; the most common is the **self-balancing binary search tree**. If this same procedure is done using such a tree, the overall worst-case time is  $O(n \log n)$ , which is **asymptotically optimal** for a **comparison sort**. In practice, the poor **cache** performance and added overhead in time and space for a tree-based sort (particularly for **node allocation**) make it inferior to other asymptotically optimal sorts such as **heapsort** for static list sorting. On the other hand, it is one of the most efficient methods of *incremental sorting*, adding items to a list over time while keeping the list sorted at all times.

## Verification

Sometimes we already have a binary tree, and we need to determine whether it is a BST. This is an interesting problem which has a simple recursive solution.

The BST property—every node on the right subtree has to be larger than the current node and every node on the left subtree has to be smaller than (or equal to) — should not be the case as only unique values should be in the tree — this also poses the question as to if such nodes should be left or right of this parent) the current node—is the key to figuring out whether a tree is a BST or not. On first thought it might look like we can simply traverse the tree, at every node check whether the node contains a value larger than the value at the left child and smaller than the value on the right child, and if this condition holds for all the nodes in the tree then we have a BST. This is the so-called “Greedy approach,” making a decision based on local properties. But this approach clearly won’t work for the following tree:

20 /\ 10 30 /\ 5 40

In the tree above, each node meets the condition that the node contains a value larger than its left child and smaller than its right child hold, and yet it is not a BST: the value 5 is on the right subtree of the node containing 20, a vio-

lation of the BST property!

How do we solve this? It turns out that instead of making a decision based solely on the values of a node and its children, we also need information flowing down from the parent as well. In the case of the tree above, if we could remember about the node containing the value 20, we would see that the node with value 5 is violating the BST property contract.

So the condition we need to check at each node is: a) if the node is the left child of its parent, then it must be smaller than (or equal to) the parent and it must pass down the value from its parent to its right subtree to make sure none of the nodes in that subtree is greater than the parent, and similarly b) if the node is the right child of its parent, then it must be larger than the parent and it must pass down the value from its parent to its left subtree to make sure none of the nodes in that subtree is lesser than the parent.

A simple but elegant recursive solution in C++ can explain this further:

```
struct TreeNode { int data; TreeNode *left; TreeNode
*right; }; bool isBST(TreeNode *node, int minData,
int maxData) { if(node == NULL) return true; if(node->data < minData || node->data > maxData) return false;
return isBST(node->left, minData, node->data) &&
isBST(node->right, node->data, maxData); }
```

The initial call to this function can be something like this:

```
if(isBST(root, INT_MIN, INT_MAX)) { puts("This is a
BST."); } else { puts("This is NOT a BST!"); }
```

Essentially we keep creating a valid range (starting from **[MIN\_VALUE, MAX\_VALUE]**) and keep shrinking it down for each node as we go down recursively.

## Priority queue operations

Binary search trees can serve as **priority queues**: structures that allow insertion of arbitrary key as well as lookup and deletion of the minimum (or maximum) key. Insertion works as previously explained. *Find-min* walks the tree, following left pointers as far as it can without hitting a leaf:

```
// Precondition: T is not a leaf
function find-min(T):
 while hasLeft(T): T ← left(T)
 return key(T)
```

*Find-max* is analogous: follow right pointers as far as possible. *Delete-min (max)* can simply look up the minimum (maximum), then delete it. This way, insertion and deletion both take logarithmic time, just as they do in a **binary heap**, but unlike a binary heap and most other priority queue implementations, a single tree can support all of *find-min*, *find-max*, *delete-min* and *delete-max* at the same time, making binary search trees suitable as **double-ended priority queues**.<sup>[2]:156</sup>

### 5.2.3 Types

There are many types of binary search trees. AVL trees and red-black trees are both forms of self-balancing binary search trees. A **splay tree** is a binary search tree that automatically moves frequently accessed elements nearer to the root. In a **treap** (*tree heap*), each node also holds a (randomly chosen) priority and the parent node has higher priority than its children. Tango trees are trees optimized for fast searches.

Two other titles describing binary search trees are that of a *complete* and *degenerate* tree.

A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. In complete binary tree, all nodes are far left as possible. It is a tree with  $n$  levels, where for each level  $d \leq n - 1$ , the number of existing nodes at level  $d$  is equal to  $2^d$ . This means all possible nodes exist at these levels. An additional requirement for a complete binary tree is that for the  $n$ th level, while every node does not have to exist, the nodes that do exist must fill from left to right.

A degenerate tree is a tree where for each parent node, there is only one associated child node. It is unbalanced and, in the worst case, performance degrades to that of a linked list. If your added node function does not handle re-balancing, then you can easily construct a degenerate tree by feeding it with data that is already sorted. What this means is that in a performance measurement, the tree will essentially behave like a linked list data structure.

#### Performance comparisons

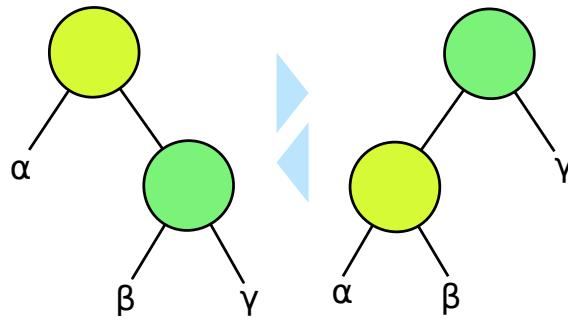
D. A. Heger (2004)<sup>[3]</sup> presented a performance comparison of binary search trees. Treap was found to have the best average performance, while red-black tree was found to have the smallest amount of performance variations.

#### Optimal binary search trees

Main article: [Optimal binary search tree](#)

If we do not plan on modifying a search tree, and we know exactly how often each item will be accessed, we can construct<sup>[4]</sup> an *optimal binary search tree*, which is a search tree where the average cost of looking up an item (the *expected search cost*) is minimized.

Even if we only have estimates of the search costs, such a system can considerably speed up lookups on average. For example, if you have a BST of English words used in a spell checker, you might balance the tree based on word frequency in **text corpora**, placing words like *the* near the root and words like *agerasia* near the leaves. Such a tree might be compared with **Huffman trees**, which similarly seek to place frequently used items near the root in order to produce a dense information encoding; however, Huff-



*Tree rotations are very common internal operations in binary trees to keep perfect, or near-to-perfect, internal balance in the tree.*

man trees store data elements only in leaves, and these elements need not be ordered.

If we do not know the sequence in which the elements in the tree will be accessed in advance, we can use **splay trees** which are asymptotically as good as any static search tree we can construct for any particular sequence of lookup operations.

**Alphabetic trees** are Huffman trees with the additional constraint on order, or, equivalently, search trees with the modification that all elements are stored in the leaves. Faster algorithms exist for *optimal alphabetic binary trees* (OABTs).

#### 5.2.4 See also

- Search tree
- Binary search algorithm
- Randomized binary search tree
- Tango trees
- Self-balancing binary search tree
- Geometry of binary search trees
- Red-black tree
- AVL trees
- Day-Stout-Warren algorithm

#### 5.2.5 References

- [1] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. p. 287. ISBN 0-262-03384-4.
- [2] Mehlhorn, Kurt; Sanders, Peter (2008). *Algorithms and Data Structures: The Basic Toolbox*. Springer.

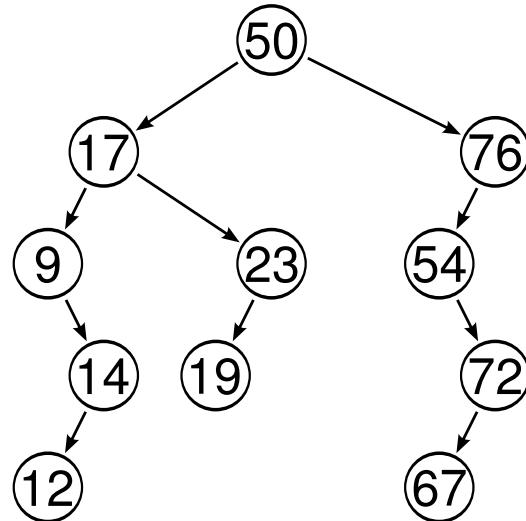
- [3] Heger, Dominique A. (2004), “A Disquisition on The Performance Behavior of Binary Search Tree Data Structures” (PDF), *European Journal for the Informatics Professional* 5 (5): 67–75
- [4] Gonnet, Gaston. “Optimal Binary Search Trees”. *Scientific Computation*. ETH Zürich. Retrieved 1 December 2013.

### 5.2.6 Further reading

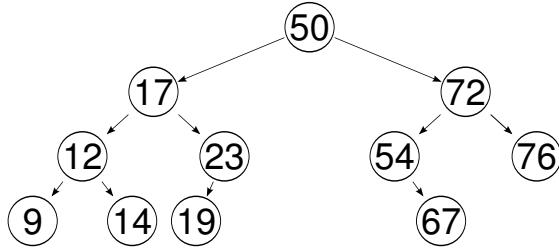
- Black, Paul E. “Binary Search Tree”. *Dictionary of Algorithms and Data Structures*. NIST.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). “12: Binary search trees, 15.5: Optimal binary search trees”. *Introduction to Algorithms* (2nd ed.). MIT Press & McGraw-Hill. pp. 253–272, 356–363. ISBN 0-262-03293-7.
- Jarc, Duane J. (3 December 2005). “Binary Tree Traversals”. *Interactive Data Structure Visualizations*. University of Maryland.
- Knuth, Donald (1997). “6.2.2: Binary Tree Searching”. *The Art of Computer Programming*. 3: “Sorting and Searching” (3rd ed.). Addison-Wesley. pp. 426–458. ISBN 0-201-89685-0.
- Long, Sean. “Binary Search Tree” (PPT). *Data Structures and Algorithms Visualization-A PowerPoint Slides Based Approach*. SUNY Oneonta.
- Parlante, Nick (2001). “Binary Trees”. *CS Education Library*. Stanford University.

### 5.2.7 External links

- Literate implementations of binary search trees in various languages on [LiteratePrograms](#)
- [Binary Tree Visualizer](#) (JavaScript animation of various BT-based data structures)
- Kovac, Kubo. “Binary Search Trees” (JAVA APPLET). Korešpondenčný seminár z programovania.
- Madru, Justin (18 August 2009). “Binary Search Tree”. *JDServer*. C++ implementation.
- [Binary Search Tree Example in Python](#)
- “References to Pointers (C++)”. *MSDN*. Microsoft. 2005. Gives an example binary tree implementation.



An example of an **unbalanced** tree; following the path from the root to a node takes an average of 3.27 node accesses



The same tree after being **height-balanced**; the average path effort decreased to 3.00 node accesses

## 5.3 Self-balancing binary search tree

In computer science, a **self-balancing** (or **height-balanced**) **binary search tree** is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions.<sup>[1]</sup>

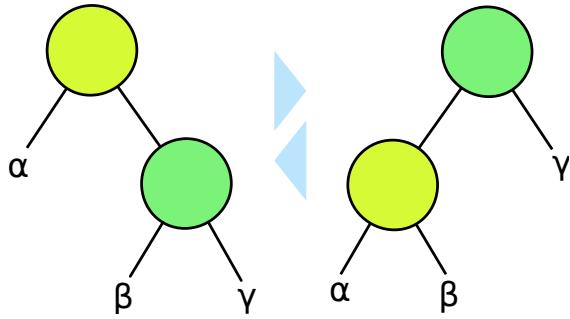
These structures provide efficient implementations for mutable ordered lists, and can be used for other abstract data structures such as associative arrays, priority queues and sets.

### 5.3.1 Overview

Most operations on a binary search tree (BST) take time directly proportional to the height of the tree, so it is desirable to keep the height small. A binary tree with height  $h$  can contain at most  $2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$  nodes. It follows that for a tree with  $n$  nodes and height  $h$ :

$$n \leq 2^{h+1} - 1$$

And that implies:



Tree rotations are very common internal operations on self-balancing binary trees to keep perfect or near-to-perfect balance.

$$h \geq \lceil \log_2(n+1) - 1 \rceil \geq \lfloor \log_2 n \rfloor.$$

In other words, the minimum height of a tree with  $n$  nodes is  $\log_2(n)$ , rounded down; that is,  $\lfloor \log_2 n \rfloor$ .<sup>[1]</sup>

However, the simplest algorithms for BST item insertion may yield a tree with height  $n$  in rather common situations. For example, when the items are inserted in sorted key order, the tree degenerates into a [linked list](#) with  $n$  nodes. The difference in performance between the two situations may be enormous: for  $n = 1,000,000$ , for example, the minimum height is  $\lfloor \log_2(1,000,000) \rfloor = 19$ .

If the data items are known ahead of time, the height can be kept small, in the average sense, by adding values in a random order, resulting in a [random binary search tree](#). However, there are many situations (such as [online algorithms](#)) where this randomization is not viable.

Self-balancing binary trees solve this problem by performing transformations on the tree (such as [tree rotations](#)) at key times, in order to keep the height proportional to  $\log_2(n)$ . Although a certain [overhead](#) is involved, it may be justified in the long run by ensuring fast execution of later operations.

Maintaining the height always at its minimum value  $\lfloor \log_2(n) \rfloor$  is not always viable; it can be proven that any insertion algorithm which did so would have an excessive overhead. Therefore, most self-balanced BST algorithms keep the height within a constant factor of this lower bound.

In the [asymptotic](#) ("Big-O") sense, a self-balancing BST structure containing  $n$  items allows the [lookup](#), [insertion](#), and [removal](#) of an item in  $O(\log n)$  worst-case time, and [ordered enumeration](#) of all items in  $O(n)$  time. For some implementations these are per-operation time bounds, while for others they are [amortized](#) bounds over a sequence of operations. These times are asymptotically optimal among all data structures that manipulate the key only through comparisons.

### 5.3.2 Implementations

Popular data structures implementing this type of tree include:

- 2-3 tree
- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

### 5.3.3 Applications

Self-balancing binary search trees can be used in a natural way to construct and maintain ordered lists, such as [priority queues](#). They can also be used for [associative arrays](#); key-value pairs are simply inserted with an ordering based on the key alone. In this capacity, self-balancing BSTs have a [number of advantages](#) and [disadvantages](#) over their main competitor, [hash tables](#). One advantage of self-balancing BSTs is that they allow fast (indeed, asymptotically optimal) enumeration of the items *in key order*, which hash tables do not provide. One disadvantage is that their lookup algorithms get more complicated when there may be multiple items with the same key. Self-balancing BSTs have better worst-case lookup performance than hash tables ( $O(\log n)$  compared to  $O(n)$ ), but have worse average-case performance ( $O(\log n)$  compared to  $O(1)$ ).

Self-balancing BSTs can be used to implement any algorithm that requires mutable ordered lists, to achieve optimal worst-case asymptotic performance. For example, if [binary tree sort](#) is implemented with a self-balanced BST, we have a very simple-to-describe yet [asymptotically optimal](#)  $O(n \log n)$  sorting algorithm. Similarly, many algorithms in [computational geometry](#) exploit variations on self-balancing BSTs to solve problems such as the [line segment intersection problem](#) and the [point location problem](#) efficiently. (For average-case performance, however, self-balanced BSTs may be less efficient than other solutions. [Binary tree sort](#), in particular, is likely to be slower than [merge sort](#), [quicksort](#), or [heapsort](#), because of the tree-balancing overhead as well as [cache access patterns](#).)

Self-balancing BSTs are flexible data structures, in that it's easy to extend them to efficiently record additional information or perform new operations. For example, one can record the number of nodes in each subtree having a certain property, allowing one to count the number of nodes in a certain key range with that property in  $O(\log n)$  time. These extensions can be used, for example, to

optimize database queries or other list-processing algorithms.

### 5.3.4 See also

- Day–Stout–Warren algorithm
- Fusion tree
- Skip list
- Sorting

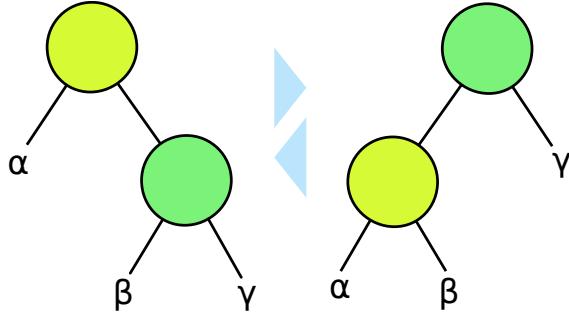
### 5.3.5 References

- [1] Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 6.2.3: Balanced Trees, pp.458–481.

### 5.3.6 External links

- Dictionary of Algorithms and Data Structures: Height-balanced binary search tree
- [GNU libavl](#), a LGPL-licensed library of binary tree implementations in C, with documentation

## 5.4 Tree rotation



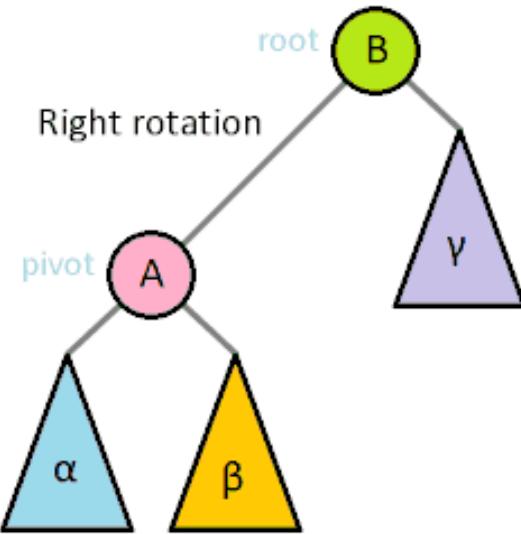
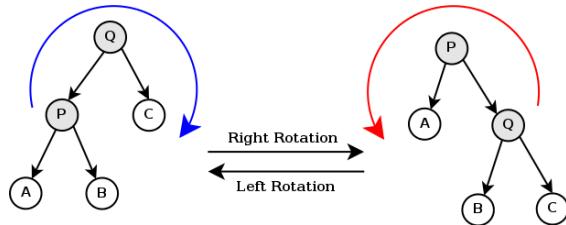
Generic tree rotations.

In discrete mathematics, **tree rotation** is an operation on a binary tree that changes the structure without interfering with the order of the elements. A tree rotation moves one node up in the tree and one node down. It is used to change the shape of the tree, and in particular to decrease its height by moving smaller subtrees down and larger subtrees up, resulting in improved performance of many tree operations.

There exists an inconsistency in different descriptions as to the definition of the **direction of rotations**. Some say that the direction of a rotation depends on the side which the tree nodes are shifted upon whilst others say that it depends on which child takes the root's place (opposite

of the former). This article takes the approach of the side where the nodes get shifted to.

### 5.4.1 Illustration

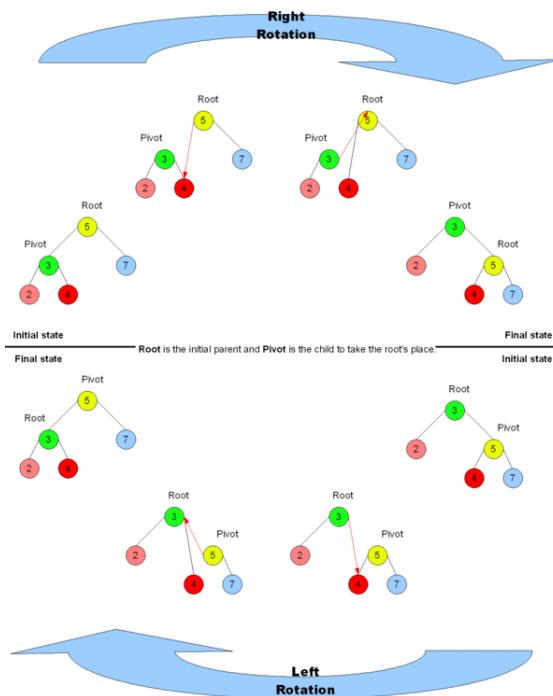


Animation of tree rotations taking place.

The right rotation operation as shown in the image to the left is performed with  $Q$  as the root and hence is a right rotation on, or rooted at,  $Q$ . This operation results in a rotation of the tree in the clockwise direction. The inverse operation is the left rotation, which results in a movement in a counter-clockwise direction (the left rotation shown above is rooted at  $P$ ). The key to understanding how a rotation functions is to understand its constraints. In particular the order of the leaves of the tree (when read left to right for example) cannot change (another way to think of it is that the order that the leaves would be visited in an in-order traversal must be the same after the operation as before). Another constraint is the main property of a binary search tree, namely that the right child is greater than the parent and the left child is less than the parent. Notice that the right child of a left child of the root of a sub-tree (for example node B in the diagram for the tree rooted at  $Q$ ) can become the left child of the root, that itself becomes the right child of the “new” root in the rotated sub-tree, without violating either of those constraints. As you can see in the diagram, the order of the leaves doesn't change. The opposite operation also preserves the order and is the second kind of rotation.

Assuming this is a **binary search tree**, as stated above, the elements must be interpreted as variables that can be compared to each other. The alphabetic characters to the left are used as placeholders for these variables. In the animation to the right, capital alphabetic characters are used as variable placeholders while lowercase Greek letters are placeholders for an entire set of variables. The circles represent individual nodes and the triangles represent subtrees. Each subtree could be empty, consist of a single node, or consist of any number of nodes.

#### 5.4.2 Detailed illustration



Pictorial description of how rotations are made.

When a subtree is rotated, the subtree side upon which it is rotated increases its height by one node while the other subtree decreases its height. This makes tree rotations useful for rebalancing a tree.

Using the terminology of **Root** for the parent node of the subtrees to rotate, **Pivot** for the node which will become the new parent node, **RS** for rotation side upon to rotate and **OS** for opposite side of rotation. In the above diagram for the root Q, the **RS** is C and the **OS** is P. The pseudo code for the rotation is:

```
Pivot = Root.OS
Root.OS = Pivot.RS
Pivot.RS = Root
Root = Pivot
```

This is a constant time operation.

The programmer must also make sure that the root's parent points to the pivot after the rotation. Also, the programmer should note that this operation may result in a new root for the entire tree and take care to update pointers accordingly.

#### 5.4.3 Inorder Invariance

The tree rotation renders the inorder traversal of the binary tree **invariant**. This implies the order of the elements are not affected when a rotation is performed in any part of the tree. Here are the inorder traversals of the trees shown above:

Left tree: ((A, P, B), Q, C) Right tree: (A, P, (B, Q, C))

Computing one from the other is very simple. The following is example **Python** code that performs that computation:

```
def right_rotation(treenode): left, Q, C = treenode.A, P, B = left return (A, P, (B, Q, C))
```

Another way of looking at it is:

Right Rotation of node Q:

Let P be Q's left child. Set Q's left child to be P's right child. Set P's right child to be Q.

Left Rotation of node P:

Let Q be P's right child. Set P's right child to be Q's left child. Set Q's left child to be P.

All other connections are left as-is.

There are also *double rotations*, which are combinations of left and right rotations. A *double left* rotation at X can be defined to be a right rotation at the right child of X followed by a left rotation at X; similarly, a *double right* rotation at X can be defined to be a left rotation at the left child of X followed by a right rotation at X.

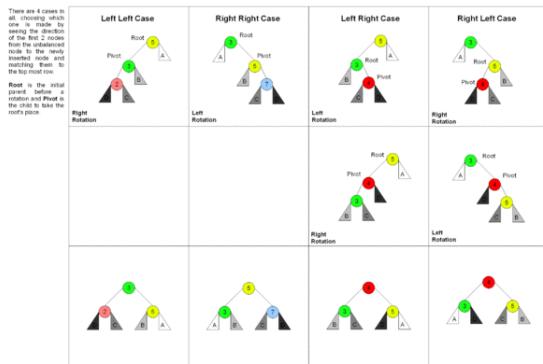
Tree rotations are used in a number of tree data structures such as **AVL trees**, **red-black trees**, **splay trees**, and **treaps**. They require only constant time because they are *local* transformations: they only operate on 5 nodes, and need not examine the rest of the tree.

#### 5.4.4 Rotations for rebalancing

A tree can be rebalanced using rotations. After a rotation, the side of the rotation increases its height by 1 whilst the side opposite the rotation decreases its height similarly. Therefore, one can strategically apply rotations to nodes whose left child and right child differ in height by more than 1. Self-balancing binary search trees apply this operation automatically. A type of tree which uses this rebalancing technique is the **AVL tree**.

#### 5.4.5 Rotation distance

The **rotation distance** between any two binary trees with the same number of nodes is the minimum number of ro-



Pictorial description of how rotations cause rebalancing in an AVL tree.

tations needed to transform one into the other. With this distance, the set of  $n$ -node binary trees becomes a **metric space**: the distance is symmetric, positive when given two different trees, and satisfies the **triangle inequality**.

It is an **open problem** whether there exists a polynomial time algorithm for calculating rotation distance.

Daniel Sleator, Robert Tarjan and William Thurston showed that the rotation distance between any two  $n$ -node trees (for  $n \geq 11$ ) is at most  $2n - 6$ , and that infinitely many pairs of trees are this far apart.<sup>[1]</sup>

#### 5.4.6 See also

- AVL tree, red-black tree, and splay tree, kinds of binary search tree data structures that use rotations to maintain balance.
- Associativity of a binary operation means that performing a tree rotation on it does not change the final result.
- The **Day–Stout–Warren algorithm** balances an unbalanced BST.
- Tamari lattice, a partially ordered set in which the elements can be defined as binary trees and the ordering between elements is defined by tree rotation.

#### 5.4.7 References

- [1] Sleator, Daniel D.; Tarjan, Robert E.; Thurston, William P. (1988), "Rotation distance, triangulations, and hyperbolic geometry", *Journal of the American Mathematical Society* (American Mathematical Society) 1 (3): 647–681, doi:10.2307/1990951, JSTOR 1990951, MR 928904.

#### 5.4.8 External links

- Java applets demonstrating tree rotations

- The **AVL Tree Rotations Tutorial (RTF)** by John Hargrove

## 5.5 Weight-balanced tree

For other uses, see **Optimal binary search tree**.

In computer science, **weight-balanced binary trees (WBTs)** are a type of self-balancing binary search trees that can be used to implement dynamic sets, dictionaries (maps) and sequences.<sup>[1]</sup> These trees were introduced by Nievergelt and Reingold in the 1970s as **trees of bounded balance**, or **BB[ $\alpha$ ] trees**.<sup>[2][3]</sup> Their more common name is due to Knuth.<sup>[4]</sup>

Like other self-balancing trees, WBTs store bookkeeping information pertaining to balance in their nodes and perform **rotations** to restore balance when it is disturbed by insertion or deletion operations. Specifically, each node stores the size of the subtree rooted at the node, and the sizes of left and right subtrees are kept within some factor of each other. Unlike the balance information in AVL trees (which store the height of subtrees) and red-black trees (which store a fictional "color" bit), the bookkeeping information in a WBT is an actually useful property for applications: the number of elements in a tree is equal to the size of its root, and the size information is exactly the information needed to implement the operations of an order statistic tree, viz., getting the  $n$ 'th largest element in a set or determining an element's index in sorted order.<sup>[5]</sup>

Weight-balanced trees are popular in the **functional programming** community and are used to implement sets and maps in **MIT Scheme**, **SLIB** and implementations of **Haskell**.<sup>[6][4]</sup>

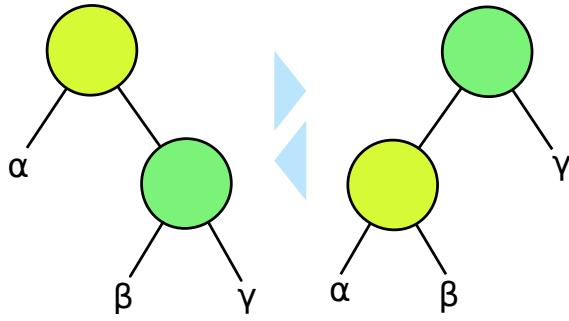
### 5.5.1 Description

A weight-balanced tree is a binary search tree that stores the sizes of subtrees in the nodes. That is, a node has fields

- *key*, of any ordered type
- *value* (optional, only for mappings)
- *left*, *right*, pointer to node
- *size*, of type integer

By definition, the size of a leaf (typically represented by a nil pointer) is zero. The size of an internal node is the sum of sizes of its two children, plus one ( $\text{size}[n] = \text{size}[n.\text{left}] + \text{size}[n.\text{right}] + 1$ ). Based on the size, one defines the weight as either equal to the size, or as  $\text{weight}[n] = \text{size}[n] + 1$ .

Operations that modify the tree must make sure that the weight of the left and right subtrees of every node remain



Binary tree rotations.

within some factor  $\alpha$  of each other, using the same rebalancing operations used in AVL trees: rotations and double rotations. Formally, node balance is defined as follows:

A node is balanced if  $\text{weight}[n.\text{left}] \geq \text{weight}[n]$  and  $\text{weight}[n.\text{right}] \geq \text{weight}[n]$ .<sup>[7]</sup>

Lower values mean more balanced trees, but not all values of  $\alpha$  are appropriate; Nievergelt and Reingold proved that

$$\alpha < 1 - \frac{1}{\sqrt{2}}$$

is a necessary condition for the balancing algorithm to work. Later work showed a lower bound of  $\frac{2}{11}$  for  $\alpha$ , although it can be made arbitrarily small if a custom (and more complicated) rebalancing algorithm is used.<sup>[7]</sup>

Applying balancing correctly guarantees a tree of  $n$  elements will have height<sup>[7]</sup>

$$h \leq \log_{\frac{1}{1-\alpha}} n = \frac{\log_2 n}{\log_2 \left( \frac{1}{1-\alpha} \right)} = O(\log n)$$

The number of balancing operations required in a sequence of  $n$  insertions and deletions is linear in  $n$ , i.e., constant in an amortized sense.<sup>[8]</sup>

## 5.5.2 References

- [1] Tsakalidis, A. K. (1984). “Maintaining order in a generalized linked list”. *Acta Informatica* **21**: 101. doi:10.1007/BF00289142.
- [2] Nievergelt, J.; Reingold, E. M. (1973). “Binary Search Trees of Bounded Balance”. *SIAM Journal on Computing* **2**: 33. doi:10.1137/0202005.
- [3] Black, Paul E. “BB( $\alpha$ ) tree”. *Dictionary of Algorithms and Data Structures*. NIST.
- [4] Hirai, Y.; Yamamoto, K. (2011). “Balancing weight-balanced trees”. *Journal of Functional Programming* **21** (3): 287. doi:10.1017/S0956796811000104.

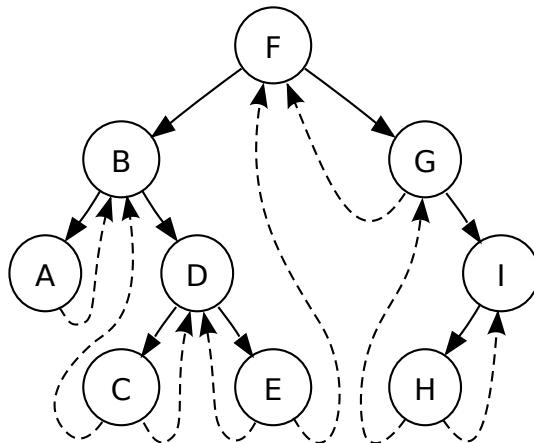
[5] Roura, Salvador (2001). *A new method for balancing binary search trees*. ICALP. Lecture Notes in Computer Science **2076**. pp. 469–480. doi:10.1007/3-540-48224-5\_39. ISBN 978-3-540-42287-7.

[6] Adams, Stephen (1993). “Functional Pearls: Efficient sets—a balancing act”. *Journal of Functional Programming* **3** (4): 553–561. doi:10.1017/S0956796800000885.

[7] Brass, Peter (2008). *Advanced Data Structures*. Cambridge University Press. pp. 61–71.

[8] Blum, Norbert; Mehlhorn, Kurt (1980). “On the average number of rebalancing operations in weight-balanced trees” (PDF). *Theoretical Computer Science* **11** (3): 303–320. doi:10.1016/0304-3975(80)90018-3.

## 5.6 Threaded binary tree



A **threaded tree**, with the special threading links shown by dashed arrows

In computing, a **threaded binary tree** is a binary tree variant that allows fast traversal: given a pointer to a node in a threaded tree, it is possible to cheaply find its in-order successor (and/or predecessor).

### 5.6.1 Motivation

Binary trees, including (but not limited to) binary search trees and their variants, can be used to store a set of items in a particular order. For example, a binary search tree assumes data items are somehow ordered and maintain this ordering as part of their insertion and deletion algorithms. One useful operation on such a tree is *traversal*: visiting the items in the order in which they are stored (which matches the underlying ordering in the case of BST).

A simple recursive traversal algorithm that visits each node of a BST is the following. Assume  $t$  is a pointer to a node, or nil. “Visiting”  $t$  can mean performing any action on the node  $t$  or its contents.

Algorithm traverse( $t$ ):

- Input: a pointer  $t$  to a node (or nil)
- If  $t = \text{nil}$ , return.
- Else:
  - $\text{traverse}(\text{left-child}(t))$
  - Visit  $t$
  - $\text{traverse}(\text{right-child}(t))$

The problem with this algorithm is that, because of its recursion, it uses stack space proportional to the height of a tree. If the tree is fairly balanced, this amounts to  $O(\log n)$  space for a tree containing  $n$  elements. In the worst case, when the tree takes the form of a **chain**, the height of the tree is  $n$  so the algorithm takes  $O(n)$  space.

In 1968, Donald Knuth asked whether a non-recursive algorithm for in-order traversal exists, that uses no stack and leaves the tree unmodified. One of the solutions to this problem is tree threading, presented by J. M. Morris in 1979.<sup>[1][2]</sup>

## 5.6.2 Definition

A threaded binary tree defined as follows:

“A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.”<sup>[3]</sup>

It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack, albeit slowly. This can be useful where stack space is limited, or where a stack of parent pointers is unavailable (for finding the parent pointer via **DFS**).

To see how this is possible, consider a node  $k$  that has a right child  $r$ . Then the left pointer of  $r$  must be either a child or a thread back to  $k$ . In the case that  $r$  has a left child, that left child must in turn have either a left child of its own or a thread back to  $k$ , and so on for all successive left children. So by following the chain of left pointers from  $r$ , we will eventually find a thread pointing back to  $k$ . The situation is symmetrically similar when  $q$  is the left child of  $p$ —we can follow  $q$ ’s right children to a thread pointing ahead to  $p$ .

## 5.6.3 Types of threaded binary trees

1. Single Threaded: each node is threaded towards **either** the in-order predecessor **or** successor (left **or** right).

2. Double threaded: each node is threaded towards **both** the in-order predecessor **and** successor (left **and** right).

In Python:

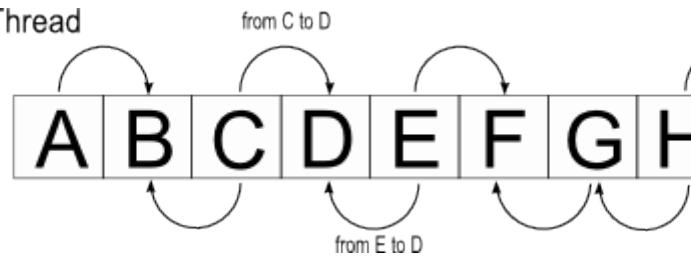
```
def parent(node): if node is node.tree.root: return None
else: x = node y = node while True: if is_thread(y.right):
p = y.right if p is None or p.left is not node: p = x while
not is_thread(p.left): p = p.left p = p.left return p elif
is_thread(x.left): p = x.left if p is None or p.right is not
node: p = y while not is_thread(p.right): p = p.right p =
p.right return p x = x.left y = y.right
```

## 5.6.4 The array of Inorder traversal

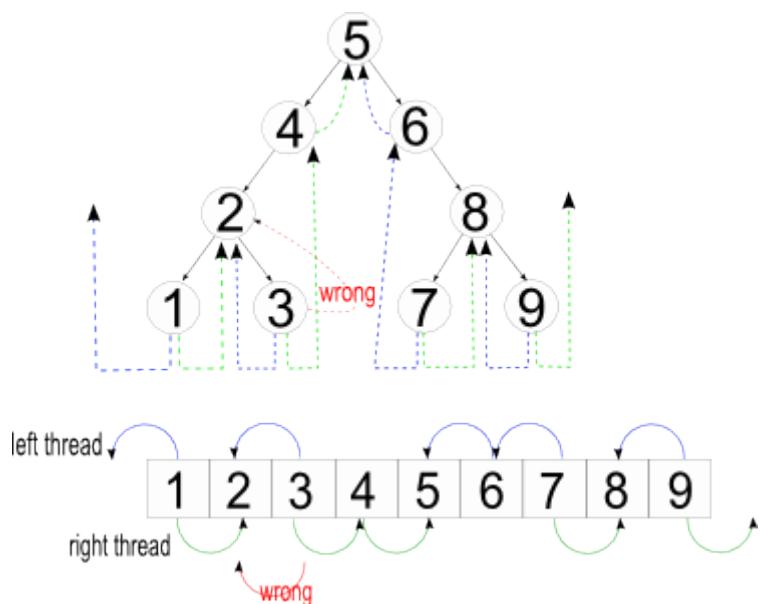
Threads are reference to the predecessors and successors of the node according to an inorder traversal.

Inorder of the threaded tree is ABCDEFGHI, the predecessor of E is D, the successor of E is F.

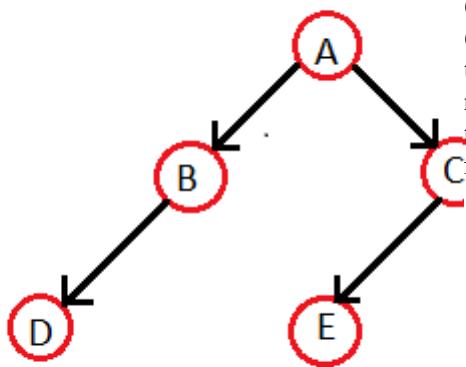
### Right Thread



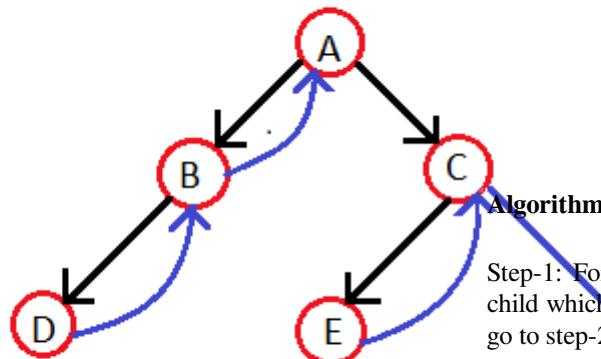
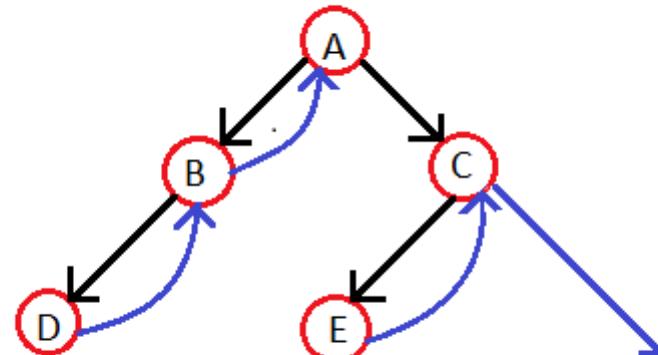
## 5.6.5 Example



Let’s make the Threaded Binary tree out of a normal binary tree



The INORDER traversal for the above tree is—D B A E C. So, the respective Threaded Binary tree will be --



**Algorithm**  
Step-1: For the current node check whether it has a left child which is not there in the visited list. If it has then go to step-2 or else step-3.

Step-2: Put that left child in the list of visited nodes and make it your current node in consideration. Go to step-6.

Step-3: For the current node check whether it has a right child. If it has then go to step-4 else go to step-5

The a=b formula was used in the late 1800s

### 5.6.6 Null link

In an m-way threaded binary tree with n nodes, there are  $n*m - (n-1)$  void links.

### 5.6.7 Non recursive Inorder traversal for a Threaded Binary Tree

As this is a non-recursive method for traversal, it has to be an iterative procedure; meaning, all the steps for the traversal of a node have to be under a loop so that the

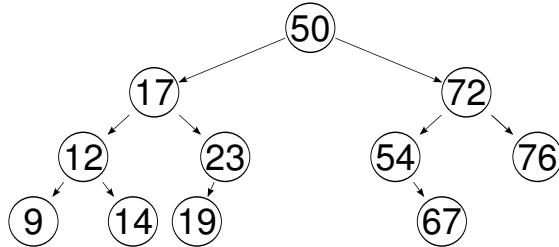
### 5.6.8 References

- [1] Morris, Joseph M. (1979). "Traversing binary trees simply and cheaply". *Information Processing Letters* **9** (5). doi:10.1016/0020-0190(79)90068-1.
- [2] Mateti and, Prabhaker; Manghirmalani, Ravi (1988). "Morris' tree traversal algorithm reconsidered". *Science of Computer Programming* **11**: 29–43. doi:10.1016/0167-6423(88)90063-9.
- [3] Van Wyk, Christopher J. Data Structures and C Programs, Addison-Wesley, 1988, p. 175. ISBN 978-0-201-16116-8.

### 5.6.9 External links

- Tutorial on threaded binary trees
- GNU libavl 2.0.2, Section on threaded binary search trees

## 5.7 AVL tree



Example AVL tree

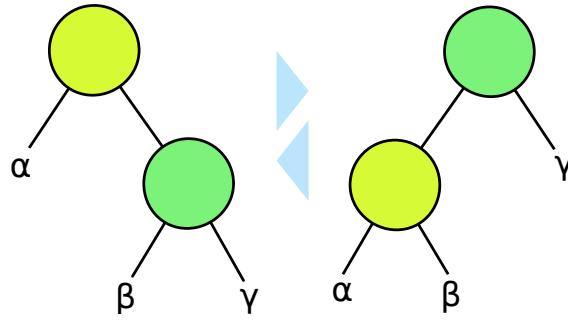
In computer science, an **AVL tree** (Georgy Adelson-Velsky and Landis' tree, named after the inventors) is a self-balancing binary search tree. It was the first such data structure to be invented.<sup>[1]</sup> In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take  $O(\log n)$  time in both the average and worst cases, where  $n$  is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

The AVL tree is named after its two Soviet inventors, **Georgy Adelson-Velsky** and **E. M. Landis**, who published it in their 1962 paper “An algorithm for the organization of information”.<sup>[2]</sup>

AVL trees are often compared with **red-black trees** because both support the same set of operations and take  $O(\log n)$  time for the basic operations. For lookup-intensive applications, AVL trees are faster than red-black trees because they are more rigidly balanced.<sup>[3]</sup> Similar to red-black trees, AVL trees are height-balanced. Both are in general not weight-balanced nor  $\mu$ -balanced for any  $\mu \leq \frac{1}{2}$ ,<sup>[4]</sup> that is, sibling nodes can have hugely differing numbers of descendants.

### 5.7.1 Operations

Basic operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced **binary search tree**, but modifications are followed by zero or more operations called **tree rotations**, which help to restore the height balance of the subtrees.



Tree rotations

### Searching

Searching for a specific key in an AVL Tree can be done the same way as that of a normal unbalanced **Binary Search Tree**.

### Traversal

Once a node has been found in a balanced tree, the *next* or *previous* nodes can be explored in **amortized** constant time. Some instances of exploring these “nearby” nodes require traversing up to  $\log(n)$  links (particularly when moving from the rightmost leaf of the root’s left subtree to the root or from the root to the leftmost leaf of the root’s right subtree; in the example AVL tree, moving from node 14 to the *next but one* node 19 takes 4 steps). However, exploring all  $n$  nodes of the tree in this manner would use each link exactly twice: one traversal to enter the subtree rooted at that node, another to leave that node’s subtree after having explored it. And since there are  $n-1$  links in any tree, the amortized cost is found to be  $2 \times (n-1)/n$ , or approximately 2.

### Insertion

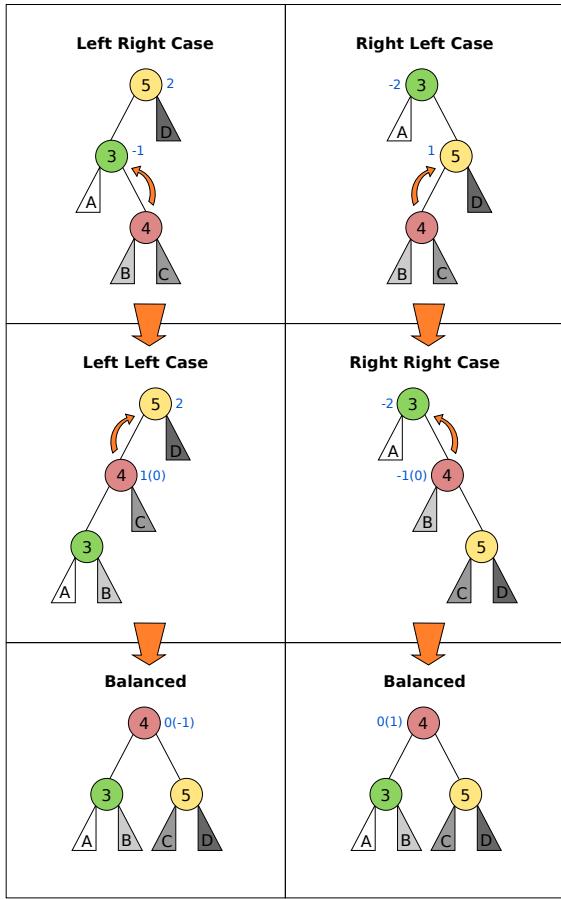
After inserting a node, it is necessary to check each of the node’s ancestors for consistency with the rules of AVL (“retracing”). The balance factor is calculated as follows:

$$\text{balanceFactor} = \text{height(left subtree)} - \text{height(right subtree)}$$

Since with a single insertion the height of an AVL subtree cannot increase by more than one, the temporary balance factor of a node will be in the range from  $-2$  to  $+2$ . For each node checked, if the balance factor remains in the range from  $-1$  to  $+1$  then only corrections of the balance factor, but no rotations are necessary. However, if the balance factor becomes less than  $-1$  or greater than  $+1$ , the subtree rooted at this node is unbalanced.

### Description of the Rotations

Let us first assume the balance factor of a node  $P$  is  $2$  (as opposed to the other possible unbalanced value  $-2$ ). This



Pictorial description of how rotations rebalance a node in AVL tree. The numbered circles represent the nodes being rebalanced. The lettered triangles represent subtrees which are themselves balanced AVL trees. A blue number next to a node denotes possible balance factors (those in parentheses occurring only in case of deletion).

case is depicted in the left column of the illustration with  $P:=5$ . We then look at the left subtree (the higher one) with root  $N$ . If this subtree does not lean to the right - i.e.  $N$  has balance factor 1 (or, when deletion also 0) - we can rotate the whole tree to the right to get a balanced tree. This is labelled as the “Left Left Case” in the illustration with  $N:=4$ . If the subtree does lean to the right - i.e.  $N:=3$  has balance factor -1 - we first rotate the subtree to the left and end up the previous case. This second case is labelled as “Left Right Case” in the illustration.

If the balance factor of the node  $P$  is -2 (this case is depicted in the right column of the illustration  $P:=3$ ) we can mirror the above algorithm. I.e. if the root  $N$  of the (higher) right subtree has balance factor -1 (or, when deletion also 0) we can rotate the whole tree to the left to get a balanced tree. This is labelled as the “Right Right Case” in the illustration with  $N:=4$ . If the root  $N:=5$  of the right subtree has balance factor 1 (“Right Left Case”) we can rotate the subtree to the right to end up in the “Right Right Case”.

The whole retracing loop for an insertion looks like this:

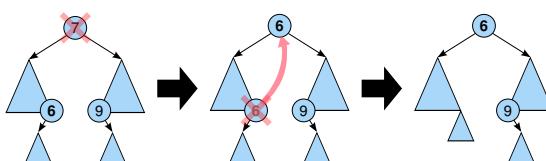
```
// N is the child of P whose height increases by 1. do
{ if (N == left_child(P)) { if (balance_factor(P) == 1) { // The left column in the picture // Temporary balance_factor(P) == 2 ==> rebalancing is required.
if (balance_factor(N) == -1) { // Left Right Case
rotate_left(N); // Reduce to Left Left Case } // Left Left Case
rotate_right(P); break; // Leave the loop } if (balance_factor(P) == -1) { balance_factor(P) = 0; // N's height increase is absorbed at P. break; // Leave the loop } balance_factor(P) = 1; // Height increases at P } else {
// N == right_child(P), the child whose height increases by 1. if (balance_factor(P) == -1) { // The right column in the picture // Temporary balance_factor(P) == -2 ==> rebalancing is required. if (balance_factor(N) == 1) { // Right Left Case
rotate_right(N); // Reduce to Right Right Case } // Right Right Case
rotate_left(P); break; // Leave the loop } if (balance_factor(P) == 1) { balance_factor(P) = 0; // N's height increase is absorbed at P. break; // Leave the loop } balance_factor(P) = -1; // Height increases at P } N = P; P = parent(N); } while
(P != null); // Possibly up to the root
```

After a rotation a subtree has the same height as before, so retracing can stop. In order to restore the balance factors of all nodes, first observe that all nodes requiring correction lie along the path used during the initial insertion. If the above procedure is applied to nodes along this path, starting from the bottom (i.e. the inserted node), then every node in the tree will again have a balance factor of -1, 0, or 1.

The time required is  $O(\log n)$  for lookup, plus a maximum of  $O(\log n)$  retracing levels on the way back to the root, so the operation can be completed in  $O(\log n)$  time.

## Deletion

Let node  $X$  be the node with the value we need to delete, and let node  $Y$  be a node in the tree we need to find to take node  $X$ 's place, and let node  $Z$  be the actual node we take out of the tree.



Deleting a node with two children from a binary search tree using the in-order predecessor (rightmost node in the left subtree, labelled 6).

Steps to consider when deleting a node in an AVL tree are the following:

1. If node  $X$  is a leaf or has only one child, skip to step 5 with  $Z:=X$ .

2. Otherwise, determine node Y by finding the largest node in node X's left subtree (the in-order predecessor of X – it does not have a right child) or the smallest in its right subtree (the in-order successor of X – it does not have a left child).
3. Exchange all the child and parent links of node X with those of node Y. In this step, the in-order sequence between nodes X and Y is temporarily disturbed, but the tree structure doesn't change.
4. Choose node Z to be all the child and parent links of old node Y = those of new node X.
5. If node Z has a subtree (which then is a leaf) attach it to Z's parent.
6. If node Z was the root (its parent is null), update root.
7. Delete node Z.
8. Retrace the path back up the tree (starting with node Z's parent) to the root, adjusting the balance factors as needed.

Since with a single deletion the height of an AVL subtree cannot decrease by more than one, the temporary balance factor of a node will be in the range from  $-2$  to  $+2$ .

If the balance factor becomes  $\pm 2$  then the subtree is unbalanced and needs to be rotated. The various cases of rotations are depicted in section “[Insertion](#)”.

The whole retracing loop for a deletion looks like this:

```
// N is the child of P whose height decreases by 1. do
{ if (N == right_child(P)) { if (balance_factor(P) ==
1) { // The left column in the picture // Temporary
balance_factor(P) == 2 ==> rebalancing is required. S =
left_child(P); // Sibling of N B = balance_factor(S); if
(B == -1) { // Left Right Case rotate_left(S); // Reduce
to Left Left Case } // Left Left Case rotate_right(P); if
(B == 0) // (in the picture the small blue (0) at node 4)
break; // Height does not change: Leave the loop } if
(balance_factor(P) == 0) { balance_factor(P) = 1; // N's
height decrease is absorbed at P. break; // Leave the loop
} balance_factor(P) = 0; // Height decreases at P } else
{ // N == left_child(P), the child whose height decreases
by 1. if (balance_factor(P) == -1) { // The right column
in the picture // Temporary balance_factor(P) == -2
==> rebalancing is required. S = right_child(P); // Sibling
of N B = balance_factor(S); if (B == 1) { // Right
Left Case rotate_right(S); // Reduce to Right Right Case
} // Right Right Case rotate_left(P); if (B == 0) // (in
the picture the small blue (0) at node 4) break; // Height
does not change: Leave the loop } if (balance_factor(P) ==
0) { balance_factor(P) = -1; // N's height decrease is
absorbed at P. break; // Leave the loop } balance_factor(P) =
0; // Height decreases at P } N = P; P = parent(N); } while (P != null); // Possibly up
```

to the root

The retracing can stop if the balance factor becomes  $\pm 1$  indicating that the height of that subtree has remained unchanged. This can also result from a rotation when the higher child tree has a balance factor of 0.

If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. This can also result from a rotation.

The time required is  $O(\log n)$  for lookup, plus a maximum of  $O(\log n)$  retracing levels on the way back to the root, so the operation can be completed in  $O(\log n)$  time.

## 5.7.2 Comparison to other structures

Both AVL trees and red-black trees are self-balancing binary search trees and they are very similar mathematically.<sup>[5]</sup> The operations to balance the trees are different, but both occur on the average in  $O(1)$  with maximum in  $O(\log n)$ . The real difference between the two is the limiting height. For a tree of size  $n$  :

- An AVL tree's height is strictly less than:<sup>[6][7]</sup>

$$\log_{\varphi}(\sqrt{5}(n+2)) - 2 = \frac{\log_2(\sqrt{5}(n+2))}{\log_2(\varphi)} - 2 = \log_{\varphi}(2) \cdot \log_2(\sqrt{5}(n+2)) - 2 \approx 1.44 \log_2(n+2) - 0.328$$

where  $\varphi$  is the golden ratio.

- A red-black tree's height is at most  $2 \log_2(n+1)$ <sup>[8]</sup>

AVL trees are more rigidly balanced than red-black trees, leading to slower insertion and removal but faster retrieval.

## 5.7.3 See also

- Trees
- Tree rotation
- Red-black tree
- Splay tree
- Scapegoat tree
- B-tree
- T-tree
- List of data structures

### 5.7.4 References

- [1] Robert Sedgewick, *Algorithms*, Addison-Wesley, 1983, ISBN 0-201-06672-6, page 199, chapter 15: Balanced Trees.
- [2] Georgy Adelson-Velsky, G.; E. M. Landis (1962). “An algorithm for the organization of information”. *Proceedings of the USSR Academy of Sciences* (in Russian) **146**: 263–266. English translation by Myron J. Ricci in *Soviet Math. Doklady*, 3:1259–1263, 1962.
- [3] Pfaff, Ben (June 2004). “Performance Analysis of BSTs in System Software” (PDF). Stanford University.
- [4] AVL trees are not weight-balanced? (meaning: AVL trees are not  $\mu$ -balanced?)  
Thereby: A Binary Tree is called  $\mu$  -balanced, with  $0 \leq \mu \leq \frac{1}{2}$ , if for every node  $N$ , the inequality  

$$\frac{1}{2} - \mu \leq \frac{|N_L|}{|N|+1} \leq \frac{1}{2} + \mu$$
 holds and  $\mu$  is minimal with this property.  $|N|$  is the number of nodes below the tree with  $N$  as root (including the root) and  $N_L$  is the left child node of  $N$ .
- [5] In fact, each AVL tree can be colored red-black.
- [6] Burkhard, Walt (Spring 2012). “AVL Dictionary Data Type Implementation”. *Advanced Data Structures* (PDF). La Jolla: A.S. Soft Reserves, UC San Diego. p. 103.
- [7] Knuth, Donald E. (2000). *Sorting and searching* (2. ed., 6. printing, newly updated and rev. ed.). Boston [u.a.]: Addison-Wesley. p. 460. ISBN 0-201-89685-0.
- [8] Proof of asymptotic bounds

### 5.7.5 Further reading

- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 458–475 of section 6.2.3: Balanced Trees.

### 5.7.6 External links

- [xdg library](#) by Dmitriy Vilkov: Serializable straight C-implementation could easily be taken from this library under [GNU-LGPL](#) and [AFL v2.0](#) licenses.
- Description from the [Dictionary of Algorithms and Data Structures](#)
- [Python Implementation](#)
- [Single C header file](#) by Ian Piumarta
- [AVL Tree Demonstration](#)
- [AVL tree applet – all operations](#)
- [Fast and efficient implementation of AVL Trees](#)

- [PHP Implementation](#)
- [AVL Threaded Tree PHP Implementation](#)
- [C++ implementation which can be used as an array](#)
- [Self balancing AVL tree with Concat and Split operations](#)

## 5.8 Red-black tree

A **red-black tree** is a [binary search tree](#) with an extra bit of data per node, its color, which can be either red or black.<sup>[1]</sup> The extra bit of storage ensures an approximately balanced tree by constraining how nodes are colored from any path from the root to the leaf.<sup>[1]</sup> Thus, it is a [data structure](#) which is a type of [self-balancing binary search tree](#).

Balance is preserved by painting each node of the tree with one of two colors (typically called 'red' and 'black') in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

The balancing of the tree is not perfect but it is good enough to allow it to guarantee searching in  $O(\log n)$  time, where  $n$  is the total number of elements in the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in  $O(\log n)$  time.<sup>[2]</sup>

Tracking the color of each node requires only 1 bit of information per node because there are only two colors. The tree does not contain any other data specific to its being a red–black tree so its memory footprint is almost identical to a classic (uncolored) binary search tree. In many cases the additional bit of information can be stored at no additional memory cost.

### 5.8.1 History

The original data structure was invented in 1972 by Rudolf Bayer<sup>[3]</sup> and named “symmetric binary B-tree”, but acquired its modern name in a paper in 1978 by Leonidas J. Guibas and Robert Sedgewick entitled “A Dichromatic Framework for Balanced Trees”.<sup>[4]</sup> The color “red” was chosen because it was the best-looking color produced by the color laser printer available to the authors while working at Xerox PARC.<sup>[5]</sup>

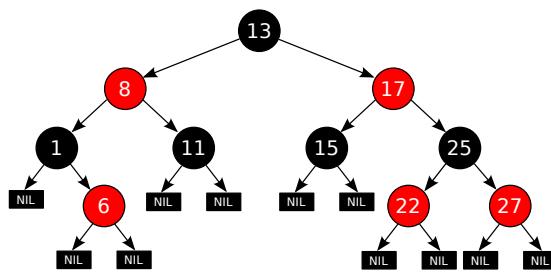
## 5.8.2 Terminology

A red–black tree is a special type of **binary tree**, used in **computer science** to organize pieces of comparable **data**, such as text fragments or numbers.

The **leaf nodes** of red–black trees do not contain data. These leaves need not be explicit in computer memory—a null child pointer can encode the fact that this child is a leaf—but it simplifies some algorithms for operating on red–black trees if the leaves really are explicit nodes. To save memory, sometimes a single **sentinel node** performs the role of all leaf nodes; all references from **internal nodes** to leaf nodes then point to the sentinel node.

Red–black trees, like all **binary search trees**, allow efficient **in-order traversal** (that is: in the order **Left–Root–Right**) of their elements. The search-time results from the traversal from root to leaf, and therefore a balanced tree of  $n$  nodes, having the least possible tree height, results in  $O(\log n)$  search time.

## 5.8.3 Properties



An example of a red–black tree

In addition to the requirements imposed on a **binary search tree** the following must be satisfied by a red–black tree:<sup>[6]</sup>

1. A node is either red or black.
2. The root is black. (This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.)
3. All leaves (NIL) are black. (All leaves are same color as the root.)
4. Every red node must have two black child nodes (and therefore it must have a black parent).
5. Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.

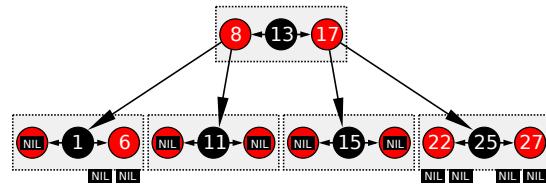
These constraints enforce a critical property of red–black trees: that the path from the root to the furthest leaf is no more than twice as long as the path from the root to the

nearest leaf. The result is that the tree is roughly height-balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red–black trees to be efficient in the worst case, unlike ordinary **binary search trees**.

To see why this is guaranteed, it suffices to consider the effect of properties 4 and 5 together. For a red–black tree  $T$ , let  $B$  be the number of black nodes in property 5. Let the shortest possible path from the root of  $T$  to any leaf consist of  $B$  black nodes. Longer possible paths may be constructed by inserting red nodes. However, property 4 makes it impossible to insert more than one consecutive red node. Therefore the longest possible path consists of  $2B$  nodes, alternating black and red (this is the worst case).

The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes. Since all maximal paths have the same number of black nodes, by property 5, this shows that no path is more than twice as long as any other path.

## 5.8.4 Analogy to B-trees of order 4



The same red–black tree as in the example above, seen as a B-tree.

A red–black tree is similar in structure to a B-tree of order<sup>[note 1]</sup> 4, where each node can contain between 1 and 3 values and (accordingly) between 2 and 4 child pointers. In such a B-tree, each node will contain only one value matching the value in a black node of the red–black tree, with an optional value before and/or after it in the same node, both matching an equivalent red node of the red–black tree.

One way to see this equivalence is to “move up” the red nodes in a graphical representation of the red–black tree, so that they align horizontally with their parent black node, by creating together a horizontal cluster. In the B-tree, or in the modified graphical representation of the red–black tree, all leaf nodes are at the same depth.

The red–black tree is then structurally equivalent to a B-tree of order 4, with a minimum fill factor of 33% of values per cluster with a maximum capacity of 3 values.

This B-tree type is still more general than a red–black tree though, as it allows ambiguity in a red–black tree conversion—multiple red–black trees can be produced from an equivalent B-tree of order 4. If a B-tree clus-

ter contains only 1 value, it is the minimum, black, and has two child pointers. If a cluster contains 3 values, then the central value will be black and each value stored on its sides will be red. If the cluster contains two values, however, either one can become the black node in the red–black tree (and the other one will be red).

So the order-4 B-tree does not maintain which of the values contained in each cluster is the root black tree for the whole cluster and the parent of the other values in the same cluster. Despite this, the operations on red–black trees are more economical in time because you don't have to maintain the vector of values. It may be costly if values are stored directly in each node rather than being stored by reference. B-tree nodes, however, are more economical in space because you don't need to store the color attribute for each node. Instead, you have to know which slot in the cluster vector is used. If values are stored by reference, e.g. objects, null references can be used and so the cluster can be represented by a vector containing 3 slots for value pointers plus 4 slots for child references in the tree. In that case, the B-tree can be more compact in memory, improving data locality.

The same analogy can be made with B-trees with larger orders that can be structurally equivalent to a colored binary tree: you just need more colors. Suppose that you add blue, then the blue–red–black tree defined like red–black trees but with the additional constraint that no two successive nodes in the hierarchy will be blue and all blue nodes will be children of a red node, then it becomes equivalent to a B-tree whose clusters will have at most 7 values in the following colors: blue, red, blue, black, blue, red, blue (For each cluster, there will be at most 1 black node, 2 red nodes, and 4 blue nodes).

For moderate volumes of values, insertions and deletions in a colored binary tree are faster compared to B-trees because colored trees don't attempt to maximize the fill factor of each horizontal cluster of nodes (only the minimum fill factor is guaranteed in colored binary trees, limiting the number of splits or junctions of clusters). B-trees will be faster for performing rotations (because rotations will frequently occur within the same cluster rather than with multiple separate nodes in a colored binary tree). However for storing large volumes, B-trees will be much faster as they will be more compact by grouping several children in the same cluster where they can be accessed locally.

All optimizations possible in B-trees to increase the average fill factors of clusters are possible in the equivalent multicolored binary tree. Notably, maximizing the average fill factor in a structurally equivalent B-tree is the same as reducing the total height of the multicolored tree, by increasing the number of non-black nodes. The worst case occurs when all nodes in a colored binary tree are black, the best case occurs when only a third of them are black (and the other two thirds are red nodes).

Notes

- [1] Using Knuth's definition of order: the maximum number of children

## 5.8.5 Applications and related data structures

Red–black trees offer worst-case guarantees for insertion time, deletion time, and search time. Not only does this make them valuable in time-sensitive applications such as [real-time applications](#), but it makes them valuable building blocks in other data structures which provide worst-case guarantees; for example, many data structures used in [computational geometry](#) can be based on red–black trees, and the [Completely Fair Scheduler](#) used in current Linux kernels uses red–black trees.

The [AVL tree](#) is another structure supporting  $O(\log n)$  search, insertion, and removal. It is more rigidly balanced than red–black trees, leading to slower insertion and removal but faster retrieval. This makes it attractive for data structures that may be built once and loaded without reconstruction, such as language dictionaries (or program dictionaries, such as the opcodes of an assembler or interpreter).

Red–black trees are also particularly valuable in [functional programming](#), where they are one of the most common [persistent data structures](#), used to construct [associative arrays](#) and [sets](#) which can retain previous versions after mutations. The persistent version of red–black trees requires  $O(\log n)$  space for each insertion or deletion, in addition to time.

For every [2-4 tree](#), there are corresponding red–black trees with data elements in the same order. The insertion and deletion operations on 2-4 trees are also equivalent to color-flipping and rotations in red–black trees. This makes 2-4 trees an important tool for understanding the logic behind red–black trees, and this is why many introductory algorithm texts introduce 2-4 trees just before red–black trees, even though 2-4 trees are not often used in practice.

In 2008, [Sedgewick](#) introduced a simpler version of the red–black tree called the [left-leaning red–black tree](#)<sup>[7]</sup> by eliminating a previously unspecified degree of freedom in the implementation. The LLRB maintains an additional invariant that all red links must lean left except during inserts and deletes. Red–black trees can be made isometric to either [2-3 trees](#),<sup>[8]</sup> or 2-4 trees,<sup>[7]</sup> for any sequence of operations. The 2-4 tree isometry was described in 1978 by [Sedgewick](#). With 2-4 trees, the isometry is resolved by a “color flip,” corresponding to a split, in which the red color of two children nodes leaves the children and moves to the parent node. The [tango tree](#), a type of tree optimized for fast searches, usually uses red–black trees as part of its data structure.

## 5.8.6 Operations

Read-only operations on a red–black tree require no modification from those used for [binary search trees](#), because every red–black tree is a special case of a simple binary search tree. However, the immediate result of an insertion or removal may violate the properties of a red–black tree. Restoring the red–black properties requires a small number ( $O(\log n)$  or [amortized  \$O\(1\)\$](#) ) of color changes (which are very quick in practice) and no more than three [tree rotations](#) (two for insertion). Although insert and delete operations are complicated, their times remain  $O(\log n)$ .

### Insertion

Insertion begins by adding the node as any [binary search tree insertion](#) does and by coloring it red. Whereas in the binary search tree, we always add a leaf, in the red–black tree, leaves contain no information, so instead we add a red interior node, with two black leaves, in place of an existing black leaf.

What happens next depends on the color of other nearby nodes. The term *uncle node* will be used to refer to the sibling of a node’s parent, as in human family trees. Note that:

- property 3 (all leaves are black) always holds.
- property 4 (both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation.
- property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is threatened only by adding a black node, repainting a red node black (or vice versa), or a rotation.

*Note:* The label **N** will be used to denote the current node (colored red). At the beginning, this is the new node being inserted, but the entire procedure may also be applied recursively to other nodes (see case 3). **P** will denote **N**’s parent node, **G** will denote **N**’s grandparent, and **U** will denote **N**’s uncle. Note that in between some cases, the roles and labels of the nodes are exchanged, but in each case, every label continues to represent the same node it represented at the beginning of the case. Any color shown in the diagram is either assumed in its case or implied by those assumptions. A numbered triangle represents a subtree of unspecified depth. A black circle atop the triangle designates a black root node, otherwise the root node’s color is unspecified.

There are several cases of red–black tree insertion to handle:

- **N** is the root node, i.e., first node of red–black tree
- **N**’s parent (**P**) is black
- **N**’s parent (**P**) and uncle (**U**) are red
- **N** is added to right of left child of grandparent, or **N** is added to left of right child of grandparent (**P** is red and **U** is black)
- **N** is added to left of left child of grandparent, or **N** is added to right of right child of grandparent (**P** is red and **U** is black)

Each case will be demonstrated with example [C](#) code. The uncle and grandparent nodes can be found by these functions:

```
struct node *grandparent(struct node *n) { if ((n != NULL) && (n->parent != NULL)) return n->parent->parent; else return NULL; } struct node *uncle(struct node *n) { struct node *g = grandparent(n); if (g == NULL) return NULL; // No grandparent means no uncle if (n->parent == g->left) return g->right; else return g->left; }
```

**Case 1:** The current node **N** is at the root of the tree. In this case, it is repainted black to satisfy property 2 (the root is black). Since this adds one black node to every path at once, property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated.

```
void insert_case1(struct node *n) { if (n->parent == NULL) n->color = BLACK; else insert_case2(n); }
```

**Case 2:** The current node’s parent **P** is black, so property 4 (both children of every red node are black) is not invalidated. In this case, the tree is still valid. Property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not threatened, because the current node **N** has two black leaf children, but because **N** is red, the paths through each of its children have the same number of black nodes as the path through the leaf it replaced, which was black, and so this property remains satisfied.

```
void insert_case2(struct node *n) { if (n->parent->color == BLACK) return; /* Tree is still valid */ else insert_case3(n); }
```

*Note:* In the following cases it can be assumed that **N** has a grandparent node **G**, because its parent **P** is red, and if it were the root, it would be black. Thus, **N** also has an uncle node **U**, although it may be a leaf in cases 4 and 5.

```
void insert_case3(struct node *n) { struct node *u = uncle(n), *g; if ((u != NULL) && (u->color == RED))
```

```
{ n->parent->color = BLACK; u->color = BLACK; g =
grandparent(n); g->color = RED; insert_case1(g); } else
{ insert_case4(n); }
```

*Note:* In the remaining cases, it is assumed that the parent node **P** is the left child of its parent. If it is the right child, *left* and *right* should be reversed throughout cases 4 and 5. The code samples take care of this.

```
void insert_case4(struct node *n) { struct node *g
= grandparent(n); if ((n == n->parent->right) &&
(n->parent == g->left)) { rotate_left(n->parent);
/* * rotate_left can be the below because of already having *g = grandparent(n) * * struct node
*saved_p=g->left, *saved_left_n=n->left; * g->left=n;
* n->left=saved_p; * saved_p->right=saved_left_n; * */
and modify the parent's nodes properly */ n = n->left;
} else if ((n == n->parent->left) && (n->parent ==
g->right)) { rotate_right(n->parent); /* * rotate_right
can be the below to take advantage of already having
*g = grandparent(n) * * struct node *saved_p=g-
>right, *saved_right_n=n->right; * g->right=n; *
n->right=saved_p; * saved_p->left=saved_right_n; * */
n = n->right; } insert_case5(n); }
void insert_case5(struct node *n) { struct node *g =
grandparent(n); n->parent->color = BLACK; g->color =
RED; if (n == n->parent->left) rotate_right(g); else
rotate_left(g); }
```

Note that inserting is actually **in-place**, since all the calls above use **tail** recursion.

## Removal

In a regular binary search tree when deleting a node with two non-leaf children, we find either the maximum element in its left subtree (which is the in-order predecessor) or the minimum element in its right subtree (which is the in-order successor) and move its value into the node being deleted (as shown [here](#)). We then delete the node we copied the value from, which must have fewer than two non-leaf children. (Non-leaf children, rather than all children, are specified here because unlike normal binary search trees, red-black trees can have leaf nodes anywhere, so that all nodes are either internal nodes with two children or leaf nodes with, by definition, zero children. In effect, internal nodes having two leaf children in a red-black tree are like the leaf nodes in a regular binary search tree.) Because merely copying a value does not violate any red-black properties, this reduces to the problem of deleting a node with at most one non-leaf child. Once we have solved that problem, the solution applies equally to the case where the node we originally want to delete has at most one non-leaf child as to the case just considered where it has two non-leaf children.

Therefore, for the remainder of this discussion we address the deletion of a node with at most one non-leaf child. We use the label **M** to denote the node to be deleted; **C** will denote a selected child of **M**, which we will also call “its child”. If **M** does have a non-leaf child, call that its child, **C**; otherwise, choose either leaf as its child, **C**.

If **M** is a red node, we simply replace it with its child **C**, which must be black by property 4. (This can only occur when **M** has two leaf children, because if the red node **M** had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would violate property 5.) All paths through the deleted node will simply pass through one fewer red node, and both the deleted node's parent and child must be black, so property 3 (all leaves are black) and property 4 (both children of every red node are black) still hold.

Another simple case is when **M** is black and **C** is red. Simply removing a black node could break Properties 4 (“Both children of every red node are black”) and 5 (“All paths from any given node to its leaf nodes contain the same number of black nodes”), but if we repaint **C** black, both of these properties are preserved.

The complex case is when both **M** and **C** are black. (This can only occur when deleting a black node which has two leaf children, because if the black node **M** had a black non-leaf child on one side but just a leaf child on the other side, then the count of black nodes on both sides would be different, thus the tree would have been an invalid red-black tree by violation of property 5.) We begin by replacing **M** with its child **C**. We will call (or *label*—that is, *relabel*) this child (in its new position) **N**, and its sibling (its new parent's other child) **S**. (**S** was previously the sibling of **M**.) In the diagrams below, we will also use **P** for **N**'s new parent (**M**'s old parent), **SL** for **S**'s left child, and **SR** for **S**'s right child (**S** cannot be a leaf because if **M** and **C** were black, then **P**'s one subtree which included **M** counted two black-height and thus **P**'s other subtree which includes **S** must also count two black-height, which cannot be the case if **S** is a leaf node).

*Note:* In between some cases, we exchange the roles and labels of the nodes, but in each case, every label continues to represent the same node it represented at the beginning of the case. Any color shown in the diagram is either assumed in its case or implied by those assumptions. White represents an unknown color (either red or black).

We will find the sibling using this function:

```
struct node *sibling(struct node *n) { if (n == n-
>parent->left) return n->parent->right; else return
n->parent->left; }
```

*Note:* In order that the tree remains well-

defined, we need that every null leaf remains a leaf after all transformations (that it will not have any children). If the node we are deleting has a non-leaf (non-null) child  $N$ , it is easy to see that the property is satisfied. If, on the other hand,  $N$  would be a null leaf, it can be verified from the diagrams (or code) for all the cases that the property is satisfied as well.

We can perform the steps outlined above with the following code, where the function `replace_node` substitutes child into  $n$ 's place in the tree. For convenience, code in this section will assume that null leaves are represented by actual node objects rather than `NULL` (the code in the *Insertion* section works with either representation).

```
void delete_one_child(struct node *n) { /* * Precondition: n has at most one non-null child. */ struct node *child = is_leaf(n->right) ? n->left : n->right; replace_node(n, child); if (n->color == BLACK) { if (child->color == RED) child->color = BLACK; else delete_case1(child); } free(n); }
```

*Note:* If  $N$  is a null leaf and we do not want to represent null leaves as actual node objects, we can modify the algorithm by first calling `delete_case1()` on its parent (the node that we delete,  $n$  in the code above) and deleting it afterwards. We can do this because the parent is black, so it behaves in the same way as a null leaf (and is sometimes called a 'phantom' leaf). And we can safely delete it at the end as  $n$  will remain a leaf after all operations, as shown above.

If both  $N$  and its original parent are black, then deleting this original parent causes paths which proceed through  $N$  to have one fewer black node than paths that do not. As this violates property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes), the tree must be rebalanced. There are several cases to consider:

**Case 1:**  $N$  is the new root. In this case, we are done. We removed one black node from every path, and the new root is black, so the properties are preserved.

```
void delete_case1(struct node *n) { if (n->parent != NULL) delete_case2(n); }
```

*Note:* In cases 2, 5, and 6, we assume  $N$  is the left child of its parent  $P$ . If it is the right child, *left* and *right* should be reversed throughout these three cases. Again, the code examples take both cases into account.

```
void delete_case2(struct node *n) { struct node *s = sibling(n); if (s->color == RED) { n->parent->color
```

```
= RED; s->color = BLACK; if (n == n->parent->left) rotate_left(n->parent); else rotate_right(n->parent); } delete_case3(n); }
```

```
void delete_case3(struct node *n) { struct node *s = sibling(n); if ((n->parent->color == BLACK) && (s->color == BLACK) && (s->left->color == BLACK) && (s->right->color == BLACK)) { s->color = RED; delete_case1(n->parent); } else delete_case4(n); }
```

```
void delete_case4(struct node *n) { struct node *s = sibling(n); if ((n->parent->color == RED) && (s->color == BLACK) && (s->left->color == BLACK) && (s->right->color == BLACK)) { s->color = RED; n->parent->color = BLACK; } else delete_case5(n); }
```

```
void delete_case5(struct node *n) { struct node *s = sibling(n); if (s->color == BLACK) { /* this if statement is trivial, due to case 2 (even though case 2 changed the sibling to a sibling's child, the sibling's child can't be red, since no red parent can have a red child). */ /* the following statements just force the red to be on the left of the left of the parent, or right of the right, so case six will rotate correctly. */ if ((n == n->parent->left) && (s->right->color == BLACK) && (s->left->color == RED)) { /* this last test is trivial too due to cases 2-4. */ s->color = RED; s->left->color = BLACK; rotate_right(s); } else if ((n == n->parent->right) && (s->left->color == BLACK) && (s->right->color == RED)) { /* this last test is trivial too due to cases 2-4. */ s->color = RED; s->right->color = BLACK; rotate_left(s); } } delete_case6(n); }
```

```
void delete_case6(struct node *n) { struct node *s = sibling(n); s->color = n->parent->color; n->parent->color = BLACK; if (n == n->parent->left) { s->right->color = BLACK; rotate_left(n->parent); } else { s->left->color = BLACK; rotate_right(n->parent); } }
```

Again, the function calls all use tail recursion, so the algorithm is **in-place**. In the algorithm above, all cases are chained in order, except in delete case 3 where it can recurse to case 1 back to the parent node: this is the only case where an in-place implementation will effectively loop (after only one rotation in case 3).

Additionally, no tail recursion ever occurs on a child node, so the tail recursion loop can only move from a child back to its successive ancestors. No more than  $O(\log n)$  loops back to case 1 will occur (where  $n$  is the total number of nodes in the tree before deletion). If a rotation occurs in case 2 (which is the only possibility of rotation within the loop of cases 1–3), then the parent of the node  $N$  becomes red after the rotation and we will exit the loop. Therefore at most one rotation will occur within this loop. Since no more than two additional rotations will occur after exiting the loop, at most three rotations occur in total.

### 5.8.7 Proof of asymptotic bounds

A red black tree which contains  $n$  internal nodes has a height of  $O(\log(n))$ .

Definitions:

- $h(v)$  = height of subtree rooted at node  $v$
- $bh(v)$  = the number of black nodes (not counting  $v$  if it is black) from  $v$  to any leaf in the subtree (called the black-height).

**Lemma:** A subtree rooted at node  $v$  has at least  $2^{bh(v)} - 1$  internal nodes.

Proof of Lemma (by induction height):

Basis:  $h(v) = 0$

If  $v$  has a height of zero then it must be *null*, therefore  $bh(v) = 0$ . So:

$$2^{bh(v)} - 1 = 2^0 - 1 = 1 - 1 = 0$$

Inductive Step:  $v$  such that  $h(v) = k$ , has at least  $2^{bh(v)} - 1$  internal nodes implies that  $v'$  such that  $h(v') = k+1$  has at least  $2^{bh(v')} - 1$  internal nodes.

Since  $v'$  has  $h(v') > 0$  it is an internal node. As such it has two children each of which have a black-height of either  $bh(v')$  or  $bh(v') - 1$  (depending on whether the child is red or black, respectively). By the inductive hypothesis each child has at least  $2^{bh(v')-1} - 1$  internal nodes, so  $v'$  has at least:

$$2^{bh(v')-1} - 1 + 2^{bh(v')-1} - 1 + 1 = 2^{bh(v')} - 1$$

internal nodes.

Using this lemma we can now show that the height of the tree is logarithmic. Since at least half of the nodes on any path from the root to a leaf are black (property 4 of a red–black tree), the black-height of the root is at least  $h(\text{root})/2$ . By the lemma we get:

$$n \geq 2^{\frac{h(\text{root})}{2}} - 1 \leftrightarrow \log_2(n + 1) \geq \frac{h(\text{root})}{2} \leftrightarrow h(\text{root}) \leq 2 \log_2(n + 1)$$

Therefore the height of the root is  $O(\log n)$ .

### Insertion complexity

In the tree code there is only one loop where the node of the root of the red–black property that we wish to restore,  $x$ , can be moved up the tree by one level at each iteration.

Since the original height of the tree is  $O(\log n)$ , there are  $O(\log n)$  iterations. So overall the insert routine has  $O(\log n)$  complexity.

### 5.8.8 Parallel algorithms

Parallel algorithms for constructing red–black trees from sorted lists of items can run in constant time or  $O(\log \log n)$  time, depending on the computer model, if the number of processors available is asymptotically proportional to the number of items. Fast search, insertion, and deletion parallel algorithms are also known.<sup>[9]</sup>

### 5.8.9 See also

- Tree data structure
- Tree rotation
- Scapegoat tree
- Splay tree
- AVL tree
- B-tree (2-3 tree, 2-3-4 tree, B+ tree, B\*-tree, UB-tree)
- T-tree
- List of data structures

### 5.8.10 Notes

- [1] Cormen, Thomas H. (2001). *Introduction To Algorithms*. Charles E Leiserson, Ronald L Rivest, Clifford Stein. MIT Press. p. 273. ISBN 0262032937.
- [2] John Morris. “Red–Black Trees”.
- [3] Rudolf Bayer (1972). “Symmetric binary B-Trees: Data structure and maintenance algorithms”. *Acta Informatica* 1 (4): 290–306. doi:10.1007/BF00289509.
- [4] Leonidas J. Guibas and Robert Sedgewick (1978). “A Dichromatic Framework for Balanced Trees”. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. pp. 8–21. doi:10.1109/SFCS.1978.3.
- [5] Robert Sedgewick (2012). *Red–Black BSTs*. Coursera. A lot of people ask why did we use the name red–black. Well, we invented this data structure, this way of looking at balanced trees, at Xerox PARC which was the home of the personal computer and many other innovations that we live with today entering[sic] graphic user interfaces, ethernet and object-oriented programings[sic] and many other things. But one of the things that was invented there was laser printing and we were very excited to have nearby color laser printer that could print things out in color and out of the colors the red looked the best. So, that’s why we picked the color red to distinguish red links, the types of links, in three nodes. So, that’s an answer to the question for people that have been asking.
- [6] Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2009). “13”. *Introduction to Algorithms* (3rd ed.). MIT Press. pp. 308–309. ISBN 978-0-262-03384-8.
- [7] <http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>

- [8] <http://www.cs.princeton.edu/courses/archive/fall08/cos226/lectures/10BalancedTrees-2x2.pdf>
- [9] Park, Heejin; Park, Kunsoo (2001). “Parallel algorithms for red–black trees”. *Theoretical computer science* (Elsevier) **262** (1–2): 415–435. doi:10.1016/S0304-3975(00)00287-5. Our parallel algorithm for constructing a red–black tree from a sorted list of  $n$  items runs in  $O(1)$  time with  $n$  processors on the CRCW PRAM and runs in  $O(\log \log n)$  time with  $n / \log \log n$  processors on the EREW PRAM.

### 5.8.11 References

- Mathworld: Red–Black Tree
- San Diego State University: CS 660: Red–Black tree notes, by Roger Whitney
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7 . Chapter 13: Red–Black Trees, pp. 273–301.
- Pfaff, Ben (June 2004). “Performance Analysis of BSTs in System Software” (PDF). Stanford University.
- Okasaki, Chris. “Red–Black Trees in a Functional Setting” (PS).

### 5.8.12 External links

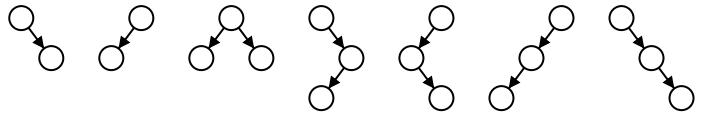
- A complete and working implementation in C
- Red–Black Tree Demonstration
- OCW MIT Lecture by Prof. Erik Demaine on Red Black Trees -
- Binary Search Tree Insertion Visualization on YouTube – Visualization of random and pre-sorted data insertions, in elementary binary search trees, and left-leaning red–black trees
- An intrusive red-black tree written in C++

## 5.9 AA tree

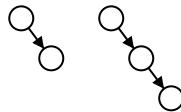
An **AA tree** in computer science is a form of balanced tree used for storing and retrieving ordered data efficiently. AA trees are named for Arne Andersson, their inventor.

AA trees are a variation of the red–black tree, a form of binary search tree which supports efficient addition and deletion of entries. Unlike red–black trees, red nodes on an AA tree can only be added as a right subchild. In other words, no red node can be a left sub-child. This results in

the simulation of a 2–3 tree instead of a 2–3–4 tree, which greatly simplifies the maintenance operations. The maintenance algorithms for a red–black tree need to consider seven different shapes to properly balance the tree:



An AA tree on the other hand only needs to consider two shapes due to the strict requirement that only right links can be red:



### 5.9.1 Balancing rotations

Whereas red–black trees require one bit of balancing metadata per node (the color), AA trees require  $O(\log(N))$  bits of metadata per node, in the form of an integer “level”. The following invariants hold for AA trees:

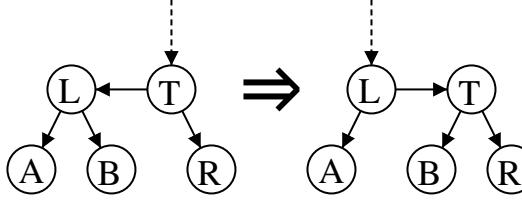
1. The level of every leaf node is one.
2. The level of every left child is exactly one less than that of its parent.
3. The level of every right child is equal to or one less than that of its parent.
4. The level of every right grandchild is strictly less than that of its grandparent.
5. Every node of level greater than one has two children.

A link where the child’s level is equal to that of its parent is called a *horizontal* link, and is analogous to a red link in the red–black tree. Individual right horizontal links are allowed, but consecutive ones are forbidden; all left horizontal links are forbidden. These are more restrictive constraints than the analogous ones on red–black trees, with the result that re-balancing an AA tree is procedurally much simpler than re-balancing a red–black tree.

Insertions and deletions may transiently cause an AA tree to become unbalanced (that is, to violate the AA tree invariants). Only two distinct operations are needed for restoring balance: “skew” and “split”. Skew is a right rotation to replace a subtree containing a left horizontal link with one containing a right horizontal link instead. Split is a left rotation and level increase to replace a subtree containing two or more consecutive right horizontal links with one containing two fewer consecutive right horizontal links. Implementation of balance-preserving insertion and deletion is simplified by relying on the skew and split

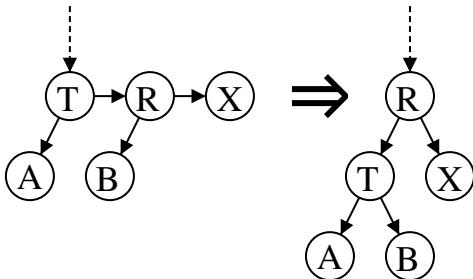
operations to modify the tree only if needed, instead of making their callers decide whether to skew or split.

**function** skew **is input:** T, a node representing an AA tree that needs to be rebalanced. **output:** Another node representing the rebalanced AA tree. **if** nil(T) **then return** Nil **else if** nil(left(T)) **then return** T **else if** level(left(T)) == level(T) **then Swap the pointers of horizontal left links.** L = left(T) left(T) := right(L) right(L) := T **return** L **else return** T **end if** **end function**



Skew:

**function** split **is input:** T, a node representing an AA tree that needs to be rebalanced. **output:** Another node representing the rebalanced AA tree. **if** nil(T) **then return** Nil **else if** nil(right(T)) **or** nil(right(right(T))) **then return** T **else if** level(T) == level(right(right(T))) **then We have two horizontal right links. Take the middle node, elevate it, and return it.** R = right(T) right(T) := left(R) left(R) := T level(R) := level(R) + 1 **return** R **else return** T **end if** **end function**



Split:

## 5.9.2 Insertion

Insertion begins with the normal binary tree search and insertion procedure. Then, as the call stack unwinds (assuming a recursive implementation of the search), it's easy to check the validity of the tree and perform any rotations as necessary. If a horizontal left link arises, a skew will be performed, and if two horizontal right links arise, a split will be performed, possibly incrementing the level of the new root node of the current subtree. Note, in the code as given above, the increment of level(T). This makes it necessary to continue checking the validity of the tree as the modifications bubble up from the leaves.

**function** insert **is input:** X, the value to be inserted, and T, the root of the tree to insert it into. **output:** A balanced version T including X. *Do the normal binary tree insertion procedure. Set the result of the recursive call to the correct child in case a new node was created or the root of the subtree changes.* **if** nil(T) **then Create a new**

*leaf node with X. return node(X, 1, Nil, Nil) **else if** X < value(T) **then** left(T) := insert(X, left(T)) **else if** X > value(T) **then** right(T) := insert(X, right(T)) **end if** Note that the case of X == value(T) is unspecified. As given, an insert will have no effect. The implementor may desire different behavior. Perform skew and then split. The conditionals that determine whether or not a rotation will occur or not are inside of the procedures, as given above. T := skew(T) T := split(T) **return** T **end function***

## 5.9.3 Deletion

As in most balanced binary trees, the deletion of an internal node can be turned into the deletion of a leaf node by swapping the internal node with either its closest predecessor or successor, depending on which are in the tree or on the implementor's whims. Retrieving a predecessor is simply a matter of following one left link and then all of the remaining right links. Similarly, the successor can be found by going right once and left until a null pointer is found. Because of the AA property of all nodes of level greater than one having two children, the successor or predecessor node will be in level 1, making their removal trivial.

To re-balance a tree, there are a few approaches. The one described by Andersson in his [original paper](#) is the simplest, and it is described here, although actual implementations may opt for a more optimized approach. After a removal, the first step to maintaining tree validity is to lower the level of any nodes whose children are two levels below them, or who are missing children. Then, the entire level must be skewed and split. This approach was favored, because when laid down conceptually, it has three easily understood separate steps:

1. Decrease the level, if appropriate.
2. Skew the level.
3. Split the level.

However, we have to skew and split the entire level this time instead of just a node, complicating our code.

**function** delete **is input:** X, the value to delete, and T, the root of the tree from which it should be deleted. **output:** T, balanced, without the value X. **if** nil(T) **then return** T **else if** X > value(T) **then** right(T) := delete(X, right(T)) **else if** X < value(T) **then** left(T) := delete(X, left(T)) **else If we're a leaf, easy, otherwise reduce to leaf case.** **if** leaf(T) **then return** Nil **else if** nil(left(T)) **then** L := successor(T) right(T) := delete(value(L), right(T)) value(T) := value(L) **else** L := predecessor(T) left(T) := delete(value(L), left(T)) value(T) := value(L) **end if** **end if** *Rebalance the tree. Decrease the level of all nodes in this level if necessary, and then skew and split all nodes in the new level.* T := decrease\_level(T) T := skew(T) right(T) := skew(right(T)) **if**

```

not nil(right(T)) right(right(T)) := skew(right(right(T)))
end if T := split(T) right(T) := split(right(T)) return T
end function function decrease_level is input: T, a
tree for which we want to remove links that skip lev-
els. output: T with its level decreased. should_be
= min(level(left(T)), level(right(T))) + 1 if should_be
< level(T) then level(T) := should_be if should_be <
level(right(T)) then level(right(T)) := should_be end if
end if return T end function

```

A good example of deletion by this algorithm is present in the [Andersson paper](#).

#### 5.9.4 Performance

The performance of an AA tree is equivalent to the performance of a red-black tree. While an AA tree makes more rotations than a red-black tree, the simpler algorithms tend to be faster, and all of this balances out to result in similar performance. A red-black tree is more consistent in its performance than an AA tree, but an AA tree tends to be flatter, which results in slightly faster search times.<sup>[1]</sup>

#### 5.9.5 See also

- Red-black tree
- B-tree
- AVL tree
- Scapegoat tree

#### 5.9.6 References

[1] "A Disquisition on The Performance Behavior of Binary Search Tree Data Structures (pages 67-75)" (PDF).

#### 5.9.7 External links

- A. Andersson. Balanced search trees made simple
- A. Andersson. A note on searching in a binary search tree
- AA-Tree Applet by Kubo Kovac
- BSTlib - Open source AA tree library for C by tri-jezdci
- AA Visual 2007 1.5 - OpenSource Delphi program for educating AA tree structures
- Thorough tutorial Julianne Walker with lots of code, including a practical implementation
- Object Oriented implementation with tests

- A Disquisition on The Performance Behavior of Binary Search Tree Data Structures (pages 67-75) - Comparison of AA trees, red-black trees, treaps, skip lists, and radix trees
- An example C implementation
- An Objective-C implementation

## 5.10 Scapegoat tree

In computer science, a **scapegoat tree** is a self-balancing binary search tree, invented by Arne Andersson<sup>[1]</sup> and again by Igal Galperin and Ronald L. Rivest.<sup>[2]</sup> It provides worst-case  $O(\log n)$  lookup time, and  $O(\log n)$  amortized insertion and deletion time.

Unlike most other self-balancing binary search trees that provide worst case  $O(\log n)$  lookup time, scapegoat trees have no additional per-node memory overhead compared to a regular **binary search tree**: a node stores only a key and two pointers to the child nodes. This makes scapegoat trees easier to implement and, due to **data structure alignment**, can reduce node overhead by up to one-third.

#### 5.10.1 Theory

A binary search tree is said to be weight-balanced if half the nodes are on the left of the root, and half on the right. An  $\alpha$ -weight-balanced node is defined as meeting a relaxed weight balance criterion:

$$\text{size(left)} \leq \alpha * \text{size(node)} \quad \text{size(right)} \leq \alpha * \text{size(node)}$$

Where size can be defined recursively as:

```
function size(node) if node = nil return 0 else return
size(node->left) + size(node->right) + 1 end
```

An  $\alpha$  of 1 therefore would describe a linked list as balanced, whereas an  $\alpha$  of 0.5 would only match **almost complete binary trees**.

A binary search tree that is  $\alpha$ -weight-balanced must also be  **$\alpha$ -height-balanced**, that is

$$\text{height(tree)} \leq \log_2/\alpha(\text{NodeCount}) + 1$$

Scapegoat trees are not guaranteed to keep  $\alpha$ -weight-balance at all times, but are always loosely  $\alpha$ -height-balanced in that

$$\text{height(scapegoat tree)} \leq \log_2/\alpha(\text{NodeCount}) + 1$$

This makes scapegoat trees similar to **red-black trees** in that they both have restrictions on their height. They differ greatly though in their implementations of determining where the rotations (or in the case of scapegoat trees, rebalances) take place. Whereas red-black trees store additional 'color' information in each node to determine the location, scapegoat trees find a **scapegoat** which isn't  $\alpha$ -weight-balanced to perform the rebalance operation on.

This is loosely similar to AVL trees, in that the actual rotations depend on 'balances' of nodes, but the means of determining the balance differs greatly. Since AVL trees check the balance value on every insertion/deletion, it is typically stored in each node; scapegoat trees are able to calculate it only as needed, which is only when a scapegoat needs to be found.

Unlike most other self-balancing search trees, scapegoat trees are entirely flexible as to their balancing. They support any  $\alpha$  such that  $0.5 < \alpha < 1$ . A high  $\alpha$  value results in fewer balances, making insertion quicker but lookups and deletions slower, and vice versa for a low  $\alpha$ . Therefore in practical applications, an  $\alpha$  can be chosen depending on how frequently these actions should be performed.

## 5.10.2 Operations

### Insertion

Insertion is implemented with the same basic ideas as an unbalanced binary search tree, however with a few significant changes.

When finding the insertion point, the depth of the new node must also be recorded. This is implemented via a simple counter that gets incremented during each iteration of the lookup, effectively counting the number of edges between the root and the inserted node. If this node violates the  $\alpha$ -height-balance property (defined above), a rebalance is required.

To rebalance, an entire subtree rooted at a **scapegoat** undergoes a balancing operation. The scapegoat is defined as being an ancestor of the inserted node which isn't  $\alpha$ -weight-balanced. There will always be at least one such ancestor. Rebalancing any of them will restore the  $\alpha$ -height-balanced property.

One way of finding a scapegoat, is to climb from the new node back up to the root and select the first node that isn't  $\alpha$ -weight-balanced.

Climbing back up to the root requires  $O(\log n)$  storage space, usually allocated on the stack, or parent pointers. This can actually be avoided by pointing each child at its parent as you go down, and repairing on the walk back up.

To determine whether a potential node is a viable scapegoat, we need to check its  $\alpha$ -weight-balanced property. To do this we can go back to the definition:

$$\text{size(left)} \leq \alpha * \text{size(node)} \quad \text{size(right)} \leq \alpha * \text{size(node)}$$

However a large optimisation can be made by realising that we already know two of the three sizes, leaving only the third having to be calculated.

Consider the following example to demonstrate this. Assuming that we're climbing back up to the root:

$$\text{size(parent)} = \text{size(node)} + \text{size(sibling)} + 1$$

But as:

$$\text{size(inserted node)} = 1.$$

The case is trivialized down to:

$$\text{size}[x+1] = \text{size}[x] + \text{size}(sibling) + 1$$

Where  $x$  = this node,  $x + 1$  = parent and  $\text{size}(sibling)$  is the only function call actually required.

Once the scapegoat is found, the subtree rooted at the scapegoat is completely rebuilt to be perfectly balanced.<sup>[2]</sup> This can be done in  $O(n)$  time by traversing the nodes of the subtree to find their values in sorted order and recursively choosing the median as the root of the subtree.

As rebalance operations take  $O(n)$  time (dependent on the number of nodes of the subtree), insertion has a worst-case performance of  $O(n)$  time. However, because these worst-case scenarios are spread out, insertion takes  $O(\log n)$  amortized time.

**Sketch of proof for cost of insertion** Define the Imbalance of a node  $v$  to be the absolute value of the difference in size between its left node and right node minus 1, or 0, whichever is greater. In other words:

$$I(v) = \max(|\text{left}(v) - \text{right}(v)| - 1, 0)$$

Immediately after rebuilding a subtree rooted at  $v$ ,  $I(v) = 0$ .

**Lemma:** Immediately before rebuilding the subtree rooted at  $v$ ,

$$I(v) = \Omega(|v|)$$

( $\Omega$  is Big O Notation.)

Proof of lemma:

Let  $v_0$  be the root of a subtree immediately after rebuilding.  $h(v_0) = \log(|v_0| + 1)$ . If there are  $\Omega(|v_0|)$  degenerate insertions (that is, where each inserted node increases the height by 1), then

$$I(v) = \Omega(|v_0|) ,$$

$$h(v) = h(v_0) + \Omega(|v_0|) \text{ and}$$

$$\log(|v|) \leq \log(|v_0| + 1) + 1 .$$

Since  $I(v) = \Omega(|v|)$  before rebuilding, there were  $\Omega(|v|)$  insertions into the subtree rooted at  $v$  that did not result in rebuilding. Each of these insertions can be performed in  $O(\log n)$  time. The final insertion that causes rebuilding costs  $O(|v|)$ . Using aggregate analysis it becomes clear that the amortized cost of an insertion is  $O(\log n)$ :

$$\frac{\Omega(|v|)O(\log n) + O(|v|)}{\Omega(|v|)} = O(\log n)$$

### Deletion

Scapegoat trees are unusual in that deletion is easier than insertion. To enable deletion, scapegoat trees need to store an additional value with the tree data structure. This property, which we will call **MaxNodeCount** sim-

ply represents the highest achieved NodeCount. It is set to NodeCount whenever the entire tree is rebalanced, and after insertion is set to  $\max(\text{MaxNodeCount}, \text{NodeCount})$ .

To perform a deletion, we simply remove the node as you would in a simple binary search tree, but if

$\text{NodeCount} \leq \alpha * \text{MaxNodeCount}$

then we rebalance the entire tree about the root, remembering to set MaxNodeCount to NodeCount.

This gives deletion its worst-case performance of  $O(n)$  time; however, it is amortized to  $O(\log n)$  average time.

**Sketch of proof for cost of deletion** Suppose the scapegoat tree has  $n$  elements and has just been rebuilt (in other words, it is a complete binary tree). At most  $n/2 - 1$  deletions can be performed before the tree must be rebuilt. Each of these deletions take  $O(\log n)$  time (the amount of time to search for the element and flag it as deleted). The  $n/2$  deletion causes the tree to be rebuilt and takes  $O(\log n) + O(n)$  (or just  $O(n)$ ) time. Using aggregate analysis it becomes clear that the amortized cost of a deletion is  $O(\log n)$ :

$$\sum_{i=1}^{\frac{n}{2}} \frac{O(\log n) + O(n)}{\frac{n}{2}} = \frac{\frac{n}{2}O(\log n) + O(n)}{\frac{n}{2}} = O(\log n)$$

### Lookup

Lookup is not modified from a standard binary search tree, and has a worst-case time of  $O(\log n)$ . This is in contrast to splay trees which have a worst-case time of  $O(n)$ . The reduced node memory overhead compared to other self-balancing binary search trees can further improve **locality of reference** and caching.

### 5.10.3 See also

- Splay tree
- Trees
- Tree rotation
- AVL tree
- B-tree
- T-tree
- List of data structures

### 5.10.4 References

- [1] Andersson, Arne (1989). *Improving partial rebuilding by using simple balance criteria*. Proc. Workshop on Algorithms and Data Structures. *Journal of Algorithms* (Springer-Verlag): 393–402. doi:10.1007/3-540-51542-9\_33.

- [2] Galperin, Igal; Rivest, Ronald L. (1993). “Scapegoat trees”. *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*: 165–174.

### 5.10.5 External links

- Scapegoat Tree Applet by Kubo Kovac
- Scapegoat Trees: Galperin and Rivest’s paper describing scapegoat trees
- On Consulting a Set of Experts and Searching (full version paper)
- Open Data Structures - Chapter 8 - Scapegoat Trees

## 5.11 Splay tree

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in  $O(\log n)$  amortized time. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985.<sup>[1]</sup>

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

### 5.11.1 Advantages

Good performance for a splay tree depends on the fact that it is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. The worst-case height—though unlikely—is  $O(n)$ , with the average being  $O(\log n)$ . Having frequently used nodes near the root is an advantage for many practical applications (also see **Locality of reference**), and is particularly useful for implementing **caches** and **garbage collection** algorithms.

Advantages include:

- Comparable performance—**average-case performance** is as efficient as other trees.<sup>[2]</sup>
- Small memory footprint—splay trees do not need to store any bookkeeping data.

- Possibility of creating a persistent data structure version of splay trees—which allows access to both the previous and new versions after an update. This can be useful in **functional programming**, and requires amortized  $O(\log n)$  space per update.
- Working well with nodes containing identical keys—contrary to other types of self-balancing trees. Even with identical keys, performance remains amortized  $O(\log n)$ . All tree operations preserve the order of the identical nodes within the tree, which is a property similar to **stable sorting algorithms**. A carefully designed *find* operation can return the leftmost or rightmost node of a given key.

### 5.11.2 Disadvantages

The most significant disadvantage of splay trees is that the height of a splay tree can be linear. For example, this will be the case after accessing all  $n$  elements in non-decreasing order. Since the height of a tree corresponds to the worst-case access time, this means that the actual cost of an operation can be high. However the **amortized** access cost of this worst case is logarithmic,  $O(\log n)$ . Also, the expected access cost can be reduced to  $O(\log n)$  by using a randomized variant.<sup>[3]</sup>

The representation of splay trees can change even when they are accessed in a 'read-only' manner (i.e. by *find* operations). This complicates the use of such splay trees in a multi-threaded environment. Specifically, extra management is needed if multiple threads are allowed to perform *find* operations concurrently. This also makes them unsuitable for general use in purely functional programming, although they can be used in limited ways to implement priority queues even there.

### 5.11.3 Operations

#### Splaying

When a node  $x$  is accessed, a splay operation is performed on  $x$  to move it to the root. To perform a splay operation we carry out a sequence of *splay steps*, each of which moves  $x$  closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

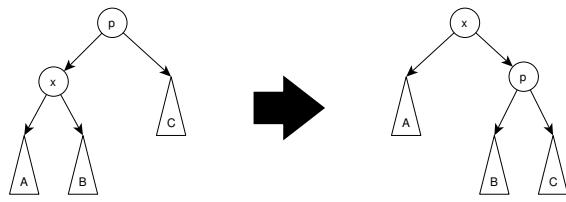
Each particular step depends on three factors:

- Whether  $x$  is the left or right child of its parent node,  $p$ ,
- whether  $p$  is the root or not, and if not
- whether  $p$  is the left or right child of its parent,  $g$  (the *grandparent* of  $x$ ).

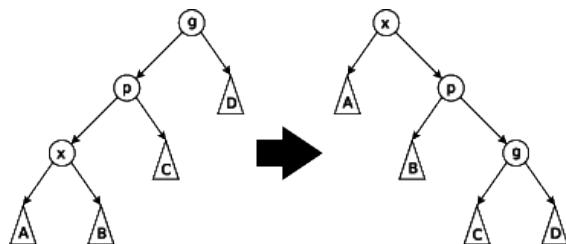
It is important to remember to set  $gg$  (the *great-grandparent* of  $x$ ) to now point to  $x$  after any splay operation. If  $gg$  is null, then  $x$  obviously is now the root and must be updated as such.

There are three types of splay steps, each of which has a left- and right-handed case. For the sake of brevity, only one of these two is shown for each type. These three types are:

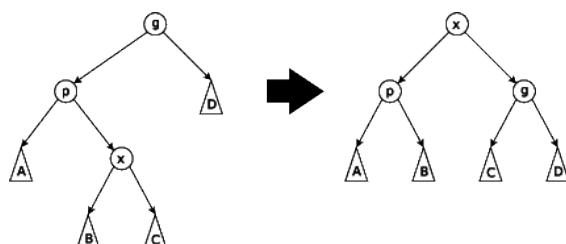
**Zig step:** this step is done when  $p$  is the root. The tree is rotated on the edge between  $x$  and  $p$ . Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when  $x$  has odd depth at the beginning of the operation.



**Zig-zig step:** this step is done when  $p$  is not the root and  $x$  and  $p$  are either both right children or are both left children. The picture below shows the case where  $x$  and  $p$  are both left children. The tree is rotated on the edge joining  $p$  with its parent  $g$ , then rotated on the edge joining  $x$  with  $p$ . Note that zig-zig steps are the only thing that differentiate splay trees from the *rotate to root* method introduced by Allen and Munro<sup>[4]</sup> prior to the introduction of splay trees.



**Zig-zag step:** this step is done when  $p$  is not the root and  $x$  is a right child and  $p$  is a left child or vice versa. The tree is rotated on the edge between  $p$  and  $x$ , and then rotated on the resulting edge between  $x$  and  $g$ .



## Join

Given two trees S and T such that all elements of S are smaller than the elements of T, the following steps can be used to join them to a single tree:

- Splay the largest item in S. Now this item is in the root of S and has a null right child.
- Set the right child of the new root to T.

## Split

Given a tree and an element  $x$ , return two new trees: one containing all elements less than or equal to  $x$  and the other containing all elements greater than  $x$ . This can be done in the following way:

- Splay  $x$ . Now it is in the root so the tree to its left contains all elements smaller than  $x$  and the tree to its right contains all elements larger than  $x$ .
- Split the right subtree from the rest of the tree.

## Insertion

To insert a value  $x$  into a splay tree:

- Insert  $x$  as with a normal **binary search tree**.
- Splay the newly inserted node  $x$  to the top of the tree.

## ALTERNATIVE:

- Use the split operation to split the tree at the value of  $x$  to two sub-trees: S and T.
- Create a new tree in which  $x$  is the root, S is its left sub-tree and T its right sub-tree.

## Deletion

To delete a node  $x$ , use the same method as with a binary search tree: if  $x$  has two children, swap its value with that of either the rightmost node of its left sub tree (its in-order predecessor) or the leftmost node of its right subtree (its in-order successor). Then remove that node instead. In this way, deletion is reduced to the problem of removing a node with 0 or 1 children. Unlike a binary search tree, in a splay tree after deletion, we splay the parent of the removed node to the top of the tree.

## ALTERNATIVE:

- The node to be deleted is first splayed, i.e. brought to the root of the tree and then deleted. leaves the tree with two sub trees.
- The two sub-trees are then joined using a “join” operation.

## 5.11.4 Implementation and variants

Splaying, as mentioned above, is performed during a second, bottom-up pass over the access path of a node. It is possible to record the access path during the first pass for use during the second, but that requires extra space during the access operation. Another alternative is to keep a parent pointer in every node, which avoids the need for extra space during access operations but may reduce overall time efficiency because of the need to update those pointers.<sup>[1]</sup>

Another method which can be used is based on the argument that we can restructure the tree on our way down the access path instead of making a second pass. This top-down splaying routine uses three sets of nodes - left tree, right tree and middle tree. The first two contain all items of original tree known to be less than or greater than current item respectively. The middle tree consists of the sub-tree rooted at the current node. These three sets are updated down the access path while keeping the splay operations in check. Another method, semisplaying, modifies the zig-zig case to reduce the amount of restructuring done in all operations.<sup>[1][5]</sup>

Below there is an implementation of splay trees in C++, which uses pointers to represent each node on the tree. This implementation is based on bottom-up splaying version and uses the second method of deletion on a splay tree. Also, unlike the above definition, this C++ version does *not* splay the tree on finds - it only splays on insertions and deletions.

```
#include <functional> #ifndef SPLAY_TREE #define SPLAY_TREE
template< typename T, typename Comp = std::less< T > > class splay_tree {
 private:
 Comp comp;
 unsigned long p_size;
 struct node {
 node *left, *right, *parent;
 T key;
 node(const T& init = T()) : left(0), right(0), parent(0), key(init) { }
 ~node() { if(left) delete left; if(right) delete right; if(parent) delete parent; }
 *root;
 void left_rotate(node *x) { node *y = x->right; if(y) { x->right = y->left; if(y->left) y->left->parent = x; y->parent = x->parent; }
 if(!x->parent) root = y; else if(x == x->parent->left) x->parent->left = y; else x->parent->right = y; if(y) y->left = x; x->parent = y; }
 void right_rotate(node *x) { node *y = x->left; if(y) { x->left = y->right; if(y->right) y->right->parent = x; y->parent = x->parent; }
 if(!x->parent) root = y; else if(x == x->parent->left) x->parent->left = y; else x->parent->right = y; if(y) y->right = x; x->parent = y; }
 void splay(node *x) { while(x->parent) { if(!x->parent->parent) { if(x->parent->left == x) right_rotate(x->parent); else left_rotate(x->parent); }
 else if(x->parent->left == x && x->parent->parent->left == x->parent) { right_rotate(x->parent->parent); right_rotate(x->parent); }
 else if(x->parent->right == x && x->parent->parent->right == x->parent) { left_rotate(x->parent->parent); left_rotate(x->parent); }
 else if(x->parent->left == x && x->parent->parent->right == x->parent) { }
 } }
 };
};
```

```

right_rotate(x->parent); left_rotate(x->parent); } else
{ left_rotate(x->parent); right_rotate(x->parent); } }
} void replace(node *u, node *v) { if(!u->parent) root
= v; else if(u == u->parent->left) u->parent->left = v;
else u->parent->right = v; if(v) v->parent = u->parent;
} node* subtree_minimum(node *u) { while(u->left)
u = u->left; return u; } node* subtree_maximum(node
*u) { while(u->right) u = u->right; return u; } public:
splay_tree() : root(0), p_size(0) { } void insert(const
T &key) { node *z = root; node *p = 0; while(z) {
p = z; if(comp(z->key, key)) z = z->right; else z =
z->left; } z = new node(key); z->parent = p; if(!p)
root = z; else if(comp(p->key, z->key)) p->right =
z; else p->left = z; splay(z); p_size++; } node* find(
const T &key) { node *z = root; while(z) { if(comp(
z->key, key)) z = z->right; else if(comp(key, z->key)
) z = z->left; } return z; } void erase(const T &key) { node
*z = find(key); if(!z) return; splay(z); if(!z->left)
replace(z, z->right); else if(!z->right) replace(z,
z->left); else { node *y = subtree_minimum(z->right);
if(y->parent != z) { replace(y, y->right); y->right =
z->right; y->right->parent = y; } replace(z, y); y->left =
z->left; y->left->parent = y; } delete z; p_size--; } const T&
minimum() { return subtree_minimum(root)->key; } const T&
maximum() { return subtree_maximum(root)->key; } bool empty(
) const { return root == 0; } unsigned long size() const { return
p_size; } }; #endif // SPLAY_TREE

```

### 5.11.5 Analysis

A simple **amortized** analysis of static splay trees can be carried out using the **potential method**. Define:

- $\text{size}(r)$  - the number of nodes in the sub-tree rooted at node  $r$  (including  $r$ ).
- $\text{rank}(r) = \log_2(\text{size}(r))$ .
- $\Phi$  = the sum of the ranks of all the nodes in the tree.

$\Phi$  will tend to be high for poorly balanced trees and low for well-balanced trees.

To apply the **potential method**, we first calculate  $\Delta\Phi$  - the change in the potential caused by a splay operation. We check each case separately. Denote by  $\text{rank}'$  the rank function after the operation.  $x$ ,  $p$  and  $g$  are the nodes affected by the rotation operation (see figures above).

**Zig step:**

$$\begin{aligned}
\Delta\Phi &= \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x) \\
&\quad [\text{since only } p \text{ and } x \text{ change ranks}] \\
&= \text{rank}'(p) - \text{rank}(x) \quad [\text{since } \text{rank}'(x) = \text{rank}(p)] \\
&\leq \text{rank}'(x) - \text{rank}(x) \quad [\text{since } \text{rank}'(p) < \text{rank}'(x)]
\end{aligned}$$

**Zig-Zig step:**

$$\begin{aligned}
\Delta\Phi &= \text{rank}'(g) - \text{rank}(g) + \text{rank}'(p) - \text{rank}(p) + \\
&\quad \text{rank}'(x) - \text{rank}(x) \\
&\leq \text{rank}'(g) + \text{rank}'(p) - \text{rank}(p) - \text{rank}(x) \quad [\text{since } \\
&\quad \text{rank}'(x) = \text{rank}(g)] \\
&\leq \text{rank}'(g) + \text{rank}'(x) - 2 \text{rank}(x) \quad [\text{since } \\
&\quad \text{rank}(x) < \text{rank}(p) \text{ and } \text{rank}'(x) > \text{rank}'(p)] \\
&\leq 3(\text{rank}'(x) - \text{rank}(x)) - 2 \quad [\text{due to the convexity} \\
&\quad \text{of the log function}]
\end{aligned}$$

**Zig-Zag step:**

$$\begin{aligned}
\Delta\Phi &= \text{rank}'(g) - \text{rank}(g) + \text{rank}'(p) - \text{rank}(p) + \\
&\quad \text{rank}'(x) - \text{rank}(x) \\
&\leq \text{rank}'(g) + \text{rank}'(p) - 2 \text{rank}(x) \quad [\text{since } \\
&\quad \text{rank}'(x) = \text{rank}(g) \text{ and } \text{rank}(x) < \text{rank}(p)] \\
&\leq 2(\text{rank}'(x) - \text{rank}(x)) - 2 \quad [\text{due to the convexity} \\
&\quad \text{of the log function}]
\end{aligned}$$

The amortized cost of any operation is  $\Delta\Phi$  plus the actual cost. The actual cost of any zig-zig or zig-zag operation is 2 since there are two rotations to make. Hence:

$$\begin{aligned}
\text{amortized-cost} &= \text{cost} + \Delta\Phi \\
&\leq 3(\text{rank}'(x) - \text{rank}(x))
\end{aligned}$$

When summed over the entire splay operation, this **telescopes** to  $3(\text{rank}(\text{root}) - \text{rank}(x))$  which is  $O(\log n)$ . The Zig operation adds an amortized cost of 1, but there's at most one such operation.

So now we know that the total *amortized* time for a sequence of  $m$  operations is:

$$T_{\text{amortized}}(m) = O(m \log n)$$

To go from the amortized time to the actual time, we must add the decrease in potential from the initial state before any operation is done ( $\Phi_i$ ) to the final state after all operations are completed ( $\Phi_f$ ).

$$\Phi_i - \Phi_f = \sum_x \text{rank}_i(x) - \text{rank}_f(x) = O(n \log n)$$

where the last inequality comes from the fact that for every node  $x$ , the minimum rank is 0 and the maximum rank is  $\log(n)$ .

Now we can finally bound the actual time:

$$T_{\text{actual}}(m) = O(m \log n + n \log n)$$

### Weighted analysis

The above analysis can be generalized in the following way.

- Assign to each node  $r$  a weight  $w(r)$ .
- Define  $\text{size}(r) =$  the sum of weights of nodes in the sub-tree rooted at node  $r$  (including  $r$ ).
- Define  $\text{rank}(r)$  and  $\Phi$  exactly as above.

The same analysis applies and the amortized cost of a splaying operation is again:

$$\text{rank}(\text{root}) - \text{rank}(x) = O(\log W - \log w(x)) = O(\log \frac{W}{w(x)})$$

where  $W$  is the sum of all weights.

The decrease from the initial to the final potential is bounded by:

$$\Phi_i - \Phi_f \leq \sum_{x \in \text{tree}} \log \frac{W}{w(x)}$$

since the maximum size of any single node is  $W$  and the minimum is  $w(x)$ .

Hence the actual time is bounded by:

$$O\left(\sum_{x \in \text{sequence}} \left(\log \frac{W}{w(x)}\right) + \sum_{x \in \text{tree}} \left(\log \frac{W}{w(x)}\right)\right)$$

### 5.11.6 Performance theorems

There are several theorems and conjectures regarding the worst-case runtime for performing a sequence  $S$  of  $m$  accesses in a splay tree containing  $n$  elements.

**Balance Theorem** The cost of performing the sequence  $S$  is  $O[m \log n + n \log n]$  (**Proof:** take a constant weight, e.g.  $w(x)=1$  for every node  $x$ . Then  $W=n$ ). This theorem implies that splay trees perform as well as static balanced binary search trees on sequences of at least  $n$  accesses.<sup>[1]</sup>

**Static Optimality Theorem** Let  $q_x$  be the number of times element  $x$  is accessed in  $S$ . The cost of performing  $S$  is  $O\left[m + \sum_{x \in \text{tree}} q_x \log \frac{m}{q_x}\right]$  (**Proof:** let  $w(x) = q_x$ . Then  $W = m$ ). This theorem implies that splay trees perform as well as an optimum static binary search tree on sequences of at least  $n$  accesses. They spend less time on the more frequent items.<sup>[1]</sup>

**Static Finger Theorem** Assume that the items are numbered from 1 through  $n$  in ascending order. Let  $f$  be any fixed element (the 'finger'). Then the cost of performing  $S$  is  $O\left[m + n \log n + \sum_{x \in \text{sequence}} \log(|x - f| + 1)\right]$

(**Proof:** let  $w(x) = 1/(|x - f| + 1)^2$ . Then  $W=O(1)$ . The net potential drop is  $O(n \log n)$  since the weight of any item is at least  $1/n^2$ ).<sup>[1]</sup>

**Dynamic Finger Theorem** Assume that the 'finger' for each step accessing an element  $y$  is the element accessed in the previous step,  $x$ . The cost of performing  $S$  is  $O\left[m + n + \sum_{x, y \in \text{sequence}}^m \log(|y - x| + 1)\right]$   
[6][7]

**Working Set Theorem** At any time during the sequence, let  $t(x)$  be the number of distinct elements accessed before the previous time element  $x$  was accessed. The cost of performing  $S$  is  $O\left[m + n \log n + \sum_{x \in \text{sequence}} \log(t(x) + 1)\right]$  (**Proof:** let  $w(x) = 1/(t(x) + 1)^2$ . Note that here the weights change during the sequence. However, the sequence of weights is still a permutation of  $1, 1/4, 1/9, \dots, 1/n^2$ . So as before  $W=O(1)$ . The net potential drop is  $O(n \log n)$ . This theorem is equivalent to splay trees having **key-independent optimality**.<sup>[1]</sup>

**Scanning Theorem** Also known as the **Sequential Access Theorem** or the **Queue theorem**. Accessing the  $n$  elements of a splay tree in symmetric order takes  $O(n)$  time, regardless of the initial structure of the splay tree.<sup>[8]</sup> The tightest upper bound proven so far is  $4.5n$ .<sup>[9]</sup>

### 5.11.7 Dynamic optimality conjecture

Main article: [Optimal binary search tree](#)

In addition to the proven performance guarantees for splay trees there is an unproven conjecture of great interest from the original Sleator and Tarjan paper. This conjecture is known as the *dynamic optimality conjecture* and it basically claims that splay trees perform as well as any other binary search tree algorithm up to a constant factor.

**Dynamic Optimality Conjecture:**<sup>[1]</sup> Let  $A$  be any binary search tree algorithm that accesses an element  $x$  by traversing the path from the root to  $x$  at a cost of  $d(x) + 1$ , and that between accesses can make any rotations in the tree at a cost of 1 per rotation. Let  $A(S)$  be the cost for  $A$  to perform the sequence  $S$  of accesses. Then the cost for a splay tree to perform the same accesses is  $O[n + A(S)]$ .

There are several corollaries of the dynamic optimality conjecture that remain unproven:

**Traversal Conjecture:**<sup>[1]</sup> Let  $T_1$  and  $T_2$  be two splay trees containing the same elements. Let  $S$  be the sequence obtained by visiting the elements in  $T_2$  in preorder (i.e., depth first search order). The total cost of performing the sequence  $S$  of accesses on  $T_1$  is  $O(n)$ .

**Deque Conjecture:**<sup>[8][10][11]</sup> Let  $S$  be a sequence of  $m$  double-ended queue operations (push, pop, inject, eject). Then the cost of performing  $S$  on a splay tree is  $O(m + n)$ .

**Split Conjecture:**<sup>[5]</sup> Let  $S$  be any permutation of the elements of the splay tree. Then the cost of deleting the elements in the order  $S$  is  $O(n)$ .

## 5.11.8 Variants

In order to reduce the number of restructuring operations, it is possible to replace the splaying with *semi-splaying*, in which an element is splayed only halfway towards the root.<sup>[1]</sup>

Another way to reduce restructuring is to do full splaying, but only in some of the access operations - only when the access path is longer than a threshold, or only in the first  $m$  access operations.<sup>[1]</sup>

## 5.11.9 See also

- Finger tree
- Link/cut tree
- Scapegoat tree
- Zipper (data structure)
- Trees
- Tree rotation
- AVL tree
- B-tree
- T-tree
- List of data structures
- Iacono's working set structure
- Geometry of binary search trees
- Splaysort, a sorting algorithm using splay trees

## 5.11.10 Notes

- [1] Sleator, Daniel D.; Tarjan, Robert E. (1985), "Self-Adjusting Binary Search Trees" (PDF), *Journal of the ACM (Association for Computing Machinery)* **32** (3): 652–686, doi:10.1145/3828.3835
- [2] Goodrich, Michael; Tamassia, Roberto; Goldwasser, Michael. *Data Structures and Algorithms in Java* (in English) (6 ed.). John Wiley & Sons, Inc. p. 506. ISBN 978-1-118-77133-4. The surprising thing about splaying is that it allows us to guarantee a logarithmic amortized running time, for insertions, deletions, and searches.
- [3] "Randomized Splay Trees: Theoretical and Experimental Results" (PDF). Retrieved 31 May 2011.
- [4] Allen, Brian; and Munro, Ian (1978), "Self-organizing search trees", *Journal of the ACM* **25** (4): 526–535, doi:10.1145/322092.322094
- [5] Lucas, Joan M. (1991), "On the Competitiveness of Splay Trees: Relations to the Union-Find Problem", *Online Algorithms, Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) Series in Discrete Mathematics and Theoretical Computer Science* Vol. 7: 95–124
- [6] Cole, Richard; Mishra, Bud; Schmidt, Jeanette; and Siegel, Alan (2000), "On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting log n-Block Sequences", *SIAM Journal on Computing* **30**: 1–43, doi:10.1137/s0097539797326988
- [7] Cole, Richard (2000), "On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof", *SIAM Journal on Computing* **30**: 44–85, doi:10.1137/S009753979732699X
- [8] Tarjan, Robert E. (1985), "Sequential access in splay trees takes linear time", *Combinatorica* **5** (4): 367–378, doi:10.1007/BF02579253
- [9] Elmasry, Amr (2004), "On the sequential access theorem and Deque conjecture for splay trees", *Theoretical Computer Science* **314** (3): 459–466, doi:10.1016/j.tcs.2004.01.019
- [10] Pettie, Seth (2008), "Splay Trees, Davenport-Schinzel Sequences, and the Deque Conjecture", *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms* **0707**: 1115–1124, arXiv:0707.2160, Bibcode:2007arXiv0707.2160P
- [11] Sundar, Rajamani (1992), "On the Deque conjecture for the splay algorithm", *Combinatorica* **12** (1): 95–124, doi:10.1007/BF01191208

## 5.11.11 References

- Knuth, Donald. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Page 478 of section 6.2.3.

### 5.11.12 External links

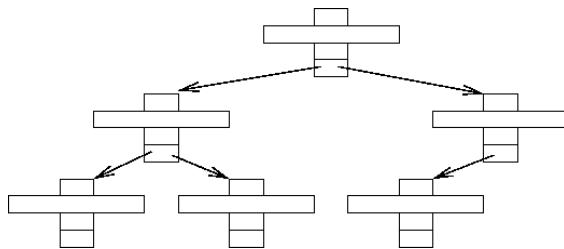
- NIST's Dictionary of Algorithms and Data Structures: Splay Tree
- Implementations in C and Java (by Daniel Sleator)
- Pointers to splay tree visualizations
- Fast and efficient implementation of Splay trees
- Top-Down Splay Tree Java implementation
- Zipper Trees
- splay tree video

Rao, Jun; Kenneth A. Ross (1999). "Cache conscious indexing for decision-support in main memory". *Proceedings of the 25th International Conference on Very Large Databases (VLDB 1999)*. Morgan Kaufmann. pp. 78–89.

Kim, Kyungwha; Junho Shim; Ig-hoon Lee (2007). "Cache conscious trees: How do they perform on contemporary commodity microprocessors?". *Proceedings of the 5th International Conference on Computational Science and Its Applications (ICCSA 2007)*. Springer. pp. 189–200. doi:10.1007/978-3-540-74472-6\_15.

The main reason seems to be that the traditional assumption of memory references having uniform cost is no longer valid given the current speed gap between cache access and main memory access.

## 5.12 T-tree



An example T-tree.

In computer science a **T-tree** is a type of binary tree data structure that is used by main-memory databases, such as Datablitz, EXtremeDB, MySQL Cluster, Oracle TimesTen and MobileLite.

A T-tree is a **balanced** index tree data structure optimized for cases where both the index and the actual data are fully kept in memory, just as a **B-tree** is an index structure optimized for storage on block oriented secondary storage devices like hard disks. T-trees seek to gain the performance benefits of in-memory tree structures such as **AVL** trees while avoiding the large storage space overhead which is common to them.

T-trees do not keep copies of the indexed data fields within the index tree nodes themselves. Instead, they take advantage of the fact that the actual data is always in main memory together with the index so that they just contain pointers to the actual data fields.

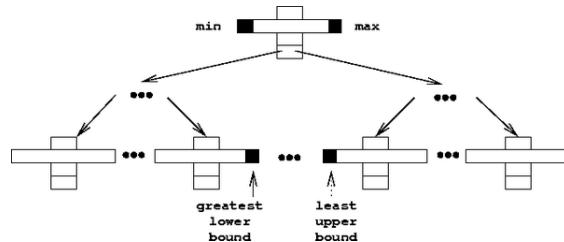
The 'T' in T-tree refers to the shape of the node data structures in the original paper that first described this type of index.<sup>[1]</sup>

### 5.12.1 Performance

Although T-trees seem to be widely used for main-memory databases, recent research indicates that they actually do not perform better than B-trees on modern hardware:

### 5.12.2 Node structures

A T-tree node usually consists of pointers to the parent node, the left and right child node, an ordered array of data pointers and some extra control data. Nodes with two subtrees are called *internal nodes*, nodes without subtrees are called *leaf nodes* and nodes with only one subtree are named *half-leaf nodes*. A node is called the *bounding node* for a value if the value is between the node's current minimum and maximum value, inclusively.



Bound values.

For each internal node, leaf or half leaf nodes exist that contain the predecessor of its smallest data value (called the *greatest lower bound*) and one that contains the successor of its largest data value (called the *least upper bound*). Leaf and half-leaf nodes can contain any number of data elements from one to the maximum size of the data array. Internal nodes keep their occupancy between predefined minimum and maximum numbers of elements

### 5.12.3 Algorithms

#### Search

- Search starts at the root node
- If the current node is the bounding node for the search value then search its data array. Search fails if the value is not found in the data array.

- If the search value is less than the minimum value of the current node then continue search in its left subtree. Search fails if there is no left subtree.
- If the search value is greater than the maximum value of the current node then continue search in its right subtree. Search fails if there is no right subtree.

## Insertion

- Search for a bounding node for the new value. If such a node exist then
  - check whether there is still space in its data array, if so then insert the new value and finish
  - if no space is available then remove the minimum value from the node's data array and insert the new value. Now proceed to the node holding the greatest lower bound for the node that the new value was inserted to. If the removed minimum value still fits in there then add it as the new maximum value of the node, else create a new right subnode for this node.
- If no bounding node was found then insert the value into the last node searched if it still fits into it. In this case the new value will either become the new minimum or maximum value. If the value doesn't fit anymore then create a new left or right subtree.

If a new node was added then the tree might need to be rebalanced, as described below.

## Deletion

- Search for bounding node of the value to be deleted. If no bounding node is found then finish.
- If the bounding node does not contain the value then finish.
- delete the value from the node's data array

Now we have to distinguish by node type:

- Internal node:

If the node's data array now has less than the minimum number of elements then move the greatest lower bound value of this node to its data value. Proceed with one of the following two steps for the half leaf or leaf node the value was removed from.

- Leaf node:

If this was the only element in the data array then delete the node. Rebalance the tree if needed.

- Half leaf node:

If the node's data array can be merged with its leaf's data array without overflow then do so and remove the leaf node. Rebalance the tree if needed.

## Rotation and balancing

A T-tree is implemented on top of an underlying self-balancing binary search tree. Specifically, Lehman and Carey's article describes a T-tree balanced like an **AVL tree**: it becomes out of balance when a node's child trees differ in height by at least two levels. This can happen after an insertion or deletion of a node. After an insertion or deletion, the tree is scanned from the leaf to the root. If an imbalance is found, one **tree rotation** or pair of rotations is performed, which is guaranteed to balance the whole tree.

When the rotation results in an internal node having fewer than the minimum number of items, items from the node's new child(ren) are moved into the internal node.

### 5.12.4 Notes

### 5.12.5 See also

- Tree (graph theory)
- Tree (set theory)
- Tree structure
- Exponential tree

## Other trees

- B-tree (2-3 tree, 2-3-4 tree, B+ tree, B\*-tree, UB-tree)
- DSW algorithm
- Dancing tree
- Fusion tree
- kd-tree
- Octree
- Quadtree
- R-tree
- Radix tree
- T-tree
- T-pyramid
- Top Trees

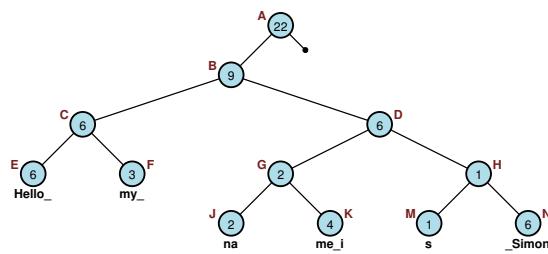
## 5.12.6 References

- [1] Tobin J. Lehman and Michael J. Carey, A Study of Index Structures for Main Memory Database Management Systems. VLDB 1986

## 5.12.7 External links

- Oracle TimesTen FAQ entry on index mentioning T-Trees
- Oracle Whitepaper: Oracle TimesTen Products and Technologies
- DataBlitz presentation mentioning T-Trees
- An Open-source T\*-tree Library

## 5.13 Rope



A simple rope built on the string of “Hello\_my\_name\_is\_Simon”.

In computer programming a **rope**, or **cord**, is a data structure composed of smaller strings that is used for efficiently storing and manipulating a very long string. For example, a text editing program may use a rope to represent the text being edited, so that operations such as insertion, deletion, and random access can be done efficiently.<sup>[1]</sup>

## 5.13.1 Description

A rope is a **binary tree** having leaf nodes that contain a short string. Each node has a weight value equal to the length of its string plus the sum of all leaf nodes' weight in its left subtree, namely the weight of a node is the total string length in its left subtree for a non-leaf node, or the string length of itself for a leaf node. Thus a node with two children divides the whole string into two parts: the left subtree stores the first part of the string. The right subtree stores the second part and its weight is the sum of the left child's weight and the length of its contained string.

The binary tree can be seen as several levels of nodes. The bottom level contains all the nodes that contain a string. Higher levels have fewer and fewer nodes. The top level consists of a single “root” node. The rope is

built by putting the nodes with short strings in the bottom level, then attaching a random half of the nodes to parent nodes in the next level.

## 5.13.2 Operations

In the following definitions,  $N$  is the length of the rope.

### Index

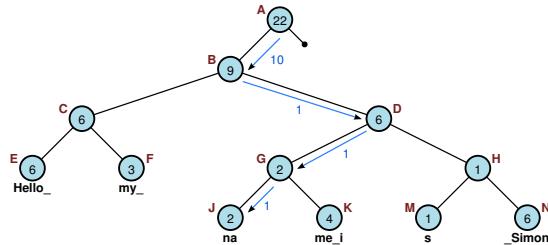


Figure 2.1: Example of index lookup on a rope.

*Definition:* Index( $i$ ): return the character at position  $i$

*Time complexity:*  $O(\log N)$

To retrieve the  $i$ -th character, we begin a recursive search from the root node:

```
// Note: Assumes 1-based indexing. function index(RopeNode node, integer i)
 if node.weight < i then return index(node.right, i - node.weight)
 else if exists(node.left) then return index(node.left, i)
 else return node.string[i]
```

For example, to find the character at  $i=10$  in Figure 2.1 shown on the right, start at the root node (A), find that 22 is greater than 10 and there is a left child, so go to the left child (B). 9 is less than 10, so subtract 9 from 10 (leaving  $i=1$ ) and go to the right child (D). Then because 6 is greater than 1 and there's a left child, go to the left child (G). 2 is greater than 1 and there's a left child, so go to the left child again (J). Finally 2 is greater than 1 but there is no left child, so the character at index 1 of the short string “na”, is the answer.

### Concat

*Definition:* Concat( $S_1, S_2$ ): concatenate two ropes,  $S_1$  and  $S_2$ , into a single rope.

*Time complexity:*  $O(1)$  (or  $O(\log N)$  time to compute the root weight)

A concatenation can be performed simply by creating a new root node with  $left = S_1$  and  $right = S_2$ , which is constant time. The weight of the parent node is set to the

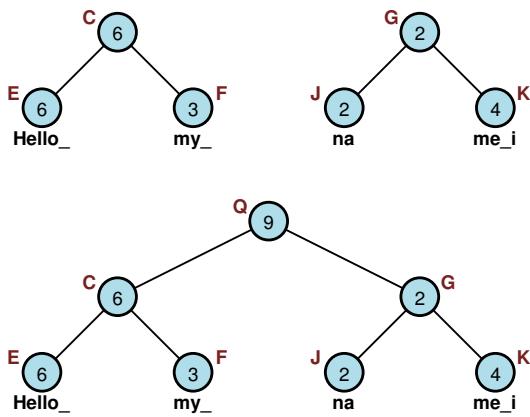


Figure 2.2: Concatenating two child ropes into a single rope.

length of the left child  $S1$ , which would take  $O(\log N)$  time, if the tree is balanced.

As most rope operations require balanced trees, the tree may need to be re-balanced after concatenation.

## Split

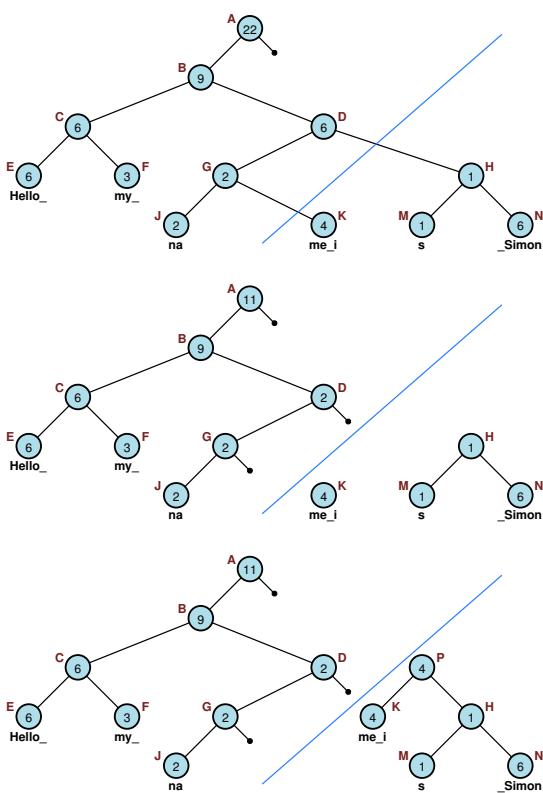


Figure 2.3: Splitting a rope in half.

**Definition:**  $\text{Split}(i, S)$ : split the string  $S$  into two new strings  $S_1$  and  $S_2$ ,  $S_1 = C_1, \dots, C_i$  and  $S_2 = C_{i+1}, \dots, C_m$ .

**Time complexity:**  $O(\log N)$

There are two cases that must be dealt with:

1. The split point is at the end of a string (i.e. after the last character of a leaf node)
2. The split point is in the middle of a string.

The second case reduces to the first by splitting the string at the split point to create two new leaf nodes, then creating a new node that is the parent of the two component strings.

For example, to split the 22-character rope pictured in Figure 2.3 into two equal component ropes of length 11, query the 12th character to locate the node **K** at the bottom level. Remove the link between **K** and the right child of **G**. Go to the parent **G** and subtract the weight of **K** from the weight of **G**. Travel up the tree and remove any right links, subtracting the weight of **K** from these nodes (only node **D**, in this case). Finally, build up the newly orphaned nodes **K** and **H** by concatenating them together and creating a new parent **P** with weight equal to the length of the left node **K**.

As most rope operations require balanced trees, the tree may need to be re-balanced after splitting.

## Insert

**Definition:**  $\text{Insert}(i, S')$ : insert the string  $S'$  beginning at position  $i$  in the string  $s$ , to form a new string  $C_1, \dots, C_i, S', C_{i+1}, \dots, C_m$ .

**Time complexity:**  $O(\log N)$ .

This operation can be done by a  $\text{Split}()$  and two  $\text{Concat}()$  operations. The cost is the sum of the three.

## Delete

**Definition:**  $\text{Delete}(i, j)$ : delete the substring  $C_i, \dots, C_{i+j-1}$ , from  $s$  to form a new string  $C_1, \dots, C_{i-1}, C_{i+j}, \dots, C_m$ .

**Time complexity:**  $O(\log N)$ .

This operation can be done by two  $\text{Split}()$  and one  $\text{Concat}()$  operation. First, split the rope in three, divided by  $i$ -th and  $i+j$ -th character respectively, which extracts the string to delete in a separate node. Then concatenate the other two nodes.

## Report

**Definition:**  $\text{Report}(i, j)$ : output the string  $C_i, \dots, C_{i+j-1}$ .

**Time complexity:**  $O(j + \log N)$

To report the string  $C_i, \dots, C_{i+j-1}$ , find the node  $u$  that contains  $C_i$  and  $\text{weight}(u) \geq j$ , and then traverse  $T$  starting at node  $u$ . Output  $C_i, \dots, C_{i+j-1}$  by doing an in-order traversal of  $T$  starting at node  $u$ .

### 5.13.3 Comparison with monolithic arrays

Advantages:

- Ropes enable much faster insertion and deletion of text than monolithic string arrays, on which operations have time complexity  $O(n)$ .
- Ropes don't require  $O(n)$  extra memory when operated upon (arrays need that for copying operations)
- Ropes don't require large contiguous memory spaces.
- If only nondestructive versions of operations are used, rope is a **persistent data structure**. For the text editing program example, this leads to an easy support for multiple **undo** levels.

Disadvantages:

- Greater overall space usage when not being operated on, mainly to store parent nodes. There is a trade-off between how much of the total memory is such overhead and how long pieces of data are being processed as strings; note that the strings in example figures above are unrealistically short for modern architectures. The overhead is always  $O(n)$ , but the constant can be made arbitrarily small.
- Increase in time to manage the extra storage
- Increased complexity of source code; greater risk for bugs

This table compares the *algorithmic* characteristics of string and rope implementations, not their “raw speed”. Array-based strings have smaller overhead, so (for example) concatenation and split operations are faster on small datasets. However, when array-based strings are used for longer strings, time complexity and memory usage for insertion and deletion of characters become unacceptably large. A rope data structure, on the other hand, has stable performance regardless of data size. Moreover, the space complexity for ropes and arrays are both  $O(n)$ . In summary, ropes are better suited when the data is large and frequently modified.

### 5.13.4 See also

- The **Cedar** programming environment, which used ropes “almost since its inception”<sup>[1]</sup>

- The **Model T enfilade**, a similar data structure from the early 1970s.
- **Gap buffer**, a data structure commonly used in text editors that allows efficient insertion and deletion operations clustered near the same location

### 5.13.5 References

- [1] Boehm, Hans-J; Atkinson, Russ; and Plass, Michael (December 1995). “Ropes: an Alternative to Strings” (PDF). *Software—Practice & Experience* (New York, NY, USA: John Wiley & Sons, Inc.) 25 (12): 1315–1330. doi:10.1002/spe.4380251203.

### 5.13.6 External links

- SGI's implementation of ropes for C++
- “C cords” implementation of ropes within the Boehm Garbage Collector library
- libstdc++ support for ropes
- Ropes for Java
- Ropes for OCaml
- ropes for Common Lisp
- pyropes for Python

## 5.14 Top Trees

A **top tree** is a **data structure** based on a binary tree for unrooted dynamic trees that is used mainly for various path-related operations. It allows simple **divide-and-conquer** algorithms. It has since been augmented to maintain dynamically various properties of a **tree** such as diameter, center and median.

A top tree  $\mathcal{R}$  is defined for an *underlying tree*  $\mathcal{T}$  and a set  $\partial\mathcal{T}$  of at most two vertices called as **External Boundary Vertices**

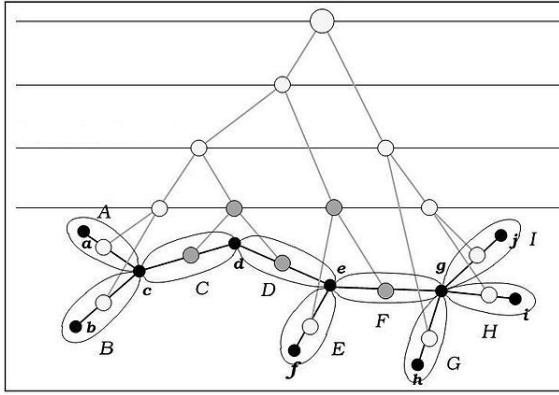
### 5.14.1 Glossary

#### Boundary Node

See **Boundary Vertex**

#### Boundary Vertex

A vertex in a connected subtree is a *Boundary Vertex* if it is connected to a vertex outside the subtree by an edge.



An image depicting a top tree built on an underlying tree (black nodes). A tree divided into edge clusters and the complete top-tree for it. Filled nodes in the top-tree are path-clusters, while small circle nodes are leaf-clusters. The big circle node is the root. Capital letters denote clusters, non-capital letters are nodes.

**External Boundary Vertices** Up to a pair of vertices in the top tree  $\mathfrak{R}$  can be called as External Boundary Vertices, they can be thought of as Boundary Vertices of the cluster which represents the entire top tree.

### Cluster

A *cluster* is a connected subtree with at most two Boundary Vertices. The set of Boundary Vertices of a given cluster  $\mathcal{C}$  is denoted as  $\partial\mathcal{C}$ . With each cluster  $\mathcal{C}$  the user may associate some meta information  $I(\mathcal{C})$ , and give methods to maintain it under the various internal operations.

**Path Cluster** If  $\pi(\mathcal{C})$  contains at least one edge then  $\mathcal{C}$  is called a *Path Cluster*.

**Point Cluster** See Leaf Cluster

**Leaf Cluster** If  $\pi(\mathcal{C})$  does not contain any edge i.e.  $\mathcal{C}$  has only one Boundary Vertex then  $\mathcal{C}$  is called a *Leaf Cluster*.

**Edge Cluster** A Cluster containing a single edge is called an *Edge Cluster*.

**Leaf Edge Cluster** A Leaf in the original Cluster is represented by a Cluster with just a single Boundary Vertex and is called a *Leaf Edge Cluster*.

**Path Edge Cluster** Edge Clusters with two Boundary Nodes are called *Path Edge Cluster*.

### Internal Node

A node in  $\mathcal{C} \setminus \partial\mathcal{C}$  is called an *Internal Node* of  $\mathcal{C}$ .

### Cluster Path

The path between the Boundary Vertices of  $\mathcal{C}$  is called the *cluster path* of  $\mathcal{C}$  and it is denoted by  $\pi(\mathcal{C})$ .

### Mergeable Clusters

Two Clusters  $\mathcal{A}$  and  $\mathcal{B}$  are *Mergeable* if  $\mathcal{A} \cap \mathcal{B}$  is a singleton set (they have exactly one node in common) and  $\mathcal{A} \cup \mathcal{B}$  is a Cluster.

## 5.14.2 Introduction

*Top trees* are used for maintaining a Dynamic forest (set of trees) under **link** and **cut** operations.

The basic idea is to maintain a balanced Binary tree  $\mathfrak{R}$  of logarithmic height in the number of nodes in the original tree  $\mathcal{T}$  ( i.e. in  $\mathcal{O}(\log n)$  time ) ; the **top tree** essentially represents the recursive subdivision of the original tree  $\mathcal{T}$  into **clusters**.

In general the tree  $\mathcal{T}$  may have weight on its edges.

There is a one to one correspondence with the edges of the original tree  $\mathcal{T}$  and the leaf nodes of the top tree  $\mathfrak{R}$  and each internal node of  $\mathfrak{R}$  represents a cluster that is formed due to the union of the clusters that are its children.

The top tree data structure can be initialized in  $\mathcal{O}(n)$  time.

Therefore the top tree  $\mathfrak{R}$  over  $(\mathcal{T}, \partial\mathcal{T})$  is a binary tree such that

- The nodes of  $\mathfrak{R}$  are clusters of  $(\mathcal{T}, \partial\mathcal{T})$ ;
- The leaves of  $\mathfrak{R}$  are the edges of  $\mathcal{T}$ ;
- Sibling clusters are neighbours in the sense that they intersect in a single vertex, and then their parent cluster is their union.
- Root of  $\mathfrak{R}$  is the tree  $\mathcal{T}$  itself, with a set of at most two External Boundary Vertices.

A tree with a single vertex has an empty top tree, and one with just an edge is just a single node.

These trees are freely **augmentable** allowing the user a wide variety of flexibility and productivity without going into the details of the internal workings of the data structure, something which is also referred to as the *Black Box*.

### 5.14.3 Dynamic Operations

The following three are the user allowable Forest Updates.

- **Link(v, w):** Where  $v$  and  $w$  are vertices in different trees  $\mathcal{T}_1$  and  $\mathcal{T}_2$ . It returns a single top tree representing  $\mathfrak{R}_v \cup \mathfrak{R}_w \cup (v, w)$
- **Cut(v, w):** Removes the edge  $(v, w)$  from a tree  $\mathcal{T}$  with top tree  $\mathfrak{R}$ , thereby turning it into two trees  $\mathcal{T}_v$  and  $\mathcal{T}_w$  and returning two top trees  $\mathfrak{R}_v$  and  $\mathfrak{R}_w$ .
- **Expose(S):** Is called as a subroutine for implementing most of the queries on a top tree.  $S$  contains at most 2 vertices. It makes original external vertices to be normal vertices and makes vertices from  $S$  the new External Boundary Vertices of the top tree. If  $S$  is nonempty it returns the new Root cluster  $\mathcal{C}$  with  $\partial\mathcal{C} = S$ . **Expose(v,w)** fails if the vertices are from different trees.

### 5.14.4 Internal Operations

The Forest updates are all carried out by a sequence of at most  $\mathcal{O}(\log n)$  Internal Operations, the sequence of which is computed in further  $\mathcal{O}(\log n)$  time. It may happen that during a tree update, a leaf cluster may change to a path cluster and the converse. Updates to top tree are done exclusively by these internal operations.

The  $I(\mathcal{C})$  is updated by calling a user defined function associated with each internal operation.

- **Merge ( $\mathcal{A}, \mathcal{B}$ ):** Here  $\mathcal{A}$  and  $\mathcal{B}$  are *Mergeable Clusters*, it returns  $\mathcal{C}$  as the parent cluster of  $\mathcal{A}$  and  $\mathcal{B}$  and with boundary vertices as the boundary vertices of  $\mathcal{A} \cup \mathcal{B}$ . Computes  $I(\mathcal{C})$  using  $I(\mathcal{A})$  and  $I(\mathcal{B})$ .
- **Split ( $\mathcal{C}$ ):** Here  $\mathcal{C}$  is the root cluster  $\mathcal{A} \cup \mathcal{B}$ . It updates  $I(\mathcal{A})$  and  $I(\mathcal{B})$  using  $I(\mathcal{C})$  and than it deletes the cluster  $\mathcal{C}$  from  $\mathfrak{R}$ .

Split is usually implemented using **Clean ( $\mathcal{C}$ )** method which calls user method for updates of  $I(\mathcal{A})$  and  $I(\mathcal{B})$  using  $I(\mathcal{C})$  and updates  $I(\mathcal{C})$  such that it's known there is no pending update needed in its children. Than the  $\mathcal{C}$  is discarded without calling user defined functions. **Clean** is often required for queries without need to **Split**. If Split does not use Clean subroutine, and Clean is required, its effect could be achieved with overhead by combining **Merge** and **Split**.

The next two functions are analogous to the above two and are used for base clusters.

- **Create ( $v, w$ ):** Creates a cluster  $\mathcal{C}$  for the edge  $(v, w)$ . Sets  $\partial\mathcal{C} = \partial(v, w)$ .  $I(\mathcal{C})$  is computed from scratch.

- **Eradicate ( $\mathcal{C}$ ):**  $\mathcal{C}$  is the edge cluster  $(v, w)$ . User defined function is called to process  $I(\mathcal{C})$  and than the cluster  $\mathcal{C}$  is deleted from the top tree.

### 5.14.5 Non local search

User can define **Choose ( $\mathcal{C}$ )**: operation which for a root (nonleaf) cluster selects one of its child clusters. The top tree blackbox provides **Search ( $\mathcal{C}$ )**: routine, which organizes **Choose** queries and reorganization of the top tree (using the Internal operations) such that it locates the only edge in intersection of all selected clusters. Sometimes the search should be limited to a path. There is a variant of nonlocal search for such purposes. If there are two external boundary vertices in the root cluster  $\mathcal{C}$ , the edge is searched only on the path  $\pi(\mathcal{C})$ . It is sufficient to do following modification: If only one of root cluster children is path cluster, it is selected by default without calling the **Choose** operation.

#### Examples of non local search

Finding  $i$ -th edge on longer path from  $v$  to  $w$  could be done by  $\mathcal{C} = \text{Expose}\{v, w\}$  followed by **Search( $\mathcal{C}$ )** with appropriate **Choose**. To implement the **Choose** we use global variable representing  $v$  and global variable representing  $i$ . **Choose** selects the cluster  $\mathcal{A}$  with  $v \in \partial\mathcal{A}$  iff length of  $\pi(\mathcal{A})$  is at least  $i$ . To support the operation the length must be maintained in the  $I$ .

Similar task could be formulated for graph with edges with nonunit lengths. In that case the distance could address an edge or a vertex between two edges. We could define **Choose** such that the edge leading to the vertex is returned in the latter case. There could be defined update increasing all edge lengths along a path by a constant. In such scenario these updates are done in constant time just in root cluster. **Clean** is required to distribute the delayed update to the children. The **Clean** should be called before the **Search** is invoked. To maintain length in  $I$  would in that case require to maintain unitlength in  $I$  as well.

Finding center of tree containing vertex  $v$  could be done by finding either bicenter edge or edge with center as one endpoint. The edge could be found by  $\mathcal{C} = \text{Expose}\{v\}$  followed by **Search( $\mathcal{C}$ )** with appropriate **Choose**. The choose selects between children  $\mathcal{A}, \mathcal{B}$  with  $a \in \partial\mathcal{A} \cap \partial\mathcal{B}$  the child with higher maxdistance ( $a$ ). To support the operation the maximal distance in the cluster subtree from a boundary vertex should be maintained in the  $I$ . That requires maintenance of the cluster path length as well.

### 5.14.6 Interesting Results and Applications

A number of interesting applications originally implemented by other methods have been easily implemented using the top tree's interface. Some of them include

- ([SLEATOR AND TARJAN 1983]). We can maintain a dynamic collection of weighted trees in  $\mathcal{O}(\log n)$  time per link and cut, supporting queries about the maximum edge weight between any two vertices in  $\mathcal{O}(\log n)$  time.
  - Proof outline: It involves maintaining at each node the maximum weight ( $\text{max\_wt}$ ) on its cluster path, if it is a point cluster then  $\text{max\_wt}(\mathcal{C})$  is initialised as  $-\infty$ . When a cluster is a union of two clusters then it is the maximum value of the two merged clusters. If we have to find the max wt between  $v$  and  $w$  then we do  $\mathcal{C} = \text{Expose}(v, w)$ , and report  $\text{max\_wt}(\mathcal{C})$ .
- ([SLEATOR AND TARJAN 1983]). In the scenario of the above application we can also add a common weight  $x$  to all edges on a given path  $v \cdot \cdot \cdot w$  in  $\mathcal{O}(\log n)$  time.
  - Proof outline: We introduce a weight called  $\text{extra}(\mathcal{C})$  to be added to all the edges in  $\pi(\mathcal{C})$ . Which is maintained appropriately ;  $\text{split}(\mathcal{C})$  requires that, for each path child  $\mathcal{A}$  of  $\mathcal{C}$ , we set  $\text{max\_wt}(\mathcal{A}) := \text{max\_wt}(\mathcal{A}) + \text{extra}(\mathcal{C})$  and  $\text{extra}(\mathcal{A}) := \text{extra}(\mathcal{A}) + \text{extra}(\mathcal{C})$ . For  $\mathcal{C} := \text{join}(\mathcal{A}, \mathcal{B})$ , we set  $\text{max\_wt}(\mathcal{C}) := \max\{\text{max\_wt}(\mathcal{A}), \text{max\_wt}(\mathcal{B})\}$  and  $\text{extra}(\mathcal{C}) := 0$ . Finally, to find the maximum weight on the path  $v \cdot \cdot \cdot w$ , we set  $\mathcal{C} := \text{Expose}(v, w)$  and return  $\text{max\_wt}(\mathcal{C})$ .
- ([GOLDBERG ET AL. 1991]). We can ask for the maximum weight in the underlying tree containing a given vertex  $v$  in  $\mathcal{O}(\log n)$  time.
  - Proof outline: This requires maintaining additional information about the maximum weight non cluster path edge in a cluster under the Merge and Split operations.
- The distance between two vertices  $v$  and  $w$  can be found in  $\mathcal{O}(\log n)$  time as  $\text{length}(\text{Expose}(v, w))$ .
  - Proof outline: We will maintain the length  $\text{length}(\mathcal{C})$  of the cluster path. The length is maintained as the maximum weight except that, if  $\mathcal{C}$  is created by a  $\text{join}(\text{Merge})$ ,  $\text{length}(\mathcal{C})$  is the sum of lengths stored with its path children.

- Queries regarding diameter of a tree and its subsequent maintenance takes  $\mathcal{O}(\log n)$  time.
- The Center and Median can be maintained under  $\text{Link}(\text{Merge})$  and  $\text{Cut}(\text{Split})$  operations and queried by non local search in  $\mathcal{O}(\log n)$  time.
- The graph could be maintained allowing to update the edge set and ask queries on edge 2-connectivity. Amortized complexity of updates is  $\mathcal{O}(\log^4 n)$ . Queries could be implemented even faster. The algorithm is not trivial,  $I(\mathcal{C})$  uses  $\Theta(\log^2 n)$  space ([HOLM, LICHTENBERG, THORUP 2000]).
- The graph could be maintained allowing to update the edge set and ask queries on vertex 2-connectivity. Amortized complexity of updates is  $\mathcal{O}(\log^5 n)$ . Queries could be implemented even faster. The algorithm is not trivial,  $I(\mathcal{C})$  uses  $\Theta(\log^2 n)$  space ([HOLM, LICHTENBERG, THORUP 2001]).
- Top trees can be used to compress trees in a way that is never much worse than **DAG** compression, but may be exponentially better. [1]

### 5.14.7 Implementation

Top trees have been implemented in a variety of ways, some of them include implementation using a *Multilevel Partition* (Top-trees and dynamic graph algorithms Jacob Holm and Kristian de Lichtenberg. Technical Report), and even by using Sleator-Tarjan s-t trees (typically with amortized time bounds), Frederickson's Topology Trees (with worst case time bounds) (Alstrup et al. Maintaining Information in Fully Dynamic Trees with Top Trees).

Amortized implementations are more simple, and with small multiplicative factors in time complexity. On the contrary the worst case implementations allow speeding up queries by switching off unneeded info updates during the query (implemented by persistence techniques). After the query is answered the original state of the top tree is used and the query version is discarded.

#### Using Multilevel Partitioning

Any partitioning of clusters of a tree  $\mathcal{T}$  can be represented by a Cluster Partition Tree  $\text{CPT}(\mathcal{T})$ , by replacing each cluster in the tree  $\mathcal{T}$  by an edge. If we use a strategy  $P$  for partitioning  $\mathcal{T}$  then the CPT would be  $\text{CPTP } \mathcal{T}$ . This is done recursively till only one edge remains.

We would notice that all the nodes of the corresponding top tree  $\mathfrak{R}$  are uniquely mapped into the edges of this multilevel partition. There may be some edges in the multilevel partition that do not correspond to any node in the

top tree, these are the edges which represent only a single child in the level below it, i.e. a simple cluster. Only the edges that correspond to composite clusters correspond to nodes in the top tree  $\mathfrak{N}$ .

A partitioning strategy is important while we partition the Tree  $\mathcal{T}$  into clusters. Only a careful strategy ensures that we end up in an  $\mathcal{O}(\log n)$  height Multilevel Partition ( and therefore the top tree).

- The number of edges in subsequent levels should decrease by a constant factor.
- If a lower level is changed by an update then we should be able to update the one immediately above it using at most a constant number of insertions and deletions.

The above partitioning strategy ensures the maintenance of the top tree in  $\mathcal{O}(\log n)$  time.

#### 5.14.8 References

- Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup, *Maintaining information in fully dynamic trees with top trees*, ACM Transactions on Algorithms (TALG), Vol. 1 (2005), 243–264, doi:10.1145/1103963.1103966
- Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup, *Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity*, Journal of the ACM (JACM), Vol. 48 Issue 4(July 2001), 723–760, doi:10.1145/502090.502095
- Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4 . Section 2.3: Trees, pp. 308–423.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7 . Section 10.4: Representing rooted trees, pp. 214–217. Chapters 12–14 (Binary Search Trees, Red-Black Trees, Augmenting Data Structures), pp. 253–320.

[1] Tree Compression with Top Trees. BILLE, GOERTZ, LANDAU, WEIMANN 2013 arXiv:1304.5702 [cs.DS]

#### 5.14.9 External links

- Maintaining Information in Fully Dynamic Trees with Top Trees. Alstrup et al
- Self Adjusting Top Trees. Tarjan and Werneck
- Self-Adjusting Top Trees. Tarjan and Werneck, Proc. 16th SoDA, 2005

## 5.15 Tango tree

A **Tango tree** is a type of **binary search tree** proposed by Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu in 2004.<sup>[1]</sup>

It is an **online** binary search tree that achieves an  $O(\log \log n)$  competitive ratio relative to the optimal **offline** binary search tree, while only using  $O(\log \log n)$  additional bits of memory per node. This improved upon the previous best known competitive ratio, which was  $O(\log n)$ .

### 5.15.1 Structure

Tango trees work by partitioning a binary search tree into a set of *preferred paths*, which are themselves stored in auxiliary trees (so the tango tree is represented as a tree of trees).

#### Reference Tree

To construct a tango tree, we simulate a **complete binary search tree** called the *reference tree*, which is simply a traditional binary search tree containing all the elements. This tree never shows up in the actual implementation, but is the conceptual basis behind the following pieces of a tango tree.

#### Preferred Paths

First, we define for each node its *preferred child*, which informally is the most-recently-touched child by a traditional binary search tree lookup. More formally, consider a **subtree**  $T$ , rooted at  $p$ , with children  $l$  (left) and  $r$  (right). We set  $r$  as the preferred child of  $p$  if the most recently accessed node in  $T$  is in the subtree rooted at  $r$ , and  $l$  as the preferred child otherwise. Note that if the most recently accessed node of  $T$  is  $p$  itself, then  $l$  is the preferred child by definition.

A preferred path is defined by starting at the root and following the preferred children until reaching a leaf node. Removing the nodes on this path partitions the remainder of the tree into a number of subtrees, and we recurse on each subtree (forming a preferred path from its root, which partitions the subtree into more subtrees).

#### Auxiliary Trees

To represent a preferred path, we store its nodes in a **balanced binary search tree**, specifically a **red-black tree**. For each non-leaf node  $n$  in a preferred path  $P$ , it has a non-preferred child  $c$ , which is the root of a new auxiliary tree. We attach this other auxiliary tree's root ( $c$ ) to  $n$  in

$P$ , thus linking the auxiliary trees together. We also augment the auxiliary tree by storing at each node the minimum and maximum depth (depth in the reference tree, that is) of nodes in the subtree under that node.

### 5.15.2 Algorithm

#### Searching

To search for an element in the tango tree, we simply simulate searching the reference tree. We start by searching the preferred path connected to the root, which is simulated by searching the auxiliary tree corresponding to that preferred path. If the auxiliary tree doesn't contain the desired element, the search terminates on the parent of the root of the subtree containing the desired element (the beginning of another preferred path), so we simply proceed by searching the auxiliary tree for that preferred path, and so forth.

#### Updating

In order to maintain the structure of the tango tree (auxiliary trees correspond to preferred paths), we must do some updating work whenever preferred children change as a result of searches. When a preferred child changes, the top part of a preferred path becomes detached from the bottom part (which becomes its own preferred path) and reattached to another preferred path (which becomes the new bottom part). In order to do this efficiently, we'll define *cut* and *join* operations on our auxiliary trees.

**Join** Our *join* operation will combine two auxiliary trees as long as they have the property that the top node of one (in the reference tree) is a child of the bottom node of the other (essentially, that the corresponding preferred paths can be concatenated). This will work based on the *concatenate* operation of red-black trees, which combines two trees as long as they have the property that all elements of one are less than all elements of the other, and *split*, which does the reverse. In the reference tree, note that there exist two nodes in the top path such that a node is in the bottom path if and only if its key-value is between them. Now, to join the bottom path to the top path, we simply *split* the top path between those two nodes, then *concatenate* the two resulting auxiliary trees on either side of the bottom path's auxiliary tree, and we have our final, joined auxiliary tree.

**Cut** Our *cut* operation will break a preferred path into two parts at a given node, a top part and a bottom part. More formally, it'll partition an auxiliary tree into two auxiliary trees, such that one contains all nodes at or above a certain depth in the reference tree, and the other contains all nodes below that depth. As in *join*, note that the top part has two nodes that bracket the bottom part. Thus,

we can simply *split* on each of these two nodes to divide the path into three parts, then *concatenate* the two outer ones so we end up with two parts, the top and bottom, as desired.

### 5.15.3 Analysis

In order to bound the competitive ratio for tango trees, we must find a lower bound on the performance of the optimal offline tree that we use as a benchmark. Once we find an upper bound on the performance of the tango tree, we can divide them to bound the competitive ratio.

#### Interleave Bound

Main article: [Interleave lower bound](#)

To find a lower bound on the work done by the optimal offline binary search tree, we again use the notion of preferred children. When considering an access sequence (a sequence of searches), we keep track of how many times a reference tree node's preferred child switches. The total number of switches (summed over all nodes) gives an asymptotic lower bound on the work done by any binary search tree algorithm on the given access sequence. This is called the *interleave lower bound*.<sup>[1]</sup>

#### Tango Tree

In order to connect this to tango trees, we will find an upper bound on the work done by the tango tree for a given access sequence. Our upper bound will be  $(k + 1)O(\log \log n)$ , where  $k$  is the number of interleaves.

The total cost is divided into two parts, searching for the element, and updating the structure of the tango tree to maintain the proper invariants (switching preferred children and re-arranging preferred paths).

**Searching** To see that the searching (not updating) fits in this bound, simply note that every time an auxiliary tree search is unsuccessful and we have to move to the next auxiliary tree, that results in a preferred child switch (since the parent preferred path now switches directions to join the child preferred path). Since all auxiliary tree searches are unsuccessful except the last one (we stop once a search is successful, naturally), we search  $k + 1$  auxiliary trees. Each search takes  $O(\log \log n)$ , because an auxiliary tree's size is bounded by  $\log n$ , the height of the reference tree.

**Updating** The update cost fits within this bound as well, because we only have to perform one *cut* and one *join* for every visited auxiliary tree. A single *cut* or *join* operation takes only a constant number of searches, *splits*,

and concatenates, each of which takes logarithmic time in the size of the auxiliary tree, so our update cost is  $(k+1)O(\log \log n)$ .

### Competitive Ratio

Tango trees are  $O(\log \log n)$ -competitive, because the work done by the optimal offline binary search tree is at least linear in  $k$  (the total number of preferred child switches), and the work done by the tango tree is at most  $(k+1)O(\log \log n)$ .

#### 5.15.4 See also

- Splay tree
- Optimal binary search tree
- Red-black tree
- Tree (data structure)

#### 5.15.5 References

- [1] Demaine, E. D.; Harmon, D.; Iacono, J.; Pătrașcu, M. (2007). “Dynamic Optimality—Almost”. *SIAM Journal on Computing* **37**: 240. doi:10.1137/S0097539705447347.

## 5.16 Van Emde Boas tree

A **Van Emde Boas tree** (or **Van Emde Boas priority queue**; Dutch pronunciation: [van 'emdə 'bo:as]), also known as a **vEB tree**, is a **tree data structure** which implements an **associative array** with  $m$ -bit integer keys. It performs all operations in  $O(\log m)$  time, or equivalently in  $O(\log \log M)$  time, where  $M=2^m$  is the maximum number of elements that can be stored in the tree. The  $M$  is not to be confused with the actual number of elements stored in the tree, by which the performance of other tree data-structures is often measured. The vEB tree has good space efficiency when it contains a large number of elements, as discussed below. It was invented by a team led by Peter van Emde Boas in 1975.<sup>[1]</sup>

### 5.16.1 Supported operations

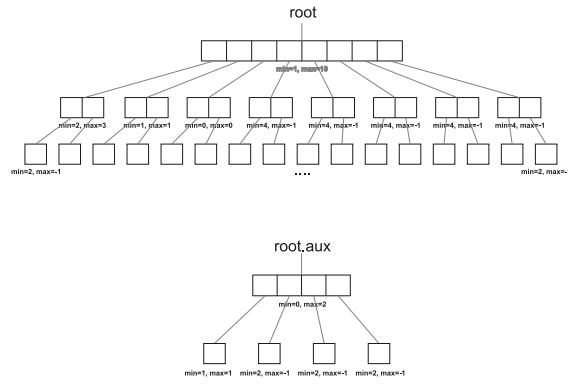
A vEB supports the operations of an *ordered associative array*, which includes the usual associative array operations along with two more *order* operations, *FindNext* and *FindPrevious*:<sup>[2]</sup>

- *Insert*: insert a key/value pair with an  $m$ -bit key
- *Delete*: remove the key/value pair with a given key

- *Lookup*: find the value associated with a given key
- *FindNext*: find the key/value pair with the smallest key at least a given  $k$
- *FindPrevious*: find the key/value pair with the largest key at most a given  $k$

A vEB tree also supports the operations *Minimum* and *Maximum*, which return the minimum and maximum element stored in the tree respectively.<sup>[3]</sup> These both run in  $O(1)$  time, since the minimum and maximum element are stored as attributes in each tree.

### 5.16.2 How it works



An example Van Emde Boas tree with dimension 5 and the root's aux structure after 1, 2, 3, 5, 8 and 10 have been inserted.

For the sake of simplicity, let  $\log_2 m = k$  for some integer  $k$ . Define  $M=2^m$ . A vEB tree  $T$  over the universe  $\{0, \dots, M-1\}$  has a root node that stores an array  $T.children$  of length  $\sqrt{M}$ .  $T.children[i]$  is a pointer to a vEB tree that is responsible for the values  $\{i\sqrt{M}, \dots, (i+1)\sqrt{M}-1\}$ . Additionally,  $T$  stores two values  $T.min$  and  $T.max$  as well as an auxiliary vEB tree  $T.aux$ .

Data is stored in a vEB tree as follows: The smallest value currently in the tree is stored in  $T.min$  and largest value is stored in  $T.max$ . Note that  $T.min$  is not stored anywhere else in the vEB tree, while  $T.max$  is. If  $T$  is empty then we use the convention that  $T.max=-1$  and  $T.min=M$ . Any other value  $x$  is stored in the subtree  $T.children[i]$  where  $i = \left\lfloor \frac{x}{\sqrt{M}} \right\rfloor$ . The auxiliary tree  $T.aux$  keeps track of which children are non-empty, so  $T.aux$  contains the value  $j$  if and only if  $T.children[j]$  is non-empty.

### FindNext

The operation  $FindNext(T, x)$  that searches for the successor of an element  $x$  in a vEB tree proceeds as follows: If  $x \leq T.min$  then the search is complete, and the answer is  $T.min$ . If  $x > T.max$  then the next element does not exist, return  $M$ . Otherwise, let  $i=x/\sqrt{M}$ . If  $x \leq T.children[i].max$  then the value being searched for

is contained in  $T.children[i]$  so the search proceeds recursively in  $T.children[i]$ . Otherwise, we search for the value  $i$  in  $T.aux$ . This gives us the index  $j$  of the first subtree that contains an element larger than  $x$ . The algorithm then returns  $T.children[j].min$ . The element found on the children level needs to be composed with the high bits to form a complete next element.

```
function FindNext(T, x). if x ≤ T.min then return T.min if x > T.max then // no next element return M i = floor(x/√M) lo = x % √M hi = x - lo if lo ≤ T.children[i].max then return hi + FindNext(T.children[i], lo) return hi + T.children[FindNext(T.aux, i)].min end
```

Note that, in any case, the algorithm performs  $O(1)$  work and then possibly recurses on a subtree over a universe of size  $M^{1/2}$  (an  $m/2$  bit universe). This gives a recurrence for the running time of  $T(m) = T(m/2) + O(1)$ , which resolves to  $O(\log m) = O(\log \log M)$ .

## Insert

The call  $insert(T, x)$  that inserts a value  $x$  into a vEB tree  $T$  operates as follows:

If  $T$  is empty then we set  $T.min = T.max = x$  and we are done.

Otherwise, if  $x < T.min$  then we insert  $T.min$  into the subtree  $i$  responsible for  $T.min$  and then set  $T.min = x$ . If  $T.children[i]$  was previously empty, then we also insert  $i$  into  $T.aux$ .

Otherwise, if  $x > T.max$  then we insert  $x$  into the subtree  $i$  responsible for  $x$  and then set  $T.max = x$ . If  $T.children[i]$  was previously empty, then we also insert  $i$  into  $T.aux$ .

Otherwise,  $T.min < x < T.max$  so we insert  $x$  into the subtree  $i$  responsible for  $x$ . If  $T.children[i]$  was previously empty, then we also insert  $i$  into  $T.aux$ .

In code:

```
function Insert(T, x) if T.min > T.max then // T is empty
 T.min = T.max = x; return if T.min == T.max then if
 x < T.min then T.min = x if x > T.max then T.max =
 x if x < T.min then swap(x, T.min) if x > T.max then
 T.max = x i = floor(x / √M) Insert(T.children[i], x %
 √M) if T.children[i].min == T.children[i].max then Insert(T.aux, i) end
```

The key to the efficiency of this procedure is that inserting an element into an empty vEB tree takes  $O(1)$  time. So, even though the algorithm sometimes makes two recursive calls, this only occurs when the first recursive call was into an empty subtree. This gives the same running time recurrence of  $T(m) = T(m/2) + O(1)$  as before.

## Delete

Deletion from vEB trees is the trickiest of the operations. The call  $Delete(T, x)$  that deletes a value  $x$  from a vEB tree  $T$  operates as follows:

If  $T.min = T.max = x$  then  $x$  is the only element stored in the tree and we set  $T.min = M$  and  $T.max = -1$  to indicate that the tree is empty.

Otherwise, if  $x = T.min$  then we need to find the second-smallest value  $y$  in the vEB tree, delete it from its current location, and set  $T.min = y$ . The second-smallest value  $y$  is either  $T.max$  or  $T.children[T.aux.min].min$ , so it can be found in  $O(1)$  time. In the latter case we delete  $y$  from the subtree that contains it.

Similarly, if  $x = T.max$  then we need to find the second-largest value  $y$  in the vEB tree and set  $T.max = y$ . The second-largest value  $y$  is either  $T.min$  or  $T.children[T.aux.max].max$ , so it can be found in  $O(1)$  time. We also delete  $x$  from the subtree that contains it.

In case where  $x$  is not  $T.min$  or  $T.max$ , and  $T$  has no other elements, we know  $x$  is not in  $T$  and return without further operations.

Otherwise, we have the typical case where  $x \neq T.min$  and  $x \neq T.max$ . In this case we delete  $x$  from the subtree  $T.children[i]$  that contains  $x$ .

In any of the above cases, if we delete the last element  $x$  or  $y$  from any subtree  $T.children[i]$  then we also delete  $i$  from  $T.aux$

In code:

```
function Delete(T, x) if T.min == T.max == x then
 T.min = M T.max = -1 return if x == T.min then if
 T.aux is empty then T.min = T.max return else x =
 T.children[T.aux.min].min T.min = x if x == T.max then
 if T.aux is empty then T.max = T.min return else T.max =
 T.children[T.aux.max].max if T.aux is empty then re-
 turn i = floor(x / √M) Delete(T.children[i], x %
 √M) if
 T.children[i] is empty then Delete(T.aux, i) end
```

Again, the efficiency of this procedure hinges on the fact that deleting from a vEB tree that contains only one element takes only constant time. In particular, the last line of code only executes if  $x$  was the only element in  $T.children[i]$  prior to the deletion.

## Discussion

The assumption that  $\log m$  is an integer is unnecessary. The operations  $x/\sqrt{M}$  and  $x\% \sqrt{M}$  can be replaced by taking only higher-order  $\text{ceil}(m/2)$  and the lower-order  $\text{floor}(m/2)$  bits of  $x$ , respectively. On any existing machine, this is more efficient than division or remainder computations.

The implementation described above uses pointers and occupies a total space of  $O(M) = O(2^m)$ . This can be

seen as follows. The recurrence is  $S(M) = O(\sqrt{M}) + (\sqrt{M} + 1) \cdot S(\sqrt{M})$ . Resolving that would lead to  $S(M) \in (1 + \sqrt{M})^{\log \log M} + \log \log M \cdot O(\sqrt{M})$ . One can, fortunately, also show that  $S(M) = M - 2$  by induction.<sup>[4]</sup>

In practical implementations, especially on machines with *shift-by-k* and *find first zero* instructions, performance can further be improved by switching to a **bit array** once  $m$  equal to the **word size** (or a small multiple thereof) is reached. Since all operations on a single word are constant time, this does not affect the asymptotic performance, but it does avoid the majority of the pointer storage and several pointer dereferences, achieving a significant practical savings in time and space with this trick.

An obvious optimization of vEB trees is to discard empty subtrees. This makes vEB trees quite compact when they contain many elements, because no subtrees are created until something needs to be added to them. Initially, each element added creates about  $\log(m)$  new trees containing about  $m/2$  pointers all together. As the tree grows, more and more subtrees are reused, especially the larger ones. In a full tree of  $2^m$  elements, only  $O(2^m)$  space is used. Moreover, unlike a binary search tree, most of this space is being used to store data: even for billions of elements, the pointers in a full vEB tree number in the thousands.

However, for small trees the overhead associated with vEB trees is enormous: on the order of  $\sqrt{M}$ . This is one reason why they are not popular in practice. One way of addressing this limitation is to use only a fixed number of bits per level, which results in a **trie**. Alternatively, each table may be replaced by a **hash table**, reducing the space to  $O(n)$  (where  $n$  is the number of elements stored in the data structure) at the expense of making the data structure randomized. Other structures, including **y-fast tries** and **x-fast tries** have been proposed that have comparable update and query times and also use randomized hash tables to reduce the space to  $O(n)$  or  $O(n \log M)$ .

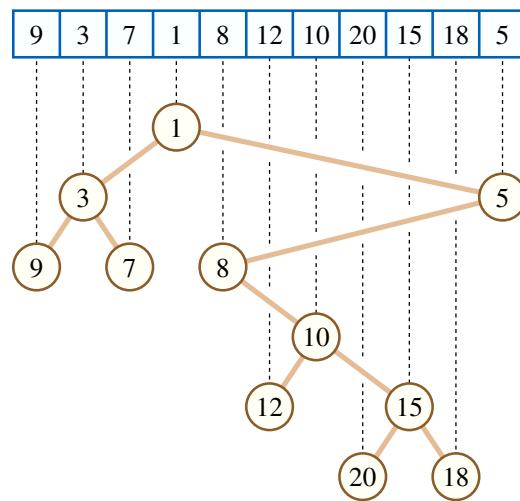
### 5.16.3 References

- [1] Peter van Emde Boas: *Preserving order in a forest in less than logarithmic time* (*Proceedings of the 16th Annual Symposium on Foundations of Computer Science* 10: 75-84, 1975)
- [2] Gudmund Skovbjerg Frandsen: *Dynamic algorithms: Course notes on van Emde Boas trees* (PDF) (University of Aarhus, Department of Computer Science)
- [3]
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Third Edition. MIT Press, 2009. ISBN 0-262-53305-8. Chapter 20: The van Emde Boas tree, pp. 531-560.
- [4] Rex, A. "Determining the space complexity of van Emde Boas trees". Retrieved 2011-05-27.

### Further reading

- Erik Demaine, Shantanu Sen, and Jeff Lindy. Massachusetts Institute of Technology. 6.897: Advanced Data Structures (Spring 2003). Lecture 1 notes: Fixed-universe successor problem, van Emde Boas. Lecture 2 notes: More van Emde Boas, ....
- van Emde Boas, P.; Kaas, R.; Zijlstra, E. (1976). "Design and implementation of an efficient priority queue". *Mathematical Systems Theory* 10: 99-127. doi:10.1007/BF01683268.

## 5.17 Cartesian tree



A sequence of numbers and the Cartesian tree derived from them.

In **computer science**, a **Cartesian tree** is a **binary tree** derived from a sequence of numbers; it can be uniquely defined from the properties that it is **heap-ordered** and that a **symmetric (in-order)** traversal of the tree returns the original sequence. Introduced by Vuillemin (1980) in the context of **geometric range searching** data structures, Cartesian trees have also been used in the definition of the **treap** and **randomized binary search tree** data structures for **binary search** problems. The Cartesian tree for a sequence may be constructed in **linear time** using a stack-based algorithm for finding all nearest smaller values in a sequence.

### 5.17.1 Definition

The Cartesian tree for a sequence of distinct numbers can be uniquely defined by the following properties:

1. The Cartesian tree for a sequence has one node for each number in the sequence. Each node is associated with a single sequence value.

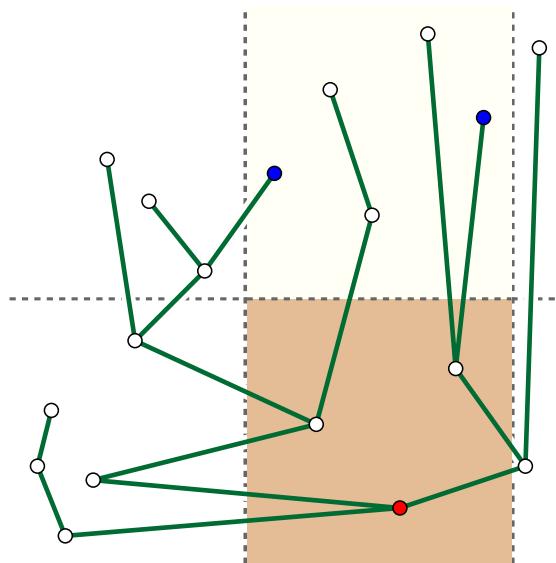
2. A symmetric (in-order) traversal of the tree results in the original sequence. That is, the left subtree consists of the values earlier than the root in the sequence order, while the right subtree consists of the values later than the root, and a similar ordering constraint holds at each lower node of the tree.
3. The tree has the **heap property**: the parent of any non-root node has a smaller value than the node itself.<sup>[1]</sup>

Based on the heap property, the root of the tree must be the smallest number in the sequence. From this, the tree itself may also be defined recursively: the root is the minimum value of the sequence, and the left and right subtrees are the Cartesian trees for the subsequences to the left and right of the root value. Therefore, the three properties above uniquely define the Cartesian tree.

If a sequence of numbers contains repetitions, the Cartesian tree may be defined by determining a consistent tie-breaking rule (for instance, determining that the first of two equal elements is treated as the smaller of the two) before applying the above rules.

An example of a Cartesian tree is shown in the figure above.

### 5.17.2 Range searching and lowest common ancestors



*Two-dimensional range-searching using a Cartesian tree: the bottom point (red in the figure) within a three-sided region with two vertical sides and one horizontal side (if the region is nonempty) may be found as the nearest common ancestor of the leftmost and rightmost points (the blue points in the figure) within the slab defined by the vertical region boundaries. The remaining points in the three-sided region may be found by splitting it by a vertical line through the bottom point and recursing.*

Cartesian trees may be used as part of an efficient data structure for **range minimum queries**, a range searching problem involving queries that ask for the minimum value in a contiguous subsequence of the original sequence.<sup>[2]</sup> In a Cartesian tree, this minimum value may be found at the **lowest common ancestor** of the leftmost and rightmost values in the subsequence. For instance, in the subsequence (12,10,20,15) of the sequence shown in the first illustration, the minimum value of the subsequence (10) forms the lowest common ancestor of the leftmost and rightmost values (12 and 15). Because lowest common ancestors may be found in constant time per query, using a data structure that takes linear space to store and that may be constructed in linear time,<sup>[3]</sup> the same bounds hold for the range minimization problem.

Bender & Farach-Colton (2000) reversed this relationship between the two data structure problems by showing that lowest common ancestors in an input tree could be solved efficiently applying a non-tree-based technique for range minimization. Their data structure uses an **Euler tour** technique to transform the input tree into a sequence and then finds range minima in the resulting sequence. The sequence resulting from this transformation has a special form (adjacent numbers, representing heights of adjacent nodes in the tree, differ by  $\pm 1$ ) which they take advantage of in their data structure; to solve the range minimization problem for sequences that do not have this special form, they use Cartesian trees to transform the range minimization problem into a lowest common ancestor problem, and then apply the Euler tour technique to transform the problem again into one of range minimization for sequences with this special form.

The same range minimization problem may also be given an alternative interpretation in terms of two dimensional range searching. A collection of finitely many points in the **Cartesian plane** may be used to form a Cartesian tree, by sorting the points by their  $x$ -coordinates and using the  $y$ -coordinates in this order as the sequence of values from which this tree is formed. If  $S$  is the subset of the input points within some vertical slab defined by the inequalities  $L \leq x \leq R$ ,  $p$  is the leftmost point in  $S$  (the one with minimum  $x$ -coordinate), and  $q$  is the rightmost point in  $S$  (the one with maximum  $x$ -coordinate) then the lowest common ancestor of  $p$  and  $q$  in the Cartesian tree is the bottommost point in the slab. A three-sided range query, in which the task is to list all points within a region bounded by the three inequalities  $L \leq x \leq R$  and  $y \leq T$ , may be answered by finding this bottommost point  $b$ , comparing its  $y$ -coordinate to  $T$ , and (if the point lies within the three-sided region) continuing recursively in the two slabs bounded between  $p$  and  $b$  and between  $b$  and  $q$ . In this way, after the leftmost and rightmost points in the slab are identified, all points within the three-sided region may be listed in constant time per point.<sup>[4]</sup>

The same construction, of lowest common ancestors in a Cartesian tree, makes it possible to construct a data structure with linear space that allows the distances between

pairs of points in any **ultrametric space** to be queried in constant time per query. The distance within an ultrametric is the same as the **minimax path weight** in the **minimum spanning tree** of the metric.<sup>[5]</sup> From the minimum spanning tree, one can construct a Cartesian tree, the root node of which represents the heaviest edge of the minimum spanning tree. Removing this edge partitions the minimum spanning tree into two subtrees, and Cartesian trees recursively constructed for these two subtrees form the children of the root node of the Cartesian tree. The leaves of the Cartesian tree represent points of the metric space, and the lowest common ancestor of two leaves in the Cartesian tree is the heaviest edge between those two points in the minimum spanning tree, which has weight equal to the distance between the two points. Once the minimum spanning tree has been found and its edge weights sorted, the Cartesian tree may be constructed in linear time.<sup>[6]</sup>

### 5.17.3 Treaps

*Main article: Treap*

Because a Cartesian tree is a binary tree, it is natural to use it as a **binary search tree** for an ordered sequence of values. However, defining a Cartesian tree based on the same values that form the search keys of a binary search tree does not work well: the Cartesian tree of a sorted sequence is just a **path**, rooted at its leftmost endpoint, and binary searching in this tree degenerates to **sequential search** in the path. However, it is possible to generate more-balanced search trees by generating **priorities** values for each search key that are different than the key itself, sorting the inputs by their key values, and using the corresponding sequence of priorities to generate a Cartesian tree. This construction may equivalently be viewed in the geometric framework described above, in which the  $x$ -coordinates of a set of points are the search keys and the  $y$ -coordinates are the priorities.

This idea was applied by Seidel & Aragon (1996), who suggested the use of random numbers as priorities. The data structure resulting from this random choice is called a **treap**, due to its combination of binary search tree and binary heap features. An insertion into a treap may be performed by inserting the new key as a leaf of an existing tree, choosing a priority for it, and then performing **tree rotation** operations along a path from the node to the root of the tree to repair any violations of the heap property caused by this insertion; a deletion may similarly be performed by a constant amount of change to the tree followed by a sequence of rotations along a single path in the tree.

If the priorities of each key are chosen randomly and independently once whenever the key is inserted into the tree, the resulting Cartesian tree will have the same properties as a **random binary search tree**, a tree computed by

inserting the keys in a randomly chosen permutation starting from an empty tree, with each insertion leaving the previous tree structure unchanged and inserting the new node as a leaf of the tree. Random binary search trees had been studied for much longer, and are known to behave well as search trees (they have **logarithmic depth** with high probability); the same good behavior carries over to treaps. It is also possible, as suggested by Aragon and Seidel, to reprioritize frequently-accessed nodes, causing them to move towards the root of the treap and speeding up future accesses for the same keys.

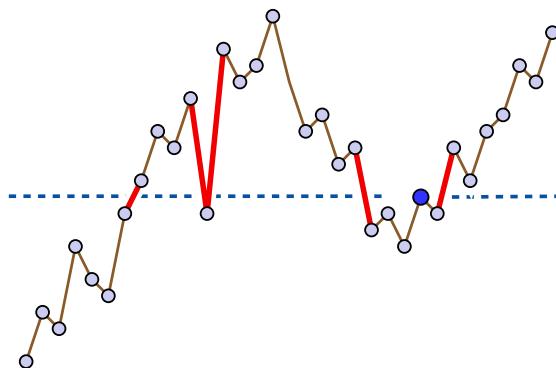
### 5.17.4 Efficient construction

A Cartesian tree may be constructed in linear time from its input sequence. One method is to simply process the sequence values in left-to-right order, maintaining the Cartesian tree of the nodes processed so far, in a structure that allows both upwards and downwards traversal of the tree. To process each new value  $x$ , start at the node representing the value prior to  $x$  in the sequence and follow the path from this node to the root of the tree until finding a value  $y$  smaller than  $x$ . This node  $y$  is the parent of  $x$ , and the previous right child of  $y$  becomes the new left child of  $x$ . The total time for this procedure is linear, because the time spent searching for the parent  $y$  of each new node  $x$  can be **charged** against the number of nodes that are removed from the rightmost path in the tree.<sup>[4]</sup>

An alternative linear-time construction algorithm is based on the **all nearest smaller values** problem. In the input sequence, one may define the *left neighbor* of a value  $x$  to be the value that occurs prior to  $x$ , is smaller than  $x$ , and is closer in position to  $x$  than any other smaller value. The *right neighbor* is defined symmetrically. The sequence of left neighbors may be found by an algorithm that maintains a **stack** containing a subsequence of the input. For each new sequence value  $x$ , the stack is popped until it is empty or its top element is smaller than  $x$ , and then  $x$  is pushed onto the stack. The left neighbor of  $x$  is the top element at the time  $x$  is pushed. The right neighbors may be found by applying the same stack algorithm to the reverse of the sequence. The parent of  $x$  in the Cartesian tree is either the left neighbor of  $x$  or the right neighbor of  $x$ , whichever exists and has a larger value. The left and right neighbors may also be constructed efficiently by **parallel algorithms**, so this formulation may be used to develop efficient parallel algorithms for Cartesian tree construction.<sup>[7]</sup>

### 5.17.5 Application in sorting

Levcopoulos & Petersson (1989) describe a **sorting algorithm** based on Cartesian trees. They describe the algorithm as based on a tree with the maximum at the root, but it may be modified straightforwardly to support a Cartesian tree with the convention that the minimum value is



Pairs of consecutive sequence values (shown as the thick red edges) that bracket a sequence value  $x$  (the darker blue point). The cost of including  $x$  in the sorted order produced by the Levcopoulos–Petersson algorithm is proportional to the logarithm of this number of bracketing pairs.

at the root. For consistency, it is this modified version of the algorithm that is described below.

The Levcopoulos–Petersson algorithm can be viewed as a version of **selection sort** or **heap sort** that maintains a **priority queue** of candidate minima, and that at each step finds and removes the minimum value in this queue, moving this value to the end of an output sequence. In their algorithm, the priority queue consists only of elements whose parent in the Cartesian tree has already been found and removed. Thus, the algorithm consists of the following steps:

1. Construct a Cartesian tree for the input sequence
2. Initialize a priority queue, initially containing only the tree root
3. While the priority queue is non-empty:
  - Find and remove the minimum value  $x$  in the priority queue
  - Add  $x$  to the output sequence
  - Add the Cartesian tree children of  $x$  to the priority queue

As Levcopoulos and Petersson show, for input sequences that are already nearly sorted, the size of the priority queue will remain small, allowing this method to take advantage of the nearly-sorted input and run more quickly. Specifically, the worst-case running time of this algorithm is  $O(n \log k)$ , where  $k$  is the average, over all values  $x$  in the sequence, of the number of consecutive pairs of sequence values that bracket  $x$ . They also prove a lower bound stating that, for any  $n$  and  $k = \omega(1)$ , any comparison-based sorting algorithm must use  $\Omega(n \log k)$  comparisons for some inputs.

### 5.17.6 History

Cartesian trees were introduced and named by Vuillemin (1980). The name is derived from the **Cartesian coordinate system** for the plane: in Vuillemin's version of this structure, as in the two-dimensional range searching application discussed above, a Cartesian tree for a point set has the sorted order of the points by their  $x$ -coordinates as its symmetric traversal order, and it has the heap property according to the  $y$ -coordinates of the points. Gabow, Bentley & Tarjan (1984) and subsequent authors followed the definition here in which a Cartesian tree is defined from a sequence; this change generalizes the geometric setting of Vuillemin to allow sequences other than the sorted order of  $x$ -coordinates, and allows the Cartesian tree to be applied to non-geometric problems as well.

### 5.17.7 Notes

- [1] In some references, the ordering is reversed, so the parent of any node always has a larger value and the root node holds the maximum value.
- [2] Gabow, Bentley & Tarjan (1984); Bender & Farach-Colton (2000).
- [3] Harel & Tarjan (1984); Schieber & Vishkin (1988).
- [4] Gabow, Bentley & Tarjan (1984).
- [5] Hu (1961); Leclerc (1981)
- [6] Demaine, Landau & Weimann (2009).
- [7] Berkman, Schieber & Vishkin (1993).

### 5.17.8 References

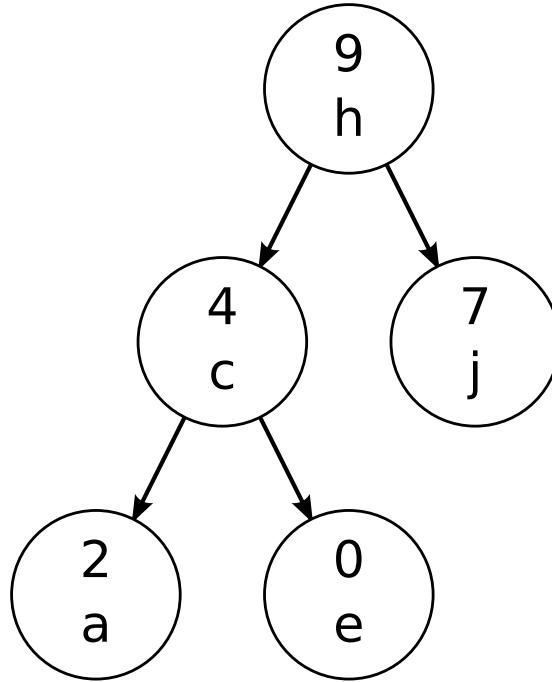
- Bender, Michael A.; Farach-Colton, Martin (2000), “The LCA problem revisited”, *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, Springer-Verlag, Lecture Notes in Computer Science 1776, pp. 88–94.
- Berkman, Omer; Schieber, Baruch; Vishkin, Uzi (1993), “Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values”, *Journal of Algorithms* 14 (3): 344–370, doi:10.1006/jagm.1993.101.
- Demaine, Erik D.; Landau, Gad M.; Weimann, Oren (2009), “On Cartesian trees and range minimum queries”, *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5–12, 2009*, Lecture Notes in Computer Science 5555, pp. 341–353, doi:10.1007/978-3-642-02927-1\_29, ISBN 978-3-642-02926-4.

- Fischer, Johannes; Heun, Volker (2006), “Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE”, *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science **4009**, Springer-Verlag, pp. 36–48, doi:10.1007/11780441\_5, ISBN 978-3-540-35455-0
- Fischer, Johannes; Heun, Volker (2007), “A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array”, *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, Lecture Notes in Computer Science **4614**, Springer-Verlag, pp. 459–470, doi:10.1007/978-3-540-74450-4\_41, ISBN 978-3-540-74449-8
- Gabow, Harold N.; Bentley, Jon Louis; Tarjan, Robert E. (1984), “Scaling and related techniques for geometry problems”, *STOC '84: Proc. 16th ACM Symp. Theory of Computing*, New York, NY, USA: ACM, pp. 135–143, doi:10.1145/800057.808675, ISBN 0-89791-133-4.
- Harel, Dov; Tarjan, Robert E. (1984), “Fast algorithms for finding nearest common ancestors”, *SIAM Journal on Computing* **13** (2): 338–355, doi:10.1137/0213024.
- Hu, T. C. (1961), “The maximum capacity route problem”, *Operations Research* **9** (6): 898–900, doi:10.1287/opre.9.6.898, JSTOR 167055.
- Leclerc, Bruno (1981), “Description combinatoire des ultramétriques”, *Centre de Mathématique Sociale. École Pratique des Hautes Études. Mathématiques et Sciences Humaines* (in French) (73): 5–37, 127, MR 623034.
- Levcopoulos, Christos; Petersson, Ola (1989), “Heapsort - Adapted for Presorted Files”, *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **382**, London, UK: Springer-Verlag, pp. 499–509, doi:10.1007/3-540-51542-9\_41.
- Seidel, Raimund; Aragon, Cecilia R. (1996), “Randomized Search Trees”, *Algorithmica* **16** (4/5): 464–497, doi:10.1007/s004539900061.
- Schieber, Baruch; Vishkin, Uzi (1988), “On finding lowest common ancestors: simplification and parallelization”, *SIAM Journal on Computing* **17** (6): 1253–1262, doi:10.1137/0217079.
- Vuillemin, Jean (1980), “A unifying look at data structures”, *Commun. ACM* (New York, NY, USA: ACM) **23** (4): 229–239, doi:10.1145/358841.358852.

## 5.18 Treap

In computer science, the **treap** and the **randomized binary search tree** are two closely related forms of binary search tree data structures that maintain a dynamic set of ordered keys and allow binary searches among the keys. After any sequence of insertions and deletions of keys, the shape of the tree is a random variable with the same probability distribution as a random binary tree; in particular, with high probability its height is proportional to the logarithm of the number of keys, so that each search, insertion, or deletion operation takes logarithmic time to perform.

### 5.18.1 Description



A treap with alphabetic key and numeric max heap order

The treap was first described by Cecilia R. Aragon and Raimund Seidel in 1989;<sup>[1][2]</sup> its name is a portmanteau of tree and heap. It is a Cartesian tree in which each key is given a (randomly chosen) numeric priority. As with any binary search tree, the inorder traversal order of the nodes is the same as the sorted order of the keys. The structure of the tree is determined by the requirement that it be heap-ordered: that is, the priority number for any non-leaf node must be greater than or equal to the priority of its children. Thus, as with Cartesian trees more generally, the root node is the maximum-priority node, and its left and right subtrees are formed in the same manner from the subsequences of the sorted order to the left and right of that node.

An equivalent way of describing the treap is that it could be formed by inserting the nodes highest-priority-first

into a binary search tree without doing any rebalancing. Therefore, if the priorities are independent random numbers (from a distribution over a large enough space of possible priorities to ensure that two nodes are very unlikely to have the same priority) then the shape of a treap has the same probability distribution as the shape of a **random binary search tree**, a search tree formed by inserting the nodes without rebalancing in a randomly chosen insertion order. Because random binary search trees are known to have logarithmic height with high probability, the same is true for treaps.

Aragon and Seidel also suggest assigning higher priorities to frequently accessed nodes, for instance by a process that, on each access, chooses a random number and replaces the priority of the node with that number if it is higher than the previous priority. This modification would cause the tree to lose its random shape; instead, frequently accessed nodes would be more likely to be near the root of the tree, causing searches for them to be faster.

Naor and Nissim<sup>[3]</sup> describe an application in maintaining authorization certificates in **public-key cryptosystems**.

## 5.18.2 Operations

Treaps support the following basic operations:

- To search for a given key value, apply a standard **binary search algorithm** in a binary search tree, ignoring the priorities.
- To insert a new key  $x$  into the treap, generate a random priority  $y$  for  $x$ . Binary search for  $x$  in the tree, and create a new node at the leaf position where the binary search determines a node for  $x$  should exist. Then, as long as  $x$  is not the root of the tree and has a larger priority number than its parent  $z$ , perform a **tree rotation** that reverses the parent-child relation between  $x$  and  $z$ .
- To delete a node  $x$  from the treap, if  $x$  is a leaf of the tree, simply remove it. If  $x$  has a single child  $z$ , remove  $x$  from the tree and make  $z$  be the child of the parent of  $x$  (or make  $z$  the root of the tree if  $x$  had no parent). Finally, if  $x$  has two children, swap its position in the tree with the position of its immediate successor  $z$  in the sorted order, resulting in one of the previous cases. In this final case, the swap may violate the heap-ordering property for  $z$ , so additional rotations may need to be performed to restore this property.

### Bulk operations

In addition to the single-element insert, delete and lookup operations, several fast “bulk” operations have been defined on treaps: **union**, **intersection** and **set difference**. These rely on two helper operations, **split** and **merge**.

- To split a treap into two smaller treaps, those smaller than key  $x$ , and those larger than key  $x$ , insert  $x$  into the treap with maximum priority—larger than the priority of any node in the treap. After this insertion,  $x$  will be the root node of the treap, all values less than  $x$  will be found in the left subtrep, and all values greater than  $x$  will be found in the right subtrep. This costs as much as a single insertion into the treap.
- Merging two treaps that are the product of a former split, one can safely assume that the greatest value in the first treap is less than the smallest value in the second treap. Insert a value  $x$ , such that  $x$  is larger than this max-value in the first treap, and smaller than the min-value in the second treap, and assign it the minimum priority. After insertion it will be a leaf node, and can easily be deleted. The result is one treap merged from the two original treaps. This is effectively “undoing” a split, and costs the same.

The union of two treaps  $t_1$  and  $t_2$ , representing sets  $A$  and  $B$  is a treap  $t$  that represents  $A \cup B$ . The following recursive algorithm computes the union:

```
function union(t1, t2): if t1 = nil: return t2 if t2 = nil: return t1 if priority(t1) < priority(t2): swap t1 and t2 t<, t> ← split t2 on key(t1) return new node(key(t1), union(left(t1), t<), union(right(t1), t>))
```

Here, **split** is presumed to return two trees: one holding the keys less its input key, one holding the greater keys. (The algorithm is **non-destructive**, but an in-place destructive version exists as well.)

The algorithm for intersection is similar, but requires the **join** helper routine. The complexity of each of union, intersection and difference is  $O(m \log n/m)$  for treaps of sizes  $m$  and  $n$ , with  $m \leq n$ . Moreover, since the recursive calls to union are independent of each other, they can be executed in parallel.<sup>[4]</sup>

## 5.18.3 Randomized binary search tree

The randomized binary search tree, introduced by Martínez and Roura subsequently to the work of Aragon and Seidel on treaps,<sup>[5]</sup> stores the same nodes with the same random distribution of tree shape, but maintains different information within the nodes of the tree in order to maintain its randomized structure.

Rather than storing random priorities on each node, the randomized binary search tree stores a small integer at each node, the number of its descendants (counting itself as one); these numbers may be maintained during tree rotation operations at only a constant additional amount of time per rotation. When a key  $x$  is to be inserted into a tree that already has  $n$  nodes, the insertion algorithm chooses with probability  $1/(n + 1)$  to place  $x$  as the new

root of the tree, and otherwise it calls the insertion procedure recursively to insert  $x$  within the left or right subtree (depending on whether its key is less than or greater than the root). The numbers of descendants are used by the algorithm to calculate the necessary probabilities for the random choices at each step. Placing  $x$  at the root of a subtree may be performed either as in the treap by inserting it at a leaf and then rotating it upwards, or by an alternative algorithm described by Martínez and Roura that splits the subtree into two pieces to be used as the left and right children of the new node.

The deletion procedure for a randomized binary search tree uses the same information per node as the insertion procedure, and like the insertion procedure it makes a sequence of  $O(\log n)$  random decisions in order to join the two subtrees descending from the left and right children of the deleted node into a single tree. If the left or right subtree of the node to be deleted is empty, the join operation is trivial; otherwise, the left or right child of the deleted node is selected as the new subtree root with probability proportional to its number of descendants, and the join proceeds recursively.

#### 5.18.4 Comparison

The information stored per node in the randomized binary tree is simpler than in a treap (a small integer rather than a high-precision random number), but it makes a greater number of calls to the random number generator ( $O(\log n)$  calls per insertion or deletion rather than one call per insertion) and the insertion procedure is slightly more complicated due to the need to update the numbers of descendants per node. A minor technical difference is that, in a treap, there is a small probability of a collision (two keys getting the same priority), and in both cases there will be statistical differences between a true random number generator and the [pseudo-random number generator](#) typically used on digital computers. However, in any case the differences between the theoretical model of perfect random choices used to design the algorithm and the capabilities of actual random number generators are vanishingly small.

Although the treap and the randomized binary search tree both have the same random distribution of tree shapes after each update, the history of modifications to the trees performed by these two data structures over a sequence of insertion and deletion operations may be different. For instance, in a treap, if the three numbers 1, 2, and 3 are inserted in the order 1, 3, 2, and then the number 2 is deleted, the remaining two nodes will have the same parent-child relationship that they did prior to the insertion of the middle number. In a randomized binary search tree, the tree after the deletion is equally likely to be either of the two possible trees on its two nodes, independently of what the tree looked like prior to the insertion of the middle number.

#### 5.18.5 See also

- Finger search

#### 5.18.6 References

- [1] Aragon, Cecilia R.; Seidel, Raimund (1989), “Randomized Search Trees” (PDF), *Proc. 30th Symp. Foundations of Computer Science (FOCS 1989)*, Washington, D.C.: IEEE Computer Society Press, pp. 540–545, doi:10.1109/SFCS.1989.63531, ISBN 0-8186-1982-1
- [2] Seidel, Raimund; Aragon, Cecilia R. (1996), “Randomized Search Trees”, *Algorithmica* **16** (4/5): 464–497, doi:10.1007/s004539900061
- [3] Naor, M.; Nissim, K. (April 2000), “Certificate revocation and certificate update” (PDF), *IEEE Journal on Selected Areas in Communications* **18** (4): 561–570, doi:10.1109/49.839932.
- [4] Blelloch, Guy E.; Reid-Miller, Margaret, (1998), “Fast set operations using treaps”, *Proc. 10th ACM Symp. Parallel Algorithms and Architectures (SPAA 1998)*, New York, NY, USA: ACM, pp. 16–26, doi:10.1145/277651.277660, ISBN 0-89791-989-0.
- [5] Martínez, Conrado; Roura, Salvador (1997), “Randomized binary search trees”, *Journal of the ACM* **45** (2): 288–323, doi:10.1145/274787.274812

#### 5.18.7 External links

- Collection of treap references and info by Cecilia Aragon
- Open Data Structures - Section 7.2 - Treap: A Randomized Binary Search Tree
- Treap Applet by Kubo Kovac
- Animated treap
- Randomized binary search trees. Lecture notes from a course by Jeff Erickson at UIUC. Despite the title, this is primarily about treaps and skip lists; randomized binary search trees are mentioned only briefly.
- A high performance key-value store based on treap by Junyi Sun
- VB6 implementation of treaps. Visual basic 6 implementation of treaps as a COM object.
- ActionScript3 implementation of a treap
- Pure Python and Cython in-memory treap and dup-treap
- Treaps in C#. By Roy Clemmons
- Pure Go in-memory, immutable treaps
- Pure Go persistent treap key-value storage library

# Chapter 6

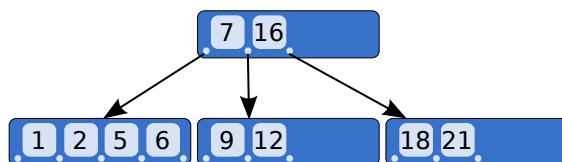
## B-trees

### 6.1 B-tree

Not to be confused with **Binary tree**.

In computer science, a **B-tree** is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a **binary search tree** in that a node can have more than two children (Comer 1979, p. 123). Unlike **self-balancing binary search trees**, the B-tree is optimized for systems that read and write large blocks of data. It is commonly used in **databases** and **filesystems**.

#### 6.1.1 Overview



A B-tree of order 2 (Bayer & McCreight 1972) or order 5 (Knuth 1998).

In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a **2-3 tree**), each internal node may have only 2 or 3 child nodes.

Each internal node of a B-tree will contain a number of keys. The keys act as separation values which divide its **subtrees**. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys:  $a_1$  and  $a_2$ . All values in the leftmost subtree will be less than  $a_1$ , all values in the middle subtree will be between  $a_1$  and  $a_2$ ,

and all values in the rightmost subtree will be greater than  $a_2$ .

Usually, the number of keys is chosen to vary between  $d$  and  $2d$ , where  $d$  is the minimum number of keys, and  $d+1$  is the minimum **degree** or **branching factor** of the tree. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has  $2d$  keys, then adding a key to that node can be accomplished by splitting the  $2d$  key node into two  $d$  key nodes and adding the key to the parent node. Each split node has the required minimum number of keys. Similarly, if an internal node and its neighbor each have  $d$  keys, then a key may be deleted from the internal node by combining with its neighbor. Deleting the key would make the internal node have  $d-1$  keys; joining the neighbor would add  $d$  keys plus one more key brought down from the neighbor's parent. The result is an entirely full node of  $2d$  keys.

The number of branches (or child nodes) from a node will be one more than the number of keys stored in the node. In a 2-3 B-tree, the internal nodes will store either one key (with two child nodes) or two keys (with three child nodes). A B-tree is sometimes described with the parameters  $(d+1) - (2d+1)$  or simply with the highest branching order,  $(2d+1)$ .

A B-tree is kept balanced by requiring that all leaf nodes be at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node farther away from the root.

B-trees have substantial advantages over alternative implementations when the time to access the data of a node greatly exceeds the time spent processing that data, because then the cost of accessing the node may be amortized over multiple operations within the node. This usually occurs when the node data are in **secondary storage** such as **disk drives**. By maximizing the number of keys within each **internal node**, the height of the tree decreases and the number of expensive node accesses is reduced. In addition, rebalancing of the tree occurs less often. The maximum number of child nodes depends on the information that must be stored for each child node and the size of a full **disk block** or an analogous size in secondary

storage. While 2-3 B-trees are easier to explain, practical B-trees using secondary storage need a large number of child nodes to improve performance.

## Variants

The term **B-tree** may refer to a specific design or it may refer to a general class of designs. In the narrow sense, a B-tree stores keys in its internal nodes but need not store those keys in the records at the leaves. The general class includes variations such as the **B+ tree** and the **B\***.

- In the **B+ tree**, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access (Comer 1979, p. 129).
- The **B\***-tree balances more neighboring internal nodes to keep the internal nodes more densely packed (Comer 1979, p. 129). This variant requires non-root nodes to be at least 2/3 full instead of 1/2 (Knuth 1998, p. 488). To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with a node next to it. When both nodes are full, then the two nodes are split into three. Deleting nodes is somewhat more complex than inserting however.
- B-trees can be turned into **order statistic trees** to allow rapid searches for the  $N$ th record in key order, or counting the number of records between any two records, and various other related operations.<sup>[1]</sup>

## Etymology

Rudolf Bayer and Ed McCreight invented the B-tree while working at Boeing Research Labs in 1971 (Bayer & McCreight 1972), but they did not explain what, if anything, the *B* stands for. Douglas Comer explains:

The origin of “B-tree” has never been explained by the authors. As we shall see, “balanced,” “broad,” or “bushy” might apply. Others suggest that the “B” stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as “Bayer”-trees. (Comer 1979, p. 123 footnote 1)

Donald Knuth speculates on the etymology of B-trees in his May, 1980 lecture on the topic “CS144C classroom lecture about disk storage and B-trees”, suggesting the “B” may have originated from Boeing or from Bayer’s name.<sup>[2]</sup>

After a talk at CPM 2013 (24th Annual Symposium on Combinatorial Pattern Matching, Bad Herrenalb, Germany, June 17–19, 2013), Ed McCreight answered a

question on B-tree’s name by Martin Farach-Colton saying: “Bayer and I were in a lunch time where we get to think a name. And we were, so, B, we were thinking... B is, you know... We were working for Boeing at the time, we couldn’t use the name without talking to lawyers. So, there is a B. It has to do with balance, another B. Bayer was the senior author, who did have several years older than I am and had many more publications than I did. So there is another B. And so, at the lunch table we never did resolve whether there was one of those that made more sense than the rest. What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees.”<sup>[3]</sup>

### 6.1.2 The database problem

This section describes a problem faced by database designers, outlines a series of increasingly effective solutions to the problem, and ends by describing how the B-tree solves the problem completely.

#### Time to search a sorted file

Usually, sorting and searching algorithms have been characterized by the number of comparison operations that must be performed using **order notation**. A **binary search** of a sorted table with  $N$  records, for example, can be done in roughly  $\lceil \log_2 N \rceil$  comparisons. If the table had 1,000,000 records, then a specific record could be located with at most 20 comparisons:  $\lceil \log_2(1,000,000) \rceil = 20$ .

Large databases have historically been kept on disk drives. The time to read a record on a disk drive far exceeds the time needed to compare keys once the record is available. The time to read a record from a disk drive involves a **seek time** and a rotational delay. The seek time may be 0 to 20 or more milliseconds, and the rotational delay averages about half the rotation period. For a 7200 RPM drive, the rotation period is 8.33 milliseconds. For a drive such as the Seagate ST3500320NS, the track-to-track seek time is 0.8 milliseconds and the average reading seek time is 8.5 milliseconds.<sup>[4]</sup> For simplicity, assume reading from disk takes about 10 milliseconds.

Naively, then, the time to locate one record out of a million would take 20 disk reads times 10 milliseconds per disk read, which is 0.2 seconds.

The time won’t be that bad because individual records are grouped together in a **disk block**. A disk block might be 16 kilobytes. If each record is 160 bytes, then 100 records could be stored in each block. The disk read time above was actually for an entire block. Once the disk head is in position, one or more disk blocks can be read with little delay. With 100 records per block, the last 6 or so comparisons don’t need to do any disk reads—the comparisons are all within the last disk block read.

To speed the search further, the first 13 to 14 comparisons (which each required a disk access) must be sped up.

### An index speeds the search

A significant improvement can be made with an **index**. In the example above, initial disk reads narrowed the search range by a factor of two. That can be improved substantially by creating an auxiliary index that contains the first record in each disk block (sometimes called a **sparse index**). This auxiliary index would be 1% of the size of the original database, but it can be searched more quickly. Finding an entry in the auxiliary index would tell us which block to search in the main database; after searching the auxiliary index, we would have to search only that one block of the main database—at a cost of one more disk read. The index would hold 10,000 entries, so it would take at most 14 comparisons. Like the main database, the last 6 or so comparisons in the aux index would be on the same disk block. The index could be searched in about 8 disk reads, and the desired record could be accessed in 9 disk reads.

The trick of creating an auxiliary index can be repeated to make an auxiliary index to the auxiliary index. That would make an aux-aux index that would need only 100 entries and would fit in one disk block.

Instead of reading 14 disk blocks to find the desired record, we only need to read 3 blocks. Reading and searching the first (and only) block of the aux-aux index identifies the relevant block in aux-index. Reading and searching that aux-index block identifies the relevant block in the main database. Instead of 150 milliseconds, we need only 30 milliseconds to get the record.

The auxiliary indices have turned the search problem from a binary search requiring roughly  $\log_2 N$  disk reads to one requiring only  $\log_b N$  disk reads where  $b$  is the blocking factor (the number of entries per block:  $b = 100$  entries per block;  $\log_b 1,000,000 = 3$  reads).

In practice, if the main database is being frequently searched, the aux-aux index and much of the aux index may reside in a disk cache, so they would not incur a disk read.

### Insertions and deletions cause trouble

If the database does not change, then compiling the index is simple to do, and the index need never be changed. If there are changes, then managing the database and its index becomes more complicated.

Deleting records from a database doesn't cause much trouble. The index can stay the same, and the record can just be marked as deleted. The database stays in sorted order. If there is a large number of deletions, then the searching and storage become less efficient.

Insertions can be very slow in a sorted sequential file because room for the inserted record must be made. Inserting a record before the first record in the file requires shifting all of the records down one. Such an operation is just too expensive to be practical. A trick is to leave some space lying around to be used for insertions. Instead of densely storing all the records in a block, the block can have some free space to allow for subsequent insertions. Those records would be marked as if they were “deleted” records.

Both insertions and deletions are fast as long as space is available on a block. If an insertion won't fit on the block, then some free space on some nearby block must be found and the auxiliary indices adjusted. The hope is that enough space is nearby such that a lot of blocks do not need to be reorganized. Alternatively, some out-of-sequence disk blocks may be used.

### The B-tree uses all those ideas

The B-tree uses all of the ideas described above. In particular, a B-tree:

- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks to speed insertions and deletions
- keeps the index balanced with an elegant recursive algorithm

In addition, a B-tree minimizes waste by making sure the interior nodes are at least half full. A B-tree can handle an arbitrary number of insertions and deletions.

### 6.1.3 Technical description

#### Terminology

Unfortunately, the literature on B-trees is not uniform in its terminology (Folk & Zoellick 1992, p. 362).

Bayer & McCreight (1972), Comer (1979), and others define the **order** of B-tree as the minimum number of keys in a non-root node. Folk & Zoellick (1992) points out that terminology is ambiguous because the maximum number of keys is not clear. An order 3 B-tree might hold a maximum of 6 keys or a maximum of 7 keys. Knuth (1998, p. 483) avoids the problem by defining the **order** to be maximum number of children (which is one more than the maximum number of keys).

The term **leaf** is also inconsistent. Bayer & McCreight (1972) considered the leaf level to be the lowest level of keys, but Knuth considered the leaf level to be one level

below the lowest keys (Folk & Zoellick 1992, p. 363). There are many possible implementation choices. In some designs, the leaves may hold the entire data record; in other designs, the leaves may only hold pointers to the data record. Those choices are not fundamental to the idea of a B-tree.<sup>[5]</sup>

There are also unfortunate choices like using the variable  $k$  to represent the number of children when  $k$  could be confused with the number of keys.

For simplicity, most authors assume there are a fixed number of keys that fit in a node. The basic assumption is the key size is fixed and the node size is fixed. In practice, variable length keys may be employed (Folk & Zoellick 1992, p. 379).

### Definition

According to Knuth's definition, a B-tree of order  $m$  is a tree which satisfies the following properties:

1. Every node has at most  $m$  children.
2. Every non-leaf node (except root) has at least  $\lceil m/2 \rceil$  children.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with  $k$  children contains  $k-1$  keys.
5. All leaves appear in the same level

Each internal node's keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys:  $a_1$  and  $a_2$ . All values in the leftmost subtree will be less than  $a_1$ , all values in the middle subtree will be between  $a_1$  and  $a_2$ , and all values in the rightmost subtree will be greater than  $a_2$ .

**Internal nodes** Internal nodes are all nodes except for leaf nodes and the root node. They are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of  $U$  children and a **minimum** of  $L$  children. Thus, the number of elements is always 1 less than the number of child pointers (the number of elements is between  $L-1$  and  $U-1$ ).  $U$  must be either  $2L$  or  $2L-1$ ; therefore each internal node is at least half full. The relationship between  $U$  and  $L$  implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there's room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

**The root node** The root node's number of children has the same upper limit as internal nodes, but has no lower limit. For example, when there are fewer than  $L-1$  elements in the entire tree, the root will be the only node in the tree, with no children at all.

**Leaf nodes** Leaf nodes have the same restriction on the number of elements, but have no children, and no child pointers.

A B-tree of depth  $n+1$  can hold about  $U$  times as many items as a B-tree of depth  $n$ , but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

Some balanced trees store values only at leaf nodes, and use different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree, and may use the same structure for all nodes. However, since leaf nodes never have children, the B-trees benefit from improved performance if they use a specialized structure.

### 6.1.4 Best case and worst case heights

Let  $h$  be the height of the classic B-tree. Let  $n > 0$  be the number of entries in the tree.<sup>[6]</sup> Let  $m$  be the maximum number of children a node can have. Each node can have at most  $m-1$  keys.

It can be shown (by induction for example) that a B-tree of height  $h$  with all its nodes completely filled has  $n = m^h - 1$  entries. Hence, the best case height of a B-tree is:

$$\lceil \log_m(n+1) \rceil.$$

Let  $d$  be the minimum number of children an internal (non-root) node can have. For an ordinary B-tree,  $d = \lceil m/2 \rceil$ .

Comer (1979, p. 127) and Cormen et al. (2001, pp. 383–384) give the worst case height of a B-tree (where the root node is considered to have height 0) as

$$h \leq \left\lceil \log_d \left( \frac{n+1}{2} \right) \right\rceil.$$

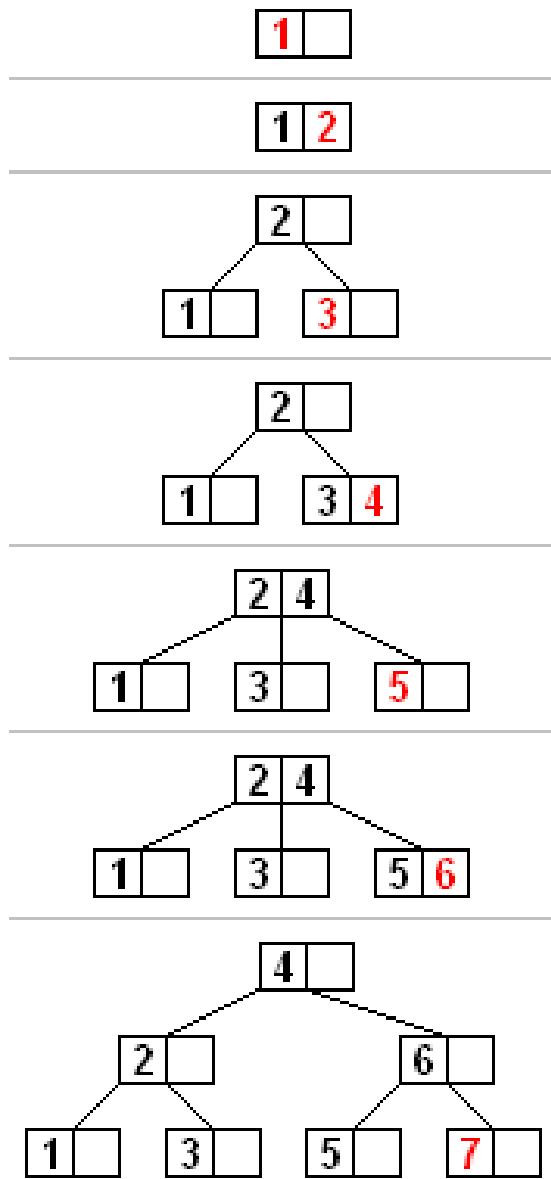
### 6.1.5 Algorithms

#### Search

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search chooses the child pointer (subtree) whose separation values are on either side of the search value.

Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest.

## Insertion



A B Tree insertion example with each iteration. The nodes of this B tree have at most 3 children (Knuth order 3).

All insertions start at a leaf node. To insert a new element, search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:

1. If the node contains fewer than the maximum legal number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.

2. Otherwise the node is full, evenly split it into two nodes so:

- (a) A single median is chosen from among the leaf's elements and the new element.
- (b) Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
- (c) The separation value is inserted in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

If the splitting goes all the way up to the root, it creates a new root with a single separator value and two children, which is why the lower bound on the size of internal nodes does not apply to the root. The maximum number of elements per node is  $U-1$ . When a node is split, one element moves to the parent, but one element is added. So, it must be possible to divide the maximum number  $U-1$  of elements into two legal nodes. If this number is odd, then  $U=2L$  and one of the new nodes contains  $(U-2)/2 = L-1$  elements, and hence is a legal node, and the other contains one more element, and hence it is legal too. If  $U-1$  is even, then  $U=2L-1$ , so there are  $2L-2$  elements in the node. Half of this number is  $L-1$ , which is the minimum number of elements allowed per node.

An improved algorithm (Mond & Raz 1985) supports a single pass down the tree from the root to the node where the insertion will take place, splitting any full nodes encountered on the way. This prevents the need to recall the parent nodes into memory, which may be expensive if the nodes are on secondary storage. However, to use this improved algorithm, we must be able to send one element to the parent and split the remaining  $U-2$  elements into two legal nodes, without adding a new element. This requires  $U = 2L$  rather than  $U = 2L-1$ , which accounts for why some textbooks impose this requirement in defining B-trees.

## Deletion

There are two popular strategies for deletion from a B-tree.

1. Locate and delete the item, then restructure the tree to regain its invariants, **OR**
2. Do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring

The algorithm below uses the former strategy.

There are two special cases to consider when deleting an element:

1. The element in an internal node is a separator for its child nodes
2. Deleting an element may put its node under the minimum number of elements and children

The procedures for these cases are in order below.

### Deletion from a leaf node

1. Search for the value to delete.
2. If the value is in a leaf node, simply delete it from the node.
3. If underflow happens, rebalance the tree as described in section “Rebalancing after deletion” below.

**Deletion from an internal node** Each element in an internal node acts as a separation value for two subtrees, therefore we need to find a replacement for separation. Note that the largest element in the left subtree is still less than the separator. Likewise, the smallest element in the right subtree is still greater than the separator. Both of those elements are in leaf nodes, and either one can be the new separator for the two subtrees. Algorithmically described below:

1. Choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree), remove it from the leaf node it is in, and replace the element to be deleted with the new separator.
2. The previous step deleted an element (the new separator) from a leaf node. If that leaf node is now deficient (has fewer than the required number of nodes), then rebalance the tree starting from the leaf node.

**Rebalancing after deletion** Rebalancing starts from a leaf and proceeds toward the root until the tree is balanced. If deleting an element from a node has brought it under the minimum size, then some elements must be redistributed to bring all nodes up to the minimum. Usually, the redistribution involves moving an element from a sibling node that has more than the minimum number of nodes. That redistribution operation is called a **rotation**. If no sibling can spare an element, then the deficient node must be **merged** with a sibling. The merge causes the parent to lose a separator element, so the parent may become deficient and need rebalancing. The merging and rebalancing may continue all the way to the root. Since the minimum element count doesn't apply to the root, making the root be the only deficient node is not a problem. The algorithm to rebalance the tree is as follows:

- If the deficient node's right sibling exists and has more than the minimum number of elements, then rotate left
  1. Copy the separator from the parent to the end of the deficient node (the separator moves down; the deficient node now has the minimum number of elements)
  2. Replace the separator in the parent with the first element of the right sibling (right sibling loses one node but still has at least the minimum number of elements)
  3. The tree is now balanced
- Otherwise, if the deficient node's left sibling exists and has more than the minimum number of elements, then rotate right
  1. Copy the separator from the parent to the start of the deficient node (the separator moves down; deficient node now has the minimum number of elements)
  2. Replace the separator in the parent with the last element of the left sibling (left sibling loses one node but still has at least the minimum number of elements)
  3. The tree is now balanced
- Otherwise, if both immediate siblings have only the minimum number of elements, then merge with a sibling sandwiching their separator taken off from their parent
  1. Copy the separator to the end of the left node (the left node may be the deficient node or it may be the sibling with the minimum number of elements)
  2. Move all elements from the right node to the left node (the left node now has the maximum number of elements, and the right node – empty)
  3. Remove the separator from the parent along with its empty right child (the parent loses an element)
    - If the parent is the root and now has no elements, then free it and make the merged node the new root (tree becomes shallower)
    - Otherwise, if the parent has fewer than the required number of elements, then rebalance the parent

**Note:** The rebalancing operations are different for B+ trees (e.g., rotation is different because parent has copy of the key) and B\*-tree (e.g., three siblings are merged into two siblings).

## Sequential access

While freshly loaded databases tend to have good sequential behavior, this behavior becomes increasingly difficult to maintain as a database grows, resulting in more random I/O and performance challenges.<sup>[7]</sup>

## Initial construction

In applications, it is frequently useful to build a B-tree to represent a large existing collection of data and then update it incrementally using standard B-tree operations. In this case, the most efficient way to construct the initial B-tree is not to insert every element in the initial collection successively, but instead to construct the initial set of leaf nodes directly from the input, then build the internal nodes from these. This approach to B-tree construction is called **bulkloading**. Initially, every leaf but the last one has one extra element, which will be used to build the internal nodes.

For example, if the leaf nodes have maximum size 4 and the initial collection is the integers 1 through 24, we would initially construct 4 leaf nodes containing 5 values each and 1 which contains 4 values:

We build the next level up from the leaves by taking the last element from each leaf node except the last one. Again, each node except the last will contain one extra value. In the example, suppose the internal nodes contain at most 2 values (3 child pointers). Then the next level up of internal nodes would be:

This process is continued until we reach a level with only one node and it is not overfilled. In the example only the root level remains:

### 6.1.6 In filesystems

In addition to its use in databases, the B-tree is also used in filesystems to allow quick random access to an arbitrary block in a particular file. The basic problem is turning the file block  $i$  address into a disk block (or perhaps to a cylinder-head-sector) address.

Some operating systems require the user to allocate the maximum size of the file when the file is created. The file can then be allocated as contiguous disk blocks. Converting to a disk block: the operating system just adds the file block address to the starting disk block of the file. The scheme is simple, but the file cannot exceed its created size.

Other operating systems allow a file to grow. The resulting disk blocks may not be contiguous, so mapping logical blocks to physical blocks is more involved.

MS-DOS, for example, used a simple **File Allocation Table** (FAT). The FAT has an entry for each disk block,<sup>[note 1]</sup> and that entry identifies whether its block is

used by a file and if so, which block (if any) is the next disk block of the same file. So, the allocation of each file is represented as a **linked list** in the table. In order to find the disk address of file block  $i$ , the operating system (or disk utility) must sequentially follow the file's linked list in the FAT. Worse, to find a free disk block, it must sequentially scan the FAT. For MS-DOS, that was not a huge penalty because the disks and files were small and the FAT had few entries and relatively short file chains. In the **FAT12** filesystem (used on floppy disks and early hard disks), there were no more than 4,080<sup>[note 2]</sup> entries, and the FAT would usually be resident in memory. As disks got bigger, the FAT architecture began to confront penalties. On a large disk using FAT, it may be necessary to perform disk reads to learn the disk location of a file block to be read or written.

**TOPS-20** (and possibly **TENEX**) used a 0 to 2 level tree that has similarities to a B-tree. A disk block was 512 36-bit words. If the file fit in a 512 ( $2^9$ ) word block, then the file directory would point to that physical disk block. If the file fit in  $2^{18}$  words, then the directory would point to an aux index; the 512 words of that index would either be NULL (the block isn't allocated) or point to the physical address of the block. If the file fit in  $2^{27}$  words, then the directory would point to a block holding an aux-aux index; each entry would either be NULL or point to an aux index. Consequently, the physical disk block for a  $2^{27}$  word file could be located in two disk reads and read on the third.

Apple's filesystem **HFS+**, Microsoft's **NTFS**,<sup>[8]</sup> AIX (jfs2) and some **Linux** filesystems, such as **btrfs** and **Ext4**, use B-trees.

B\*-trees are used in the **HFS** and **Reiser4** file systems.

### 6.1.7 Variations

#### Access concurrency

Lehman and Yao<sup>[9]</sup> showed that all read locks could be avoided (and thus concurrent access greatly improved) by linking the tree blocks at each level together with a “next” pointer. This results in a tree structure where both insertion and search operations descend from the root to the leaf. Write locks are only required as a tree block is modified. This maximizes access concurrency by multiple users, an important consideration for databases and/or other B-tree based **ISAM** storage methods. The cost associated with this improvement is that empty pages cannot be removed from the btree during normal operations. (However, see<sup>[10]</sup> for various strategies to implement node merging, and source code at.<sup>[11]</sup>)

United States Patent 5283894, granted in 1994, appears to show a way to use a 'Meta Access Method'<sup>[12]</sup> to allow concurrent B+ tree access and modification without locks. The technique accesses the tree 'upwards' for both searches and updates by means of additional in-memory

indexes that point at the blocks in each level in the block cache. No reorganization for deletes is needed and there are no 'next' pointers in each block as in Lehman and Yao.

### 6.1.8 See also

- R-tree
- 2-3 tree
- 2-3-4 tree

### 6.1.9 Notes

- [1] For FAT, what is called a “disk block” here is what the FAT documentation calls a “cluster”, which is fixed-size group of one or more contiguous whole physical disk sectors. For the purposes of this discussion, a cluster has no significant difference from a physical sector.
- [2] Two of these were reserved for special purposes, so only 4078 could actually represent disk blocks (clusters).

### 6.1.10 References

- [1] Counted B-Trees, retrieved 2010-01-25
- [2] Knuth's video lectures from Stanford
- [3] Talk's video, retrieved 2014-01-17
- [4] Seagate Technology LLC, Product Manual: Barracuda ES.2 Serial ATA, Rev. F., publication 100468393, 2008 , page 6
- [5] Bayer & McCreight (1972) avoided the issue by saying an index element is a (physically adjacent) pair of  $(x, a)$  where  $x$  is the key, and  $a$  is some associated information. The associated information might be a pointer to a record or records in a random access, but what it was didn't really matter. Bayer & McCreight (1972) states, “For this paper the associated information is of no further interest.”
- [6] If  $n$  is zero, then no root node is needed, so the height of an empty tree is not well defined.
- [7] “Cache Oblivious B-trees”. State University of New York (SUNY) at Stony Brook. Retrieved 2011-01-17.
- [8] Mark Russinovich. “Inside Win2K NTFS, Part 1”. Microsoft Developer Network. Archived from the original on 13 April 2008. Retrieved 2008-04-18.
- [9] “Efficient locking for concurrent operations on B-trees”. Portal.acm.org. doi:10.1145/319628.319663. Retrieved 2012-06-28.
- [10] <http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA232287&Location=U2&doc=GetTRDoc.pdf>
- [11] “Downloads - high-concurrency-btree - High Concurrency B-Tree code in C - GitHub Project Hosting”. Retrieved 2014-01-27.

- [12] Lockless Concurrent B+Tree

### General

- Bayer, R.; McCreight, E. (1972), “Organization and Maintenance of Large Ordered Indexes” (PDF), *Acta Informatica* 1 (3): 173–189, doi:10.1007/bf00288683
- Comer, Douglas (June 1979), “The Ubiquitous B-Tree”, *Computing Surveys* 11 (2): 123–137, doi:10.1145/356770.356776, ISSN 0360-0300.
- Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2001), *Introduction to Algorithms* (Second ed.), MIT Press and McGraw-Hill, pp. 434–454, ISBN 0-262-03293-7. Chapter 18: B-Trees.
- Folk, Michael J.; Zoellick, Bill (1992), *File Structures* (2nd ed.), Addison-Wesley, ISBN 0-201-55713-4
- Knuth, Donald (1998), *Sorting and Searching, The Art of Computer Programming*, Volume 3 (Second ed.), Addison-Wesley, ISBN 0-201-89685-0. Section 6.2.4: Multiway Trees, pp. 481–491. Also, pp. 476–477 of section 6.2.3 (Balanced Trees) discusses 2-3 trees.
- Mond, Yehudit; Raz, Yoav (1985), “Concurrency Control in B+-Trees Databases Using Preparatory Operations”, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases*: 331–334.

### Original papers

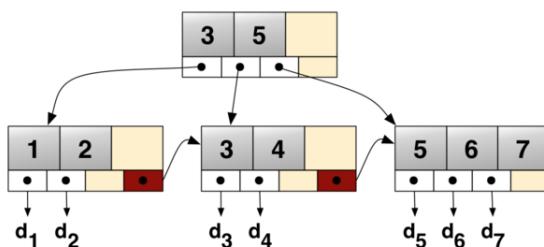
- Bayer, Rudolf; McCreight, E. (July 1970), *Organization and Maintenance of Large Ordered Indices*, Mathematical and Information Sciences Report No. 20, Boeing Scientific Research Laboratories.
- Bayer, Rudolf (1971), “Binary B-Trees for Virtual Memory”, Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California Missing or empty |title= (help). November 11–12, 1971.

### 6.1.11 External links

- B-tree lecture by David Scot Taylor, SJSU
- B-Tree animation applet by slady
- B-tree and UB-tree on Scholarpedia Curator: Dr Rudolf Bayer
- B-Trees: Balanced Tree Data Structures

- NIST's Dictionary of Algorithms and Data Structures: B-tree
- B-Tree Tutorial
- The InfinityDB BTree implementation
- Cache Oblivious B(+)‐trees
- Dictionary of Algorithms and Data Structures entry for B\*‐tree
- Open Data Structures - Section 14.2 - B-Trees
- Counted B-Trees

## 6.2 B+ tree



A simple B+ tree example linking the keys 1–7 to data values  $d_1$ – $d_7$ . The linked list (red) allows rapid in-order traversal. This particular tree's branching factor is  $b=4$ .

A **B+ tree** is an n-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves.<sup>[1]</sup> The root may be either a leaf or a node with two or more children.<sup>[2]</sup>

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout (number of pointers to child nodes in a node,<sup>[1]</sup> typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

The NTFS, ReiserFS, NSS, XFS, JFS, ReFS, and BFS filesystems all use this type of tree for metadata indexing; BFS also uses B+ trees for storing directories.<sup>[3]</sup> Relational database management systems such as IBM DB2,<sup>[4]</sup> Informix,<sup>[4]</sup> Microsoft SQL Server,<sup>[4]</sup> Oracle 8,<sup>[4]</sup> Sybase ASE,<sup>[4]</sup> and SQLite<sup>[5]</sup> support this type of tree for table indices. Key-value database management systems such as CouchDB<sup>[6]</sup> and Tokyo Cabinet<sup>[7]</sup> support this type of tree for data access.

### 6.2.1 Overview

The order, or branching factor,  $b$  of a B+ tree measures the capacity of nodes (i.e., the number of children nodes) for internal nodes in the tree. The actual number of children for a node, referred to here as  $m$ , is constrained for internal nodes so that  $\lceil b/2 \rceil \leq m \leq b$ . The root is an exception: it is allowed to have as few as two children.<sup>[1]</sup> For example, if the order of a B+ tree is 7, each internal node (except for the root) may have between 4 and 7 children; the root may have between 2 and 7. Leaf nodes have no children, but are constrained so that the number of keys must be at least  $\lceil b/2 \rceil$  and at most  $b - 1$ . In the situation where a B+ tree is nearly empty, it only contains one node, which is a leaf node. (The root is also the single leaf, in this case.) This node is permitted to have as little as one key if necessary, and at most  $b$ .

### 6.2.2 Algorithms

#### Search

The root of a B+ Tree represents the whole range of values in the tree, where every internal node is a subinterval.

We are looking for a value  $k$  in the B+ Tree. Starting from the root, we are looking for the leaf which may contain the value  $k$ . At each node, we figure out which internal pointer we should follow. An internal B+ Tree node has at most  $d \leq b$  children, where every one of them represents a different sub-interval. We select the corresponding node by searching on the key values of the node.

**Function:** `search (k) return tree_search (k, root);` **Function:** `tree_search (k, node)` **if** node is a leaf **then return** node; **switch**  $k$  **do** **case**  $k < k_0$  **return** `tree_search(k, p_0)`; **case**  $k_i \leq k < k_{i+1}$  **return** `tree_search(k, p_{i+1})`; **case**  $k_d \leq k$  **return** `tree_search(k, p_{d+1})`;

This pseudocode assumes that no duplicates are allowed.

#### prefix key compression

- It is important to increase fan-out, as this allows to direct searches to the leaf level more efficiently.
- Index Entries are only to 'direct traffic', thus we can compress them.

#### Insertion

Perform a search to determine what bucket the new record should go into.

- If the bucket is not full (at most  $b - 1$  entries after the insertion), add the record.
- Otherwise, split the bucket.

- Allocate new leaf and move half the bucket's elements to the new bucket.
- Insert the new leaf's smallest key and address into the parent.
- If the parent is full, split it too.
  - Add the middle key to the parent node.
- Repeat until a parent is found that need not split.
- If the root splits, create a new root which has one key and two pointers. (That is, the value that gets pushed to the new root gets removed from the original node)

B-trees grow at the root and not at the leaves.<sup>[1]</sup>

### Deletion

- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has fewer entries than it should,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

### merging propagates to sink

- But ... occupancy Factor of L dropped below 50% (d=2) which is not acceptable.
- Thus, L needs to be either
  - i) merged with its sibling
  - or ii) redistributed with its sibling

### Bulk-loading

Given a collection of data records, we want to create a B+ tree index on some key field. One approach is to insert each record into an empty tree. However, it is quite expensive, because each entry requires us to start from the root and go down to the appropriate leaf page. An efficient alternative is to use bulk-loading.

- The first step is to sort the data entries according to a search key in ascending order.

- We allocate an empty page to serve as the root, and insert a pointer to the first page of entries into it.
- When the root is full, we split the root, and create a new root page.
- Keep inserting entries to the right most index page just above the leaf level, until all entries are indexed.

Note (1) when the right-most index page above the leaf level fills up, it is split; (2) this action may, in turn, cause a split of the right-most index page on step closer to the root; and (3) splits only occur on the right-most path from the root to the leaf level.

### 6.2.3 Characteristics

For a  $b$ -order B+ tree with  $h$  levels of index:

- The maximum number of records stored is  $n_{max} = b^h - b^{h-1}$
- The minimum number of records stored is  $n_{min} = 2 \lceil \frac{b}{2} \rceil^{h-1}$
- The minimum number of keys is  $n_{kmin} = 2 \lceil \frac{b}{2} \rceil^h - 1$
- The maximum number of keys is  $n_{kmax} = n^h$
- The space required to store the tree is  $O(n)$
- Inserting a record requires  $O(\log_b n)$  operations
- Finding a record requires  $O(\log_b n)$  operations
- Removing a (previously located) record requires  $O(\log_b n)$  operations
- Performing a range query with  $k$  elements occurring within the range requires  $O(\log_b n + k)$  operations

### 6.2.4 Implementation

The leaves (the bottom-most index blocks) of the B+ tree are often linked to one another in a linked list; this makes range queries or an (ordered) iteration through the blocks simpler and more efficient (though the aforementioned upper bound can be achieved even without this addition). This does not substantially increase space consumption or maintenance on the tree. This illustrates one of the significant advantages of a B+tree over a B-tree; in a B-tree, since not all keys are present in the leaves, such an ordered linked list cannot be constructed. A B+tree is thus particularly useful as a database system index, where the data typically resides on disk, as it allows the B+tree to actually provide an efficient structure for housing the data itself (this is described in [6] as index structure "Alternative 1").

If a storage system has a block size of  $B$  bytes, and the keys to be stored have a size of  $k$ , arguably the most efficient B+ tree is one where  $b = (B/k) - 1$ . Although theoretically the one-off is unnecessary, in practice there is often a little extra space taken up by the index blocks (for example, the linked list references in the leaf blocks). Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease; therefore erring on the side of caution is preferable.

If nodes of the B+ tree are organized as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array.

B+ trees can also be used for data stored in RAM. In this case a reasonable choice for block size would be the size of processor's cache line.

Space efficiency of B+ trees can be improved by using some compression techniques. One possibility is to use delta encoding to compress keys stored into each block. For internal blocks, space saving can be achieved by either compressing keys or pointers. For string keys, space can be saved by using the following technique: Normally the  $i$ -th entry of an internal block contains the first key of block  $i+1$ . Instead of storing the full key, we could store the shortest prefix of the first key of block  $i+1$  that is strictly greater (in lexicographic order) than last key of block  $i$ . There is also a simple way to compress pointers: if we suppose that some consecutive blocks  $i, i+1\dots i+k$  are stored contiguously, then it will suffice to store only a pointer to the first block and the count of consecutive blocks.

All the above compression techniques have some drawbacks. First, a full block must be decompressed to extract a single element. One technique to overcome this problem is to divide each block into sub-blocks and compress them separately. In this case searching or inserting an element will only need to decompress or compress a sub-block instead of a full block. Another drawback of compression techniques is that the number of stored elements may vary considerably from a block to another depending on how well the elements are compressed inside each block.

## 6.2.5 History

The B tree was first described in the paper *Organization and Maintenance of Large Ordered Indices*. *Acta Informatica 1*: 173–189 (1972) by Rudolf Bayer and Edward M. McCreight. There is no single paper introducing the B+ tree concept. Instead, the notion of maintaining all data in leaf nodes is repeatedly brought up as an interesting variant. An early survey of B trees also covering B+ trees is Douglas Comer: "The Ubiquitous B-Tree", ACM

Computing Surveys 11(2): 121–137 (1979). Comer notes that the B+ tree was used in IBM's VSAM data access software and he refers to an IBM published article from 1973.

## 6.2.6 See also

- Binary Search Tree
- B-tree
- Divide and conquer algorithm

## 6.2.7 References

- [1] Navathe, Ramez Elmasri, Shamkant B. (2010). *Fundamentals of database systems* (6th ed. ed.). Upper Saddle River, N.J.: Pearson Education. pp. 652–660. ISBN 9780136086208.
- [2] <http://www.seanster.com/BplusTree/BplusTree.html>
- [3] Giampaolo, Dominic (1999). *Practical File System Design with the Be File System* (PDF). Morgan Kaufmann. ISBN 1-55860-497-9.
- [4] Ramakrishnan Raghu, Gehrke Johannes - Database Management Systems, McGraw-Hill Higher Education (2000), 2nd edition (en) page 267
- [5] SQLite Version 3 Overview
- [6] CouchDB Guide (see note after 3rd paragraph)
- [7] Tokyo Cabinet reference

## 6.2.8 External links

- B+ tree in Python, used to implement a list
- Dr. Monge's B+ Tree index notes
- Evaluating the performance of CSB+-trees on Multithreaded Architectures
- Effect of node size on the performance of cache conscious B+-trees
- Fractal Prefetching B+-trees
- Towards pB+-trees in the field: implementations Choices and performance
- Cache-Conscious Index Structures for Main-Memory Databases
- Cache Oblivious B(+)-trees
- The Power of B-Trees: CouchDB B+ Tree Implementation
- B+ Tree Visualization

## Implementations

- Interactive B+ Tree Implementation in C
- Memory based B+ tree implementation as C++ template library
- Stream based B+ tree implementation as C++ template library
- Open Source JavaScript B+ Tree Implementation
- Perl implementation of B+ trees
- Java/C#/Python implementations of B+ trees
- File based B+Tree in C# with threading and MVCC support
- JavaScript B+ Tree, MIT License
- JavaScript B+ Tree, Interactive and Open Source

## 6.3 Dancing tree

In computer science, a **dancing tree** is a tree data structure similar to B+ trees. It was invented by Hans Reiser, for use by the Reiser4 file system. As opposed to **self-balancing binary search trees** that attempt to keep their nodes balanced at all times, dancing trees only balance their nodes when flushing data to a disk (either because of memory constraints or because a transaction has completed).<sup>[1]</sup>

The idea behind this is to speed up file system operations by delaying optimization of the tree and only writing to disk when necessary, as writing to disk is thousands of times slower than writing to memory. Also, because this optimization is done less often than with other tree data structures, the optimization can be more extensive.

In some sense, this can be considered to be a self-balancing binary search tree that is optimized for storage on a slow medium, in that the on-disc form will always be balanced but will get no mid-transaction writes; doing so eases the difficulty (at the time) of adding and removing nodes, and instead performs these (slow) rebalancing operations at the same time as the (much slower) write to the storage medium.

However, a (negative) side effect of this behavior is witnessed in cases of unexpected shutdown, incomplete data writes, and other occurrences that may prevent the final (balanced) transaction from completing. In general, dancing trees will pose a greater difficulty for data recovery from incomplete transactions than a normal tree; though this can be addressed by either adding extra transaction logs or developing an algorithm to locate data on disk not previously present, then going through with the optimizations once more before continuing with any other pending operations/transactions.

## 6.3.1 References

- [1] Hans Reiser. “Reiser4 release notes - Dancing Tree”. Archive.org, as Namesys.com is no longer accessible. Archived from the original on 2007-10-24. Retrieved 2009-07-22.

## 6.3.2 External links

- *Software Engineering Based Reiser4 Design Principles*
- Description of the Reiser4 internal tree

## 6.4 2-3 tree

In computer science, a **2-3 tree** is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. Nodes on the outside of the tree (leaf nodes) have no children and one or two data elements.<sup>[1][2]</sup> 2-3 trees were invented by John Hopcroft in 1970.<sup>[3]</sup>

- 2 node
- 3 node

2-3 trees are an **isometry** of **AA trees**, meaning that they are equivalent data structures. In other words, for every 2-3 tree, there exists at least one AA tree with data elements in the same order. 2-3 trees are balanced, meaning that each right, center, and left subtree contains the same or close to the same amount of data.

## 6.4.1 Definitions

We say that a node is a **2-node** if and only if it has *one* data element and *two* children if it is an internal node.

We say that a node is a **3-node** if and only if it has *two* data elements and *three* children if it is an internal node.

We say that T is a **2-3 tree** if and only if one of the following statements hold:

- T is empty. In other words, T does not have any nodes.
- T is a 2-node r with data element a. If r has left child L and right child R, then L and R are non-empty 2-3 trees of the same **height**, a is greater than each element in L, and a is less than each data element in R.
- T is a 3-node r with data elements a and b, where  $a < b$ . If r has left child L, middle child M, and

right child R, then L, M, and R are non-empty 2-3 trees of equal height, a is greater than each data element in L and less than each data element in M, and b is greater than each data element in M and less than each data element in R.

## 6.4.2 Properties

- Every internal node is a 2-node or a 3-node.
- All leaves are at the same level.
- All data is kept in sorted order.

## 6.4.3 Operations

### Searching

Searching for an item in a 2-3 tree is similar to searching for an item in a [binary search tree](#). Since the data elements in each node is ordered, a search function will be directed to the correct subtree and eventually to the correct node which contains the item.

1. Let T be a 2-3 tree and d be the data element we want to find. If T is empty, then d is not in T and we're done.
2. Let r be the root of T.
3. Suppose r is a leaf. If d is not in r, then d is not in T. Otherwise, d is in T. In particular, d can be found at a leaf node. We need no further steps and we're done.
4. Suppose r is a 2-node with left child L and right child R. Let e be the data element in r. There are three cases: If d is equal to e, then we've found d in T and we're done. If  $d < e$ , then set T to L, which by definition is a 2-3 tree, and go back to step 2. If  $d > e$ , then set T to R and go back to step 2.
5. Suppose r is a 3-node with left child L, middle child M, and right child R. Let a and b be the two data elements of r, where  $a < b$ . There are four cases: If d is equal to a or b, then d is in T and we're done. If  $d < a$ , then set T to L and go back to step 2. If  $a < d < b$ , then set T to M and go back to step 2. If  $d > b$ , then set T to R and go back to step 2.

### Insertion

Insertion works by searching for the proper location of the key and adds it there. If the node becomes a 4-node then the node is split from two 2-nodes and the middle key is moved up to the parent. The diagram illustrates the process.

## 6.4.4 See also

- 2-3-4 tree
- Finger tree
- 2-3 heap
- (a,b)-tree

## 6.4.5 References

- [1] Gross, R. Hernández, J. C. Lázaro, R. Dormido, S. Ros (2001). *Estructura de Datos y Algoritmos*. Prentice Hall. ISBN 84-205-2980-X.
- [2] Aho, Alfred V.; Hopcroft, John E.; Ullman, Jeffrey D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley., p.145-147
- [3] Cormen, Thomas (2009). *Introduction to Algorithms*. London: The MIT Press. p. 504. ISBN 978-0262033848.

## 6.4.6 External links

- 2-3 Tree Java Applet
- 2-3 Tree In-depth description
- 2-3 Tree in F#
- 2-3 Tree in Python

## 6.5 2-3-4 tree

In computer science, a **2-3-4 tree** (also called a **2-4 tree**) is a self-balancing [data structure](#) that is commonly used to implement [dictionaries](#). The numbers mean a [tree](#) where every [node](#) with children ([internal node](#)) has either two, three, or four child nodes:

- a 2-node has one [data element](#), and if internal has two child nodes;
  - a 3-node has two data elements, and if internal has three child nodes;
  - a 4-node has three data elements, and if internal has four child nodes.
- 2-node
  - 3-node
  - 4-node

2-3-4 trees are B-trees of order 4;<sup>[1]</sup> like B-trees in general, they can search, insert and delete in  $O(\log n)$  time. One property of a 2-3-4 tree is that all external nodes are at the same depth.

2-3-4 trees are an **isometry** of **red-black trees**, meaning that they are equivalent data structures. In other words, for every 2-3-4 tree, there exists at least one red-black tree with data elements in the same order. Moreover, insertion and deletion operations on 2-3-4 trees that cause node expansions, splits and merges are equivalent to the color-flipping and rotations in red-black trees. Introductions to red-black trees usually introduce 2-3-4 trees first, because they are conceptually simpler. 2-3-4 trees, however, can be difficult to implement in most programming languages because of the large number of special cases involved in operations on the tree. **Red-black trees** are simpler to implement,<sup>[2]</sup> so tend to be used instead.

## 6.5.1 Properties

- Every node (leaf or internal) is a 2-node, 3-node or a 4-node, and holds one, two, or three data elements, respectively.
- All leaves are at the same depth (the bottom level).
- All data is kept in sorted order.

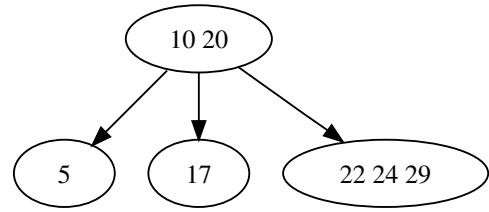
## 6.5.2 Insertion

To insert a value, we start at the root of the 2-3-4 tree:

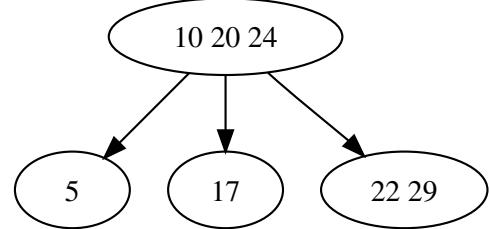
1. If the current node is a 4-node:
  - Remove and save the middle value to get a 3-node.
  - Split the remaining 3-node up into a pair of 2-nodes (the now missing middle value is handled in the next step).
  - If this is the root node (which thus has no parent):
    - the middle value becomes the new root 2-node and the tree height increases by 1. Ascend into the root.
    - Otherwise, push the middle value up into the parent node. Ascend into the parent node.
2. Find the child whose interval contains the value to be inserted.
3. If that child is a leaf, insert the value into the child node and finish.
  - Otherwise, descend into the child and repeat from step 1.<sup>[3][4]</sup>

## Example

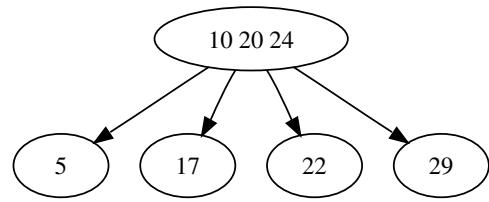
To insert the value “25” into this 2-3-4 tree:



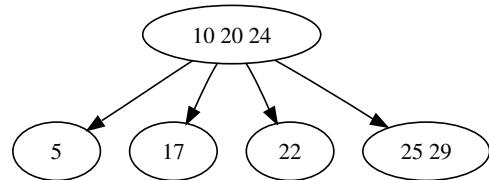
- Begin at the root (10, 20) and descend towards the rightmost child (22, 24, 29). (Its interval  $(20, \infty)$  contains 25.)
- Node (22, 24, 29) is a 4-node, so its middle element 24 is pushed up into the parent node.



- The remaining 3-node (22, 29) is split into a pair of 2-nodes (22) and (29). Ascend back into the new parent (10, 20, 24).
- Descend towards the rightmost child (29). (Its interval  $(24 - 29)$  contains 25.)



- Node (29) has no leftmost child. (The child for interval  $(24 - 29)$  is empty.) Stop here and insert value 25 into this node.



## 6.5.3 Deletion

Consider just leaving the element there, marking it “deleted,” possibly to be re-used for a future insertion.

To remove a value from the 2-3-4 tree:

### 1. Find the element to be deleted.

- If the element is not in a leaf node, remember its location and continue searching until a leaf, which will contain the element's successor, is reached. The successor can be either the largest key that is smaller than the one to be removed, or the smallest key that is larger than the one to be removed. It is simplest to make adjustments to the tree from the top down such that the leaf node found is not a 2-node. That way, after the swap, there will not be an empty leaf node.
- If the element is in a 2-node leaf, just make the adjustments below.

Make the following adjustments when a 2-node – except the root node – is encountered on the way to the leaf we want to remove:

1. If a sibling on either side of this node is a 3-node or a 4-node (thus having more than 1 key), perform a rotation with that sibling:
  - The key from the other sibling closest to this node moves up to the parent key that overlooks the two nodes.
  - The parent key moves down to this node to form a 3-node.
  - The child that was originally with the rotated sibling key is now this node's additional child.
2. If the parent is a 2-node and the sibling is also a 2-node, combine all three elements to form a new 4-node and shorten the tree. (This rule can only trigger if the parent 2-node is the root, since all other 2-nodes along the way will have been modified to not be 2-nodes. This is why “shorten the tree” here preserves balance; this is also an important assumption for the fusion operation.)
3. If the parent is a 3-node or a 4-node and all adjacent siblings are 2-nodes, do a fusion operation with the parent and an adjacent sibling:
  - The adjacent sibling and the parent key overlooking the two sibling nodes come together to form a 4-node.
  - Transfer the sibling's children to this node.

Once the sought value is reached, it can now be placed at the removed entry's location without a problem because we have ensured that the leaf node has more than 1 key.

Deletion in a 2-3-4 tree is  $O(\log n)$ , assuming transfer and fusion run in constant time ( $O(1)$ ).<sup>[3][5]</sup>

### 6.5.4 See also

- 2-3 tree
- Red-black tree
- B-tree

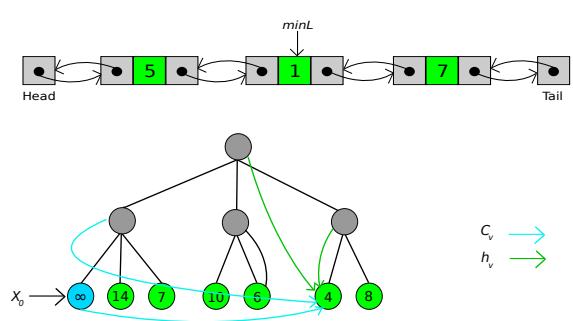
### 6.5.5 References

- [1] Knuth, Donald (1998). *Sorting and Searching. The Art of Computer Programming*. Volume 3 (Second ed.). Addison-Wesley. ISBN 0-201-89685-0.. Section 6.2.4: Multiway Trees, pp. 481–491. Also, pp. 476–477 of section 6.2.3 (Balanced Trees) discusses 2-3 trees.
- [2] Sedgewick, Robert (2008). “Left-Leaning Red-Black Trees” (PDF). *Left-Leaning Red-Black Trees*. Department of Computer Science, Purdue University.
- [3] Ford, William; Topp, William (2002), *Data Structures with C++ Using STL* (2nd ed.), New Jersey: Prentice Hall, p. 683, ISBN 0-13-085850-1
- [4] Goodrich, Michael T; Tamassia, Roberto; Mount, David M (2002), *Data Structures and Algorithms in C++*, Wiley, ISBN 0-471-20208-8
- [5] Grama, Ananth (2004). “(2,4) Trees” (PDF). *CS251: Data Structures Lecture Notes*. Department of Computer Science, Purdue University. Retrieved 2008-04-10.

### 6.5.6 External links

- Algorithms In Action, with 2-3-4 Tree animation
- Animation of a 2-3-4 Tree
- Java Applet showing a 2-3-4 Tree
- Left-leaning Red-Black Trees – Princeton University, 2008
- Open Data Structures – Section 9.1 – 2-4 Trees

## 6.6 Queaps



A Queap  $Q$  with  $k = 6$  and  $n = 9$

In computer science, a **queap** is a priority queue data structure. The data structure allows insertions and deletions of arbitrary elements, as well as retrieval of the highest-priority element. Each deletion takes amortized time logarithmic in the number of items that have been in the structure for a longer time than the removed item. Insertions take constant amortized time.

The data structure consists of a **doubly linked list** and a **2-4 tree** data structure, each modified to keep track of its minimum-priority element. The basic operation of the structure is to keep newly inserted elements in the doubly linked list, until a deletion would remove one of the list items, at which point they are all moved into the 2-4 tree. The 2-4 tree stores its elements in insertion order, rather than the more conventional priority-sorted order.

Both the data structure and its name were devised by John Iacono and Stefan Langerman.<sup>[1]</sup>

### 6.6.1 Description

A queap is a priority queue that inserts elements in  $O(1)$  amortized time, and removes the minimum element in  $O(\log(k + 2))$  if there are  $k$  items that have been in the heap for a longer time than the element to be extracted. The queap has a property called the queueish property: the time to search for element  $x$  is  $O(\lg q(x))$  where  $q(x)$  is equal to  $n - 1 - w(x)$  and  $w(x)$  is the number of distinct items that has been accessed by operations such as searching, inserting, or deleting.  $q(x)$  is defined as how many elements have not been accessed since  $x$ 's last access. Indeed, the queueish property is the complement of the splay tree working set property: the time to search for element  $x$  is  $O(\lg w(x))$ .

A queap can be represented by two data structures: a doubly linked list and a modified version of 2-4 tree. The doubly linked list,  $L$ , is used for a series of insert and locate-min operations. The queap keeps a pointer to the minimum element stored in the list. To add element  $x$  to list  $L$ , the element  $x$  is added to the end of the list and a bit variable in element  $x$  is set to one. This operation is done to determine if the element is either in the list or in a 2-4 tree.

A 2-4 tree is used when a delete operation occurs. If the item  $x$  is already in tree  $T$ , the item is removed using the 2-4 tree delete operation. Otherwise, the item  $x$  is in list  $L$  (done by checking if the bit variable is set). All the elements stored in list  $L$  are then added to the 2-4 tree, setting the bit variable of each element to zero.  $x$  is then removed from  $T$ .

A queap uses only the 2-4 tree structure properties, not a search tree. The modified 2-4 tree structure is as follows. Suppose list  $L$  has the following set of elements:  $x_1, x_2, x_3, \dots, x_k$ . When the deletion operation is invoked, the set of elements stored in  $L$  is then added to the leaves of the 2-4 tree in that order, proceeded by a dummy

leaf containing an infinite key. Each internal node of  $T$  has a pointer  $h_v$ , which points to the smallest item in subtree  $v$ . Each internal node on path  $P$  from the root to  $x_0$  has a pointer  $c_v$ , which points to the smallest key in  $T - T_v - \{r\}$ . The  $h_v$  pointers of each internal node on path  $P$  are ignored. The queap has a pointer to  $c_{x_0}$ , which points to the smallest element in  $T$ .

An application of queaps includes a unique set of high priority events and extraction of the highest priority event for processing.

### 6.6.2 Operations

Let  $minL$  be a pointer that points to the minimum element in the doubly linked list  $L$ ,  $c_{x_0}$  be the minimum element stored in the 2-4 tree,  $T$ ,  $k$  be the number of elements stored in  $T$ , and  $n$  be the total number of elements stored in queap  $Q$ . The operations are as follows:

**New( $Q$ )**: Initializes a new empty queap.

Initialize an empty doubly linked list  $L$  and 2-4 tree  $T$ . Set  $k$  and  $n$  to zero.

**Insert( $Q, x$ )**: Add the element  $x$  to queap  $Q$ .

Insert the element  $x$  in list  $L$ . Set the bit in element  $x$  to one to demonstrate that the element is in the list  $L$ . Update the  $minL$  pointer if  $x$  is the smallest element in the list. Increment  $n$  by 1.

**Minimum( $Q$ )**: Retrieve a pointer to the smallest element from queap  $Q$ .

If  $key(minL) < key(c_{x_0})$ , return  $minL$ . Otherwise return  $c_{x_0}$ .

**Delete( $Q, x$ )**: Remove element  $x$  from queap  $Q$ .

If the bit of the element  $x$  is set to one, the element is stored in list  $L$ . Add all the elements from  $L$  to  $T$ , setting the bit of each element to zero. Each element is added to the parent of the right most child of  $T$  using the insert operation of the 2-4 tree.  $L$  becomes empty. Update  $h_v$  pointers for all the nodes  $v$  whose children are new/modified, and repeat the process with the next parent until the parent is equal to the root. Walk from the root to node  $x_0$ , and update the  $c_v$  values. Set  $k$  equal to  $n$ .

If the bit of the element  $x$  is set to zero,  $x$  is a leaf of  $T$ . Delete  $x$  using the 2-4 tree delete operation. Starting from node  $x$ , walk in  $T$  to node  $x_0$ , updating  $h_v$  and  $c_v$  pointers. Decrement  $n$  and  $k$  by 1.

**DeleteMin( $Q$ ):** Delete and return the smallest element from queap  $Q$ .

Invoke the *Minimum( $Q$ )* operation. The operation returns  $min$ . Invoke the *Delete( $Q, min$ )* operation. Return  $min$ .

**CleanUp( $Q$ ):** Delete all the elements in list  $L$  and tree  $T$ .

Starting from the first element in list  $L$ , traverse the list, deleting each node.

Starting from the root of the tree  $T$ , traverse the tree using the *post-order traversal* algorithm, deleting each node in the tree.

### 6.6.3 Analysis

The running time is analyzed using the *amortized analysis*. The potential function for queap  $Q$  will be  $\phi(Q) = c|L|$  where  $Q = (T, L)$ .

**Insert( $Q, x$ ):** The cost of the operation is  $O(1)$ . The size of list  $L$  grows by one, the potential increases by some constant  $c$ .

**Minimum( $Q$ ):** The operation does not alter the data structure so the amortized cost is equal to its actual cost,  $O(1)$ .

**Delete( $Q, x$ ):** There are two cases.

#### Case 1

If  $x$  is in tree  $T$ , then the amortized cost is not modified. The delete operation is  $O(1)$  amortized 2-4 tree. Since  $x$  was removed from the tree,  $h_v$  and  $c_v$  pointers may need updating. At most, there will be  $O(lgq(x))$  updates.

#### Case 2

If  $x$  is in list  $L$ , then all the elements from  $L$  are inserted in  $T$ . This has a cost of  $a|L|$  of some constant  $a$ , amortized over the 2-4 tree. After inserting and updating the  $h_v$  and  $c_v$  pointers, the total time spent is bounded by  $2a|L|$ . The second operation is to delete  $x$  from  $T$ , and to walk on the path from  $x$  to  $x_0$ , correcting  $h_v$  and  $c_v$  values. The time is spent at most  $2a|L| + O(lgq(x))$ . If  $c > 2a$ , then the amortized cost will be  $O(lgq(x))$ . **Delete( $Q, x$ ):** is the addition of the amortized cost of **Minimum( $Q$ )** and **Delete( $Q, x$ )**, which is  $O(lgq(x))$ .

### 6.6.4 Code example

A small **java** implementation of a queap:

```
public class Queap { public int n, k; public
List<Element> l; //Element is a generic data type
```

```
public QueapTree t; //a 2-4 tree, modified for Queap
purpose public Element minL; private Queap() { n
= 0; k = 0; l = new LinkedList<Element>(); t = new
QueapTree(); } public static Queap New() { return
new Queap(); } public static void Insert(Queap Q,
Element x) { if (Q.n == 0) Q.minL = x; Q.l.add(x);
x.inList = true; if (x.compareTo(Q.minL) < 0) Q.minL
= x; } public static Element Minimum(Queap Q)
{ //t is a 2-4 tree and x0, cv are tree nodes. if
(Q.minL.compareTo(Q.t.x0.cv.key) < 0) return Q.minL;
return Q.t.x0.cv.key; } public static void Delete(Queap Q,
QueapNode x) { Q.t.deleteLeaf(x); --Q.n; --Q.k;
} public static void Delete(Queap Q, Element x) {
QueapNode n; if (x.inList) { //set inList of all the
elements in the list to false n = Q.t.insertList(Q.l, x);
Q.k = Q.n; Delete(Q, n); } else if ((n = Q.t.x0.cv).key == x)
Delete(Q, n); } public static Element DeleteMin(Queap Q) {
Element min = Minimum(Q); Delete(Q, min); return min; } }
```

### 6.6.5 See also

- Queue (data structure)
- Priority queue
- Splay tree
- 2-4 tree
- Doubly linked list
- Amortized analysis

### 6.6.6 References

- [1] John Iacono; Stefan Langerman (2005). “Queaps”. *Algorithmica* (Springer) **42** (1): 49–56. doi:10.1007/s00453-004-1139-5.

### 6.7 Fusion tree

A **fusion tree** is a type of **tree data structure** that implements an associative array on  $w$ -bit integers. It uses  $O(n)$  space and performs searches in  $O(\log w n)$  time, which is asymptotically faster than a traditional self-balancing binary search tree, and actually better than the van Emde Boas tree when  $w$  is large. It achieves this speed by exploiting certain constant-time operations that can be done on a machine word. Fusion trees were invented in 1990 by Michael Fredman and Dan Willard.<sup>[1]</sup>

Several advances have been made since Fredman and Willard’s original 1990 paper. In 1999<sup>[2]</sup> it was shown how to implement fusion trees under the  $AC^0$  model, in which multiplication no longer takes constant time. A dynamic version of fusion trees using **Hash tables** was proposed in 1996<sup>[3]</sup> which matched the  $O(\log w n)$  runtime in

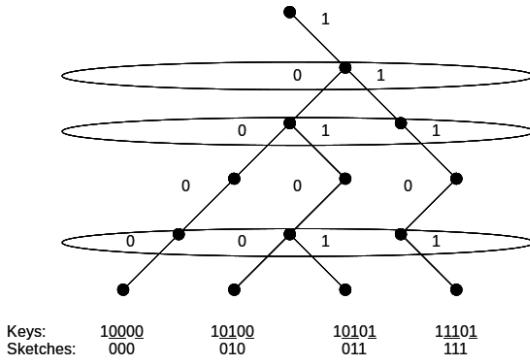
expectation. Another dynamic version using Exponential tree was proposed in 2007 [4] which yields worst-case runtimes of  $O(\log w n + \log \log u)$  per operation, where  $u$  is the size of the largest key. It remains open whether dynamic fusion trees can achieve  $O(\log w n)$  per operation with high probability.

### 6.7.1 How it works

A fusion tree is essentially a B-tree with branching factor of  $w^{1/5}$  (any small exponent is also possible), which gives it a height of  $O(\log w n)$ . To achieve the desired runtimes for updates and queries, the fusion tree must be able to search a node containing up to  $w^{1/5}$  keys in constant time. This is done by compressing (“sketching”) the keys so that all can fit into one machine word, which in turn allows comparisons to be done in parallel. The rest of this article will describe the operation of a static Fusion Tree; that is, only queries are supported.

### 6.7.2 Sketching

Sketching is the method by which each  $w$ -bit key at a node containing  $k$  keys is compressed into only  $k-1$  bits. Each key  $x$  may be thought of as a path in the full binary tree of height  $w$  starting at the root and ending at the leaf corresponding to  $x$ . To distinguish two paths, it suffices to look at their branching point (the first bit where the two keys differ). All  $k$  paths together have  $k-1$  branching points, so at most  $k-1$  bits are needed to distinguish any two of the  $k$  keys.



Visualization of the sketch function.

An important property of the sketch function is that it preserves the order of the keys. That is,  $\text{sketch}(x) < \text{sketch}(y)$  for any two keys  $x < y$ .

### 6.7.3 Approximating the sketch

If the locations of the sketch bits are  $b_1 < b_2 < \dots < b_r$ , then the sketch of the key  $x_{w-1} \dots x_1 x_0$  is the  $r$ -bit integer  $x_{b_r} x_{b_{r-1}} \dots x_{b_1}$ .

With only standard word operations, such as those of the C programming language, it is difficult to directly compute the sketch of a key in constant time. Instead, the sketch bits can be packed into a range of size at most  $r^4$ , using bitwise AND and multiplication. The bitwise AND operation serves to clear all non-sketch bits from the key, while the multiplication shifts the sketch bits into a small range. Like the “perfect” sketch, the approximate sketch preserves the order of the keys.

Some preprocessing is needed to determine the correct multiplication constant. Each sketch bit in location  $bi$  will get shifted to  $bi + mi$  via a multiplication by  $m = \sum_{i=1}^r 2^{mi}$ . For the approximate sketch to work, the following three properties must hold:

1.  $bi + mj$  are distinct for all pairs  $(i, j)$ . This will ensure that the sketch bits are uncorrupted by the multiplication.
2.  $bi + mj$  is a strictly increasing function of  $i$ . That is, the order of the sketch bits is preserved.
3.  $(br + mr) - (b_1 - m_1) \leq r^4$ . That is, the sketch bits are packed into a range of size at most  $r^4$ .

An inductive argument shows how the  $mi$  can be constructed. Let  $m_1 = w - b_1$ . Suppose that  $1 < t \leq r$  and that  $m_1, m_2, \dots, mt-1$  have already been chosen. Then pick the smallest integer  $mt$  such that both properties (1) and (2) are satisfied. Property (1) requires that  $mt \neq bi - bj + ml$  for all  $1 \leq i, j \leq r$  and  $1 \leq l \leq t-1$ . Thus, there are less than  $tr^2 \leq r^3$  values that  $mt$  must avoid. Since  $mt$  is chosen to be minimal,  $(bt + mt) \leq (bt-1 + mt-1) + r^3$ . This implies Property (3).

The approximate sketch is thus computed as follows:

1. Mask out all but the sketch bits with a bitwise AND.
2. Multiply the key by the predetermined constant  $m$ . This operation actually requires two machine words, but this can still be done in constant time.
3. Mask out all but the shifted sketch bits. These are now contained in a contiguous block of at most  $r^4 < w^{4/5}$  bits.

For the rest of this article, sketching will be taken to mean approximate sketching.

### 6.7.4 Parallel comparison

The purpose of the compression achieved by sketching is to allow all of the keys to be stored in one  $w$ -bit word. Let the *node sketch* of a node be the bit string

$$1\text{sketch}(x_1)1\text{sketch}(x_2)\dots1\text{sketch}(x_k)$$

We can assume that the sketch function uses exactly  $b \leq r^4$  bits. Then each block uses  $1 + b \leq w^{4/5}$  bits, and since  $k \leq w^{1/5}$ , the total number of bits in the node sketch is at most  $w$ .

A brief notational aside: for a bit string  $s$  and nonnegative integer  $m$ , let  $s^m$  denote the concatenation of  $s$  to itself  $m$  times. If  $t$  is also a bit string  $st$  denotes the concatenation of  $t$  to  $s$ .

The node sketch makes it possible to search the keys for any  $b$ -bit integer  $y$ . Let  $z = (0y)^k$ , which can be computed in constant time (multiply  $y$  by the constant  $(0^b 1)^k$ ). Note that  $\text{1sketch}(xi) - 0y$  is always positive, but preserves its leading 1 iff  $\text{sketch}(xi) \geq y$ . We can thus compute the smallest index  $i$  such that  $\text{sketch}(xi) \geq y$  as follows:

1. Subtract  $z$  from the node sketch.
2. Take the bitwise AND of the difference and the constant  $(10^b)^k$ . This clears all but the leading bit of each block.
3. Find the most significant bit of the result.
4. Compute  $i$ , using the fact that the leading bit of the  $i$ -th block has index  $i(b+1)$ .

## 6.7.5 Desketching

For an arbitrary query  $q$ , parallel comparison computes the index  $i$  such that

$$\text{sketch}(xi-1) \leq \text{sketch}(q) \leq \text{sketch}(xi)$$

Unfortunately, the sketch function is not in general order-preserving outside the set of keys, so it is not necessarily the case that  $xi-1 \leq q \leq xi$ . What is true is that, among all of the keys, either  $xi-1$  or  $xi$  has the longest common prefix with  $q$ . This is because any key  $y$  with a longer common prefix with  $q$  would also have more sketch bits in common with  $q$ , and thus  $\text{sketch}(y)$  would be closer to  $\text{sketch}(q)$  than any  $\text{sketch}(xj)$ .

The length longest common prefix between two  $w$ -bit integers  $a$  and  $b$  can be computed in constant time by finding the most significant bit of the **bitwise XOR** between  $a$  and  $b$ . This can then be used to mask out all but the longest common prefix.

Note that  $p$  identifies exactly where  $q$  branches off from the set of keys. If the next bit of  $q$  is 0, then the successor of  $q$  is contained in the  $p1$  subtree, and if the next bit of  $q$  is 1, then the predecessor of  $q$  is contained in the  $p0$  subtree. This suggests the following algorithm:

1. Use parallel comparison to find the index  $i$  such that  $\text{sketch}(xi-1) \leq \text{sketch}(q) \leq \text{sketch}(xi)$ .
2. Compute the longest common prefix  $p$  of  $q$  and either  $xi-1$  or  $xi$  (taking the longer of the two).

3. Let  $l-1$  be the length of the longest common prefix  $p$ .
  - (a) If the  $l$ -th bit of  $q$  is 0, let  $e = p10^{w-l}$ . Use parallel comparison to search for the successor of  $\text{sketch}(e)$ . This is the actual predecessor of  $q$ .
  - (b) If the  $l$ -th bit of  $q$  is 1, let  $e = p01^{w-l}$ . Use parallel comparison to search for the predecessor of  $\text{sketch}(e)$ . This is the actual successor of  $q$ .
4. Once either the predecessor or successor of  $q$  is found, the exact position of  $q$  among the set of keys is determined.

## 6.7.6 References

- [1] M. L. Fredman and D. E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. Proceedings of the twenty-second annual ACM symposium on Theory of Computing, 1-7, 1990.
- [2] A. Andersson, P. B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC0 instructions only. Theoretical Computer Science, 215:337-344, 1999.
- [3] R. Raman. Priority queues: Small, monotone, and trans-dichotomous. Algorithms - ESA '96, 121-137, 1996.
- [4] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. Journal of the ACM, 54:3:13, 2007.
- MIT CS 6.897: Advanced Data Structures: Lecture 4, Fusion Trees, Prof. Erik Demaine (Spring 2003)
- MIT CS 6.897: Advanced Data Structures: Lecture 5, More fusion trees; self-organizing data structures, move-to-front, static optimality, Prof. Erik Demaine (Spring 2003)
- MIT CS 6.851: Advanced Data Structures: Lecture 13, Fusion Tree notes, Prof. Erik Demaine (Spring 2007)
- MIT CS 6.851: Advanced Data Structures: Lecture 12, Fusion Tree notes, Prof. Erik Demaine (Spring 2012)

## 6.8 Bx-tree

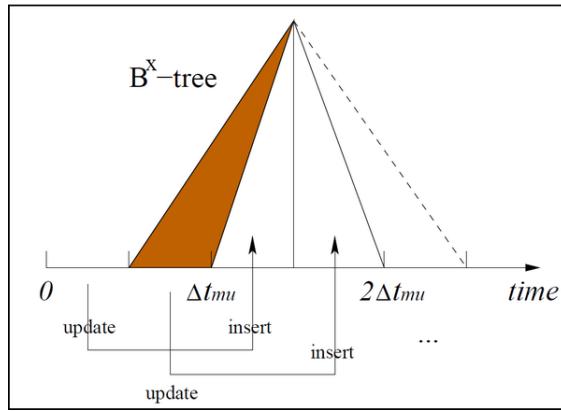
In computer science, the **B<sup>x</sup> tree** is a query and update efficient B+ tree-based index structure for moving objects.

### 6.8.1 Index structure

The base structure of the B<sup>x</sup>-tree is a B+ tree in which the internal nodes serve as a directory, each containing a

pointer to its right sibling. In the earlier version of the  $B^x$ -tree,<sup>[1]</sup> the **leaf nodes** contained the moving-object locations being indexed and corresponding index time. In the optimized version,<sup>[2]</sup> each leaf node entry contains the id, velocity, single-dimensional mapping value and the latest update time of the object. The fanout is increased by not storing the locations of moving objects, as these can be derived from the **mapping** values.

### 6.8.2 Utilizing the B+ tree for moving objects



An example of the  $B^x$ -tree with the number of index partitions equal to 2 within one maximum update interval  $tmu$ . In this example, there are maximum three partitions existing at the same time. After linearization, object locations inserted at time 0 are indexed in partition 0 with label timestamp 0.5  $tmu$ , object locations updated during time 0 to 0.5  $tmu$  are indexed in partition 1 with label timestamp  $tmu$ , and so on (as indicated by arrows). As time elapses, repeatedly the first range expires (shaded area), and a new range is appended (dashed line).

As for many other moving objects indexes, a two-dimensional moving object is modeled as a linear function as  $O = ((x, y), (vx, vy), t)$ , where  $(x, y)$  and  $(vx, vy)$  are location and **velocity** of the object at a given time instance  $t$ , i.e., the time of last update. The B+ tree is a structure for indexing single-dimensional data. In order to adopt the B+ tree as a moving object index, the  $B^x$ -tree uses a **linearization** technique which helps to integrate objects' location at time  $t$  into single dimensional value. Specifically, objects are first partitioned according to their update time. For objects within the same partition, the  $B^x$ -tree stores their locations at a given time which are estimated by **linear interpolation**. By doing so, the  $B^x$ -tree keeps a consistent view of all objects within the same partition without storing the update time of an objects.

Secondly, the space is partitioned by a grid and the location of an object is linearized within the partitions according to a space-filling curve, e.g., the **Peano** or **Hilbert** curves.

Finally, with the combination of the partition number

(time information) and the linear order (location information), an object is indexed in  $B^x$ -tree with a one-dimensional index key  $B^x$ value:

$$B^x\text{value}(O, t) = [\text{indexpartition}]_2 + [\text{xrep}]_2$$

Here index-partition is an index partition determined by the update time and xrep is the space-filling curve value of the object position at the indexed time,  $[X]_2$  denotes the binary value of x, and “+” means concatenation.

Given an object  $O ((7, 2), (-0.1, 0.05), 10)$ ,  $tmu = 120$ , the  $B^x$ value for  $O$  can be computed as follows.

1.  $O$  is indexed in partition 0 as mentioned. Therefore, indexpartition =  $(00)_2$ .
2.  $O$ 's position at the label timestamp of partition 0 is  $(1, 5)$ .
3. Using Z-curve with order = 3, the Z-value of  $O$ , i.e., xrep is  $(010011)_2$ .
4. Concatenating indexpartition and xrep,  $B^x$ value  $(00010011)_2 = 19$ .
5. Example  $O ((0, 6), (0.2, -0.3), 10)$  and  $tmu=120$  then  $O$ 's position at the label timestamp of partition: ???

### 6.8.3 Insertion, update and deletion

Given a new object, its index key is computed and then the object is inserted into the  $B^x$ -tree as in the B+ tree. An update consists of a deletion followed by an insertion. An auxiliary structure is employed to keep the latest key of each index so that an object can be deleted by searching for the key. The indexing key is computed before affecting the tree. In this way, the  $B^x$ -tree directly inherits the good properties of the B+ tree, and achieves efficient update performance.

### 6.8.4 Queries

#### Range query

A range query retrieves all objects whose location falls within the rectangular range  $q = ([qx1, qy1]; [qx2, qy2])$  at time  $tq$  not prior to the current time.

The  $B^x$ -tree uses query-window enlargement technique to answer queries. Since the  $B^x$ -tree stores an object's location as of sometime after its update time, the enlargement involves two cases: a location must either be brought back to an earlier time or forward to a later time. The main idea is to enlarge the query window so that it encloses all objects whose positions are not within query window at

its label timestamp but will enter the query window at the query timestamp.

After the enlargement, the partitions of the  $B^x$ -tree need to be traversed to find objects falling in the enlarged query window. In each partition, the use of a space-filling curve means that a range query in the native, two-dimensional space becomes a set of range queries in the transformed, one-dimensional space.<sup>[1]</sup>

To avoid excessively large query region after expansion in skewed datasets, an optimization of the query algorithm exists,<sup>[3]</sup> which improves the query efficiency by avoiding unnecessary query enlargement.

### K nearest neighbor query

K nearest neighbor query is computed by iteratively performing range queries with an incrementally enlarged search region until k answers are obtained. Another possibility is to employ similar querying ideas in The iDistance Technique.

### Other queries

The range query and K Nearest Neighbor query algorithms can be easily extended to support interval queries, continuous queries, etc.<sup>[2]</sup>

## 6.8.5 Adapting relational database engines to accommodate moving objects

Since the  $B^x$ -tree is an index built on top of a  $B^+$  tree index, all operations in the  $B^x$ -tree, including the insertion, deletion and search, are the same as those in the  $B^+$  tree. There is no need to change the implementations of these operations. The only difference is to implement the procedure of deriving the indexing key as a stored procedure in an existing DBMS. Therefore the  $B^x$ -tree can be easily integrated into existing DBMS without touching the kernel.

SpADE<sup>[4]</sup> is moving object management system built on top of a popular relational database system MySQL, which uses the  $B^x$ -tree for indexing the objects. In the implementation, moving object data is transformed and stored directly on MySQL, and queries are transformed into standard SQL statements which are efficiently processed in the relational engine. Most importantly, all these are achieved neatly and independently without infiltrating into the MySQL core.

## 6.8.6 Performance tuning

### Potential problem with data skew

The  $B^x$  tree uses a grid for space partitioning while mapping two-dimensional location into one-dimensional key. This may introduce performance degradation to both query and update operations while dealing with skewed data. If grid cell is oversize, many objects are contained in a cell. Since objects in a cell are indistinguishable to the index, there will be some overflow nodes in the underlying  $B^+$  tree. The existing of overflow pages not only destroys the balancing of the tree but also increases the update cost. As for the queries, for the given query region, large cell incurs more false positives and increases the processing time. On the other hand, if the space is partitioned with finer grid, i.e. smaller cells, each cell contains few objects. There is hardly overflow pages so that the update cost is minimized. Fewer false positives are retrieved in a query. However, more cells are needed to be searched. The increase in the number of cells searched also increases the workload of a query.

### Index tuning

The ST<sup>2</sup>B-tree<sup>[5]</sup> introduces a self-tuning framework for tuning the performance of the  $B^x$ -tree while dealing with data skew in space and data change with time. In order to deal with data skew in space, the ST<sup>2</sup>B-tree splits the entire space into regions of different object density using a set of reference points. Each region uses an individual grid whose cell size is determined by the object density inside of it.

The  $B^x$ -tree have multiple partitions regarding different time intervals. As time elapsed, each partition grows and shrinks alternately. The ST<sup>2</sup>B-tree utilizes this feature to tune the index online in order to adjust the space partitioning to make itself accommodate to the data changes with time. In particular, as a partition shrinks to empty and starts growing, it chooses a new set of reference points and new grid for each reference point according to the latest data density. The tuning is based on the latest statistics collected during a given period of time, so that the way of space partitioning is supposed to fit the latest data distribution best. By this means, the ST<sup>2</sup>B-tree is expected to minimize the effect caused by data skew in space and data changes with time.

## 6.8.7 See also

- $B^+$  tree
- Hilbert curve
- Z-order (curve)

### 6.8.8 References

- [1] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. *Query and Update Efficient B+tree based Indexing of Moving Objects*. In Proceedings of 30th International Conference on Very Large Data Bases (VLDB), pages 768-779, 2004.
- [2] Dan Lin. *Indexing and Querying Moving Objects Databases*, PhD thesis, National University of Singapore, 2006.
- [3] Jensen, C.S., D. Tiesyte, N. Tradisauskas, Robust B+-Tree-Based Indexing of Moving Objects, in Proceedings of the Seventh International Conference on Mobile Data Management, Nara, Japan, 9 pages, May 9–12, 2006.
- [4] SpADE: A SPatio-temporal Autonomic Database Engine for location-aware services.
- [5] Su Chen, Beng Chin Ooi, Kan-Lee. Tan, and Mario A. Nascimento, ST2B-tree: A Self-Tunable Spatio-Temporal B+tree for Moving Objects. In Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD), page 29-42, 2008.

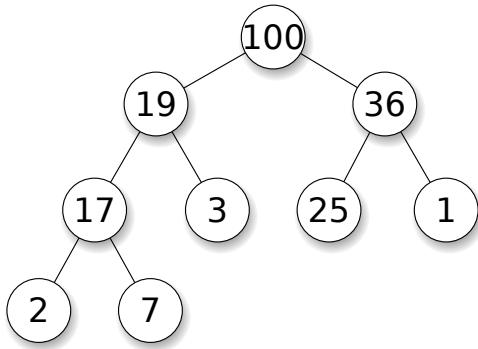
# Chapter 7

## Heaps

### 7.1 Heap

This article is about the programming data structure. For the dynamic memory area, see [Dynamic memory allocation](#).

In computer science, a **heap** is a specialized tree-based



*Example of a complete binary max-heap with node keys being integers from 1 to 100*

**Abstract data type** that satisfies the *heap property*: If A is a parent **node** of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap. Heaps can be classified further as either a "**max heap**" or a "**min heap**". In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node. In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node. Heaps are crucial in several efficient **graph algorithms** such as Dijkstra's algorithm, and in the sorting algorithm **heapsort**. A common implementation of a heap is the **binary heap**, in which the tree is a complete binary tree (see figure).

In a heap, the highest (or lowest) priority element is always stored at the root, hence the name **heap**. A heap is not a sorted structure and can be regarded as partially ordered. As visible from the Heap-diagram, there is no particular relationship among nodes on any given level, even among the siblings. When a heap is a complete binary tree, it has a smallest possible height—a heap with

$N$  nodes always has  $\log N$  height. A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority.

Note that, as shown in the graphic, there is no implied ordering between siblings or cousins and no implied sequence for an **in-order traversal** (as there would be in, e.g., a **binary search tree**). The heap relation mentioned above applies only between nodes and their parents, grandparents, etc. The maximum number of children each node can have depends on the type of heap, but in many types it is at most two, which is known as a **binary heap**.

The heap is one maximally efficient implementation of an abstract **data type** called a **priority queue**, and in fact priority queues are often referred to as "heaps", regardless of how they may be implemented. Note that despite the similarity of the name "heap" to "stack" and "queue", the latter two are abstract data types, while a heap is a specific data structure, and "priority queue" is the proper term for the abstract data type.

A **heap** data structure should not be confused with *the heap* which is a common name for the pool of memory from which **dynamically allocated memory** is allocated. The term was originally used only for the data structure.

#### 7.1.1 Operations

The common operations involving heaps are:

##### Basic

- *find-max* or *find-min*: find the maximum item of a max-heap or a minimum item of a min-heap (a.k.a. *peek*)
- *insert*: adding a new key to the heap (a.k.a., *push*<sup>[1]</sup>)
- *extract-min* [or *extract-max*]: returns the node of minimum value from a min heap [or maximum value from a max heap] after removing it from the heap (a.k.a., *pop*<sup>[2]</sup>)
- *delete-max* or *delete-min*: removing the root node of a max- or min-heap, respectively

- *replace*: pop root and push a new key. More efficient than pop followed by push, since only need to balance once, not twice, and appropriate for fixed-size heaps.<sup>[3]</sup>

## Creation

- *create-heap*: create an empty heap
- *heapify*: create a heap out of given array of elements
- *merge (union)*: joining two heaps to form a valid new heap containing all the elements of both, preserving the original heaps.
- *meld*: joining two heaps to form a valid new heap containing all the elements of both, destroying the original heaps.

## Inspection

- *size*: return the number of items in the heap.
- *is-empty*: return true if the heap is empty, false otherwise – an optimized form of size when total size is not needed.

## Internal

- *increase-key* or *decrease-key*: updating a key within a max- or min-heap, respectively
- *delete*: delete an arbitrary node (followed by moving last node and sifting to maintain heap)
- *sift-up*: move a node up in the tree, as long as needed; used to restore heap condition after insertion. Called “sift” because node moves up the tree until it reaches the correct level, as in a sieve. Often incorrectly called “shift-up”.
- *sift-down*: move a node down in the tree, similar to sift-up; used to restore heap condition after deletion or replacement.

## 7.1.2 Implementation

Heaps are usually implemented in an array (fixed size or dynamic array), and do not require pointers between elements. After an element is inserted into or deleted from a heap, the heap property is violated and the heap must be balanced by internal operations.

Full and almost full binary heaps may be represented in a very space-efficient way (as an implicit data structure) using an array alone. The first (or last) element will contain the root. The next two elements of the array contain its children. The next four contain the four children of the two child nodes, etc. Thus the children of the node at

position  $n$  would be at positions  $2n$  and  $2n + 1$  in a one-based array, or  $2n + 1$  and  $2n + 2$  in a zero-based array. This allows moving up or down the tree by doing simple index computations. Balancing a heap is done by shift-up or shift-down operations (swapping elements which are out of order). As we can build a heap from an array without requiring extra memory (for the nodes, for example), `heapsort` can be used to sort an array in-place.

Different types of heaps implement the operations in different ways, but notably, insertion is often done by adding the new element at the end of the heap in the first available free space. This will generally violate the heap property, and so the elements are then sifted up until the heap property has been reestablished. Similarly, deleting the root is done by removing the root and then putting the last element in the root and sifting down to rebalance. Thus replacing is done by deleting the root and putting the new element in the root and sifting down, avoiding a sifting up step compared to pop (sift down of last element) followed by push (sift up of new element).

Construction of a binary (or  $d$ -ary) heap out of a given array of elements may be performed in linear time using the classic [Floyd algorithm](#), with the worst-case number of comparisons equal to  $2N - 2s_2(N) - e_2(N)$  (for a binary heap), where  $s_2(N)$  is the sum of all digits of the binary representation of  $N$  and  $e_2(N)$  is the exponent of 2 in the prime factorization of  $N$ .<sup>[4]</sup> This is faster than a sequence of consecutive insertions into an originally empty heap, which is log-linear.<sup>[lower-alpha 1]</sup>

## 7.1.3 Variants

- 2–3 heap
- B-heap
- Beap
- Binary heap
- Binomial heap
- Brodal queue
- $d$ -ary heap
- Fibonacci heap
- Leftist heap
- Pairing heap
- Skew heap
- Soft heap
- Weak heap
- Leaf heap
- Radix heap

- Randomized meldable heap
- Ternary heap
- Treap

### 7.1.4 Comparison of theoretic bounds for variants

In the following time complexities<sup>[5]</sup>  $O(f)$  is an asymptotic upper bound and  $\Theta(f)$  is an asymptotically tight bound (see [Big O notation](#)). Function names assume a min-heap.

- [1] Each insertion takes  $O(\log(k))$  in the existing size of the heap, thus  $\sum_{k=1}^n O(\log k)$ . Since  $\log n/2 = (\log n) - 1$ , a constant factor (half) of these insertions are within a constant factor of the maximum, so asymptotically we can assume  $k = n$ ; formally the time is  $nO(\log n) - O(n) = O(n \log n)$ .
- [2] Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with  $n$  elements can be constructed bottom-up in  $O(n)$ .<sup>[8]</sup>
- [3] Amortized time.
- [4] Bounded by  $\Omega(\log \log n)$ ,  $O(2^{2\sqrt{\log \log n}})$ <sup>[11][12]</sup>
- [5]  $n$  is the size of the larger heap and  $m$  is the size of the smaller heap.
- [6]  $n$  is the size of the larger heap.

### 7.1.5 Applications

The heap data structure has many applications.

- **Heapsort:** One of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- **Selection algorithms:** A heap allows access to the min or max element in constant time, and other selections (such as median or  $k$ th-element) can be done in sub-linear time on data that is in a heap.<sup>[13]</sup>
- **Graph algorithms:** By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal-spanning-tree algorithm and Dijkstra's shortest-path algorithm.
- **Priority Queue:** A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods.
- **Order statistics:** The Heap data structure can be used to efficiently find the  $k$ th smallest (or largest) element in an array.

### 7.1.6 Implementations

- The [C++ Standard Template Library](#) provides the `make_heap`, `push_heap` and `pop_heap` algorithms for heaps (usually implemented as binary heaps), which operate on arbitrary random access [iterators](#). It treats the iterators as a reference to an array, and uses the array-to-heap conversion. It also provides the container adaptor `priority_queue`, which wraps these facilities in a container-like class. However, there is no standard support for the decrease/increase-key operation.
- The [Boost C++ libraries](#) include a `heaps` library. Unlike the STL it supports decrease and increase operations, and supports additional types of heap: specifically, it supports  $d$ -ary, binomial, Fibonacci, pairing and skew heaps.
- The [Java 2](#) platform (since version 1.5) provides the binary heap implementation with class `java.util.PriorityQueue<E>` in [Java Collections Framework](#). However, there is no support for the decrease/increase-key operation.
- Python has a `heapq` module that implements a priority queue using a binary heap.
- [PHP](#) has both max-heap (`SplMaxHeap`) and min-heap (`SplMinHeap`) as of version 5.3 in the [Standard PHP Library](#).
- [Perl](#) has implementations of binary, binomial, and Fibonacci heaps in the `Heap` distribution available on [CPAN](#).
- The [Go](#) language contains a `heap` package with heap algorithms that operate on an arbitrary type that satisfy a given interface.
- Apple's [Core Foundation](#) library contains a `CFBinaryHeap` structure.
- [Pharo](#) has an implementation in the `Collections-Sequenceable` package along with a set of test cases. A heap is used in the implementation of the timer event loop.

### 7.1.7 See also

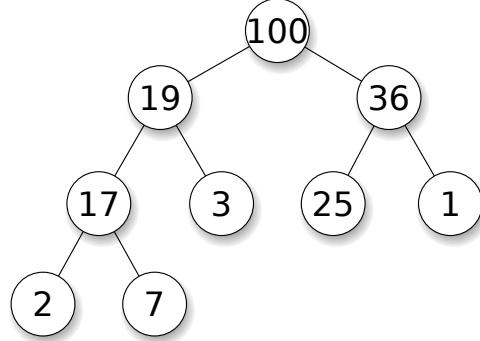
- [Sorting algorithm](#)
- [Stack \(abstract data type\)](#)
- [Queue \(abstract data type\)](#)
- [Tree \(data structure\)](#)
- Treap, a form of binary search tree based on heap-ordered trees

### 7.1.8 References

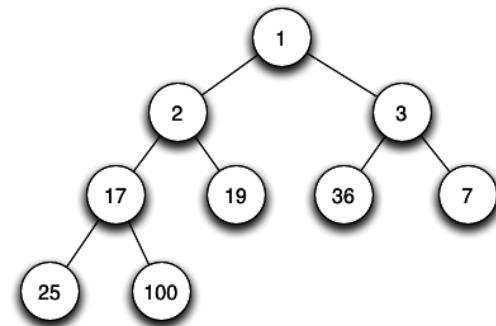
- [1] The Python Standard Library, 8.4. `heapq` — Heap queue algorithm, `heapq.heappush`
- [2] The Python Standard Library, 8.4. `heapq` — Heap queue algorithm, `heapq.heappop`
- [3] The Python Standard Library, 8.4. `heapq` — Heap queue algorithm, `heapq.heapreplace`
- [4] Suchenek, Marek A. (2012), “Elementary Yet Precise Worst-Case Analysis of Floyd’s Heap-Construction Program”, *Fundamenta Informaticae* (IOS Press) **120** (1): 75–92, doi:10.3233/FI-2012-751.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest (1990): Introduction to algorithms. MIT Press / McGraw-Hill.
- [6] Iacono, John (2000), “Improved upper bounds for pairing heaps”, *Proc. 7th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science **1851**, Springer-Verlag, pp. 63–77, doi:10.1007/3-540-44985-X\_5
- [7] Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 52–58, 1996
- [8] Goodrich, Michael T.; Tamassia, Roberto (2004). “7.3.6. Bottom-Up Heap Construction”. *Data Structures and Algorithms in Java* (3rd ed.). pp. 338–341.
- [9] Haeupler, Bernhard; Sen, Siddhartha; Tarjan, Robert E. (2009). “Rank-pairing heaps” (PDF). *SIAM J. Computing*: 1463–1485.
- [10] Brodal, G. S. L.; Lagogiannis, G.; Tarjan, R. E. (2012). *Strict Fibonacci heaps* (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12, p. 1177. doi:10.1145/2213977.2214082. ISBN 9781450312455.
- [11] Fredman, Michael Lawrence; Tarjan, Robert E. (1987). “Fibonacci heaps and their uses in improved network optimization algorithms” (PDF). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi:10.1145/28869.28874.
- [12] Pettie, Seth (2005). “Towards a Final Analysis of Pairing Heaps” (PDF). *Max Planck Institut für Informatik*.
- [13] Frederickson, Greg N. (1993), “An Optimal Algorithm for Selection in a Min-Heap”, *Information and Computation* (PDF) **104** (2), Academic Press, pp. 197–214, doi:10.1006/inco.1993.1030

### 7.1.9 External links

- [Heap at Wolfram MathWorld](#)



Example of a complete binary max heap



Example of a complete binary min heap

## 7.2 Binary heap

A **binary heap** is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints:

**Shape property** A binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

**Heap property** All nodes are *either greater than or equal to or less than or equal to* each of its children, according to a comparison predicate defined for the heap.

Heaps with a mathematical “greater than or equal to” ( $\geq$ ) comparison predicate are called *max-heaps*; those with a mathematical “less than or equal to” ( $\leq$ ) comparison predicate are called *min-heaps*. Min-heaps are often used to implement priority queues.<sup>[1][2]</sup>

Since the ordering of siblings in a heap is not specified by the heap property, a single node’s two children can be freely interchanged unless doing so violates the shape property (compare with treap). Note, however, that in

the common array-based heap, simply swapping the children might also necessitate moving the childrens' sub-tree nodes to retain the heap property.

The binary heap is a special case of the **d-ary heap** in which  $d = 2$ .

### 7.2.1 Heap operations

Both the insert and remove operations modify the heap to conform to the shape property first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take  $O(\log n)$  time.

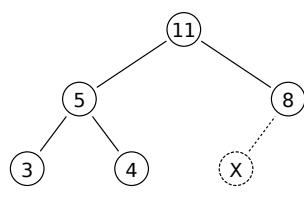
#### Insert

To add an element to a heap we must perform an *up-heap* operation (also known as *bubble-up*, *percolate-up*, *sift-up*, *trickle-up*, *heapify-up*, or *cascade-up*), by following this algorithm:

1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

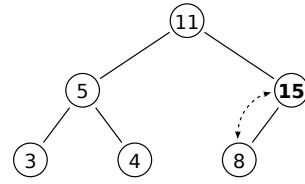
The number of operations required is dependent on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a time complexity of  $O(\log n)$ . However, in 1974, Thomas Porter and Istvan Simon proved that the function for the average number of levels an inserted node moves up is upper bounded by the constant 1.6067.<sup>[3]</sup> The average number of operations required for an insertion into a binary heap is 2.6067 since one additional comparison is made that does not result in the inserted node moving up a level. Thus, on average, binary heap insertion has a constant,  $O(1)$ , time complexity. Intuitively, this makes sense since approximately 50% of the elements are leaves and approximately 75% of the elements are in the bottom two levels, it is likely that the new element to be inserted will only move a few levels upwards to maintain the heap.

As an example of binary heap insertion, say we have a max-heap

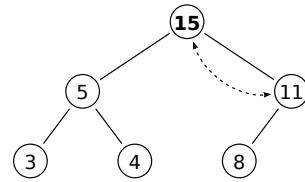


and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However,

the heap property is violated since  $15 > 8$ , so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:



However the heap property is still violated since  $15 > 11$ , so we need to swap again:



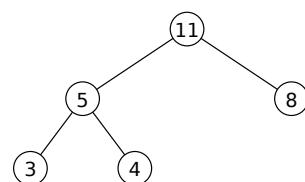
which is a valid max-heap. There is no need to check left child after this final step. Before we placed 15 on X in 1-st step, the max-heap was valid, meaning  $11 > 5$ . If  $15 > 11$ , and  $11 > 5$ , then  $15 > 5$ , because of the *transitive relation*.

#### Delete

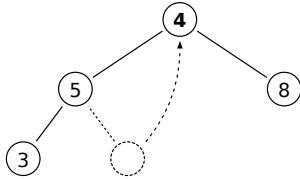
The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *sift-down*, *trickle down*, *heapify-down*, *cascade-down* and *extract-min/max*).

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

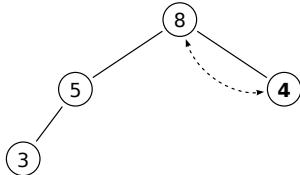
So, if we have the same max-heap as before



We remove the 11 and replace it with the 4.



Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further:



The downward-moving node is swapped with the *larger* of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position. This functionality is achieved by the **Max-Heapify** function as defined below in pseudocode for an array-backed heap  $A$  of length  $heap\_length[A]$ . Note that “A” is indexed starting at 1, not 0 as is common in many real programming languages.

**Max-Heapify** ( $A, i$ ):

```

left ← 2i
right ← 2i + 1
largest ← i
if left ≤ heap_length[A] and A[left] > A[largest] then:
 largest ← left
if right ≤ heap_length[A] and A[right] > A[largest] then:
 largest ← right
if largest ≠ i then:
 swap A[i] ↔ A[largest]
 Max-Heapify(A, largest)

```

For the above algorithm to correctly re-heapify the array, the node at index  $i$  and its two direct children must violate the heap property. If they do not, the algorithm will fall through with no change to the array. The down-heap operation (without the preceding swap) can also be used to modify the value of the root, even when an element is not being deleted.

In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or  $O(\log n)$ .

### 7.2.2 Building a heap

A heap could be built by successive insertions. This approach requires  $O(n \log n)$  time because each insertion takes  $O(\log n)$  time and there are  $n$  elements. However this is not the optimal method. The optimal method starts

by arbitrarily putting the elements on a binary tree, respecting the shape property (the tree could be represented by an array, see below). Then starting from the lowest level and moving upwards, shift the root of each subtree downward as in the deletion algorithm until the heap property is restored. More specifically if all the subtrees starting at some height  $h$  (measured from the bottom) have already been “heapified”, the trees at height  $h + 1$  can be heapified by sending their root down along the path of maximum valued children when building a max-heap, or minimum valued children when building a min-heap. This process takes  $O(h)$  operations (swaps) per node. In this method most of the heapification takes place in the lower levels. Since the height of the heap is  $\lfloor \log(n) \rfloor$ , the number of nodes at height  $h$  is  $\leq \lceil 2^{(\log n - h) - 1} \rceil = \lceil \frac{2^{\log n}}{2^{h+1}} \rceil = \lceil \frac{n}{2^{h+1}} \rceil$ . Therefore, the cost of heapifying all subtrees is:

$$\begin{aligned}
 \sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^{h+1}} O(h) &= O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}}\right) \\
 &\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\
 &= O(n)
 \end{aligned}$$

This uses the fact that the given infinite series  $h / 2^h$  converges to 2.

The exact value of the above (the worst-case number of comparisons during the heap construction) is known to be equal to:

$$2n - 2s_2(n) - e_2(n), [4]$$

where  $s_2(n)$  is the sum of all digits of the binary representation of  $n$  and  $e_2(n)$  is the exponent of 2 in the prime factorization of  $n$ .

The **Build-Max-Heap** function that follows, converts an array  $A$  which stores a complete binary tree with  $n$  nodes to a max-heap by repeatedly using **Max-Heapify** in a bottom up manner. It is based on the observation that the array elements indexed by  $\text{floor}(n/2) + 1, \text{floor}(n/2) + 2, \dots, n$  are all leaves for the tree, thus each is a one-element heap. **Build-Max-Heap** runs **Max-Heapify** on each of the remaining tree nodes.

**Build-Max-Heap** ( $A$ ):

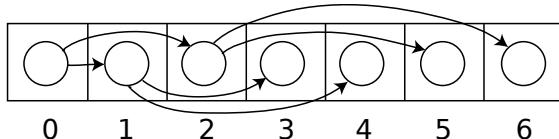
```

heap_length[A] ← length[A]
for i ← floor(length[A]/2) downto 1 do
 Max-Heapify(A, i)

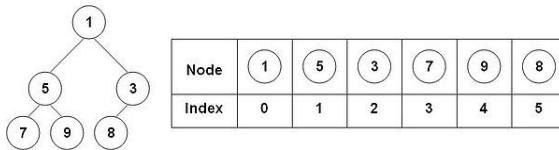
```

### 7.2.3 Heap implementation

Heaps are commonly implemented with an array. Any binary tree can be stored in an array, but because a binary



A small complete binary tree stored in an array



Comparison between a binary heap and an array implementation.

heap is always a complete binary tree, it can be stored compactly. No space is required for **pointers**; instead, the parent and children of each node can be found by arithmetic on array indices. These properties make this heap implementation a simple example of an **implicit data structure** or **Ahnentafel list**. Details depend on the root position, which in turn may depend on constraints of a **programming language** used for implementation, or programmer preference. Specifically, sometimes the root is placed at index 1, sacrificing space in order to simplify arithmetic. The **peek** operation (*find-min* or *find-max*) simply returns the value of the root, and is thus  $O(1)$ .

Let  $n$  be the number of elements in the heap and  $i$  be an arbitrary valid index of the array storing the heap. If the tree root is at index 0, with valid indices 0 through  $n - 1$ , then each element  $a$  at index  $i$  has

- children at indices  $2i + 1$  and  $2i + 2$
- its parent  $\text{floor}((i - 1) / 2)$ .

Alternatively, if the tree root is at index 1, with valid indices 1 through  $n$ , then each element  $a$  at index  $i$  has

- children at indices  $2i$  and  $2i + 1$
- its parent at index  $\text{floor}(i / 2)$ .

This implementation is used in the **heapsort** algorithm, where it allows the space in the input array to be reused to store the heap (i.e. the algorithm is done **in-place**). The implementation is also useful for use as a **Priority queue** where use of a **dynamic array** allows insertion of an unbounded number of items.

The upheap/downheap operations can then be stated in terms of an array as follows: suppose that the heap property holds for the indices  $b, b+1, \dots, e$ . The sift-down function extends the heap property to  $b-1, b, b+1, \dots, e$ . Only index  $i = b-1$  can violate the heap property. Let  $j$  be the index of the largest child of  $a[i]$  (for a max-heap, or the smallest child for a min-heap) within the range  $b, b+1, \dots, e$ .

...*e*. (If no such index exists because  $2i > e$  then the heap property holds for the newly extended range and nothing needs to be done.) By swapping the values  $a[i]$  and  $a[j]$  the heap property for position  $i$  is established. At this point, the only problem is that the heap property might not hold for index  $j$ . The sift-down function is applied **tail-recursively** to index  $j$  until the heap property is established for all elements.

The sift-down function is fast. In each step it only needs two comparisons and one swap. The index value where it is working doubles in each iteration, so that at most  $\log_2 e$  steps are required.

For big heaps and using **virtual memory**, storing elements in an array according to the above scheme is inefficient: (almost) every level is in a different **page**. **B-heaps** are binary heaps that keep subtrees in a single page, reducing the number of pages accessed by up to a factor of ten.<sup>[5]</sup>

The operation of merging two binary heaps takes  $\Theta(n)$  for equal-sized heaps. The best you can do is (in case of array implementation) simply concatenating the two heap arrays and build a heap of the result.<sup>[6]</sup> A heap on  $n$  elements can be merged with a heap on  $k$  elements using  $O(\log n \log k)$  key comparisons, or, in case of a pointer-based implementation, in  $O(\log n \log k)$  time.<sup>[7]</sup> An algorithm for splitting a heap on  $n$  elements into two heaps on  $k$  and  $n-k$  elements, respectively, based on a new view of heaps as an ordered collections of subheaps was presented in.<sup>[8]</sup> The algorithm requires  $O(\log n * \log n)$  comparisons. The view also presents a new and conceptually simple algorithm for merging heaps. When merging is a common task, a different heap implementation is recommended, such as **binomial heaps**, which can be merged in  $O(\log n)$ .

Additionally, a binary heap can be implemented with a traditional binary tree data structure, but there is an issue with finding the adjacent element on the last level on the binary heap when adding an element. This element can be determined algorithmically or by adding extra data to the nodes, called “threading” the tree—instead of merely storing references to the children, we store the inorder successor of the node as well.

It is possible to modify the heap structure to allow extraction of both the smallest and largest element in  $O(\log n)$  time.<sup>[9]</sup> To do this, the rows alternate between min heap and max heap. The algorithms are roughly the same, but, in each step, one must consider the alternating rows with alternating comparisons. The performance is roughly the same as a normal single direction heap. This idea can be generalised to a min-max-median heap.

## 7.2.4 Derivation of index equations

In an array-based heap, the children and parent of a node can be located via simple arithmetic on the node’s index. This section derives the relevant equations for heaps with

their root at index 0, with additional notes on heaps with their root at index 1.

To avoid confusion, we'll define the **level** of a node as its distance from the root, such that the root itself occupies level 0.

### Child nodes

For a general node located at index  $i$  (beginning from 0), we will first derive the index of its right child,  $\text{right} = 2i + 2$ .

Let node  $i$  be located in level  $L$ , and note that any level  $l$  contains exactly  $2^l$  nodes. Furthermore, there are exactly  $2^{l+1} - 1$  nodes contained in the layers up to and including layer  $l$  (think of binary arithmetic; 0111...111 = 1000...000 - 1). Because the root is stored at 0, the  $k$ th node will be stored at index  $(k - 1)$ . Putting these observations together yields the following expression for the **index of the last node in layer 1**.

$$\text{last}(l) = (2^{l+1} - 1) - 1 = 2^{l+1} - 2$$

Let there be  $j$  nodes after node  $i$  in layer  $L$ , such that

$$\begin{aligned} i &= \text{last}(L) - j \\ &= (2^{L+1} - 2) - j \end{aligned}$$

Each of these  $j$  nodes must have exactly 2 children, so there must be  $2j$  nodes separating  $i$ 's right child from the end of its layer ( $L + 1$ ).

$$\begin{aligned} \text{right} &= 1 + \text{last}(L - 2j) \\ &= (2^{L+2} - 2) - 2j \\ &= 2(2^{L+1} - 2 - j) + 2 \\ &= 2i + 2 \end{aligned}$$

As required.

Noting that the left child of any node is always 1 place before its right child, we get  $\text{left} = 2i + 1$ .

If the root is located at index 1 instead of 0, the last node in each level is instead at index  $2^{l+1} - 1$ . Using this throughout yields  $\text{left} = 2i$  and  $\text{right} = 2i + 1$  for heaps with their root at 1.

### Parent node

Every node is either the left or right child of its parent, so we know that either of the following is true.

1.  $i = 2 \times (\text{parent}) + 1$
2.  $i = 2 \times (\text{parent}) + 2$

Hence,

$$\text{parent} = \frac{i - 1}{2} \text{ or } \frac{i - 2}{2}$$

Now consider the expression  $\left\lfloor \frac{i - 1}{2} \right\rfloor$ .

If node  $i$  is a left child, this gives the result immediately, however, it also gives the correct result if node  $i$  is a right child. In this case,  $(i - 2)$  must be even, and hence  $(i - 1)$  must be odd.

$$\begin{aligned} \left\lfloor \frac{i - 1}{2} \right\rfloor &= \left\lfloor \frac{i - 2}{2} + \frac{1}{2} \right\rfloor \\ &= \frac{i - 2}{2} \\ &= \text{parent} \end{aligned}$$

Therefore, irrespective of whether a node is a left or right child, its parent can be found by the expression:

$$\text{parent} = \left\lfloor \frac{i - 1}{2} \right\rfloor$$

### 7.2.5 See also

- Heap
- Heapsort

### 7.2.6 Notes

### 7.2.7 References

- [1] “heapq – Heap queue algorithm”. *Python Standard Library*.
- [2] “Class PriorityQueue”. *Java™ Platform Standard Ed. 6*.
- [3] Thomas Porter; Istvan Simon (1974). “Random Insertion into a Priority Queue Structure” (PDF). *Stanford University Reports*. Stanford University. p. 13. Retrieved 31 January 2014.
- [4] Suchenek, Marek A. (2012), “Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program”, *Fundamenta Informaticae* (IOS Press) **120** (1): 75–92, doi:10.3233/FI-2012-751.

- [5] Poul-Henning Kamp. “You’re Doing It Wrong”. ACM Queue. June 11, 2010.
- [6] Chris L. Kuszmaul. “binary heap”. Dictionary of Algorithms and Data Structures, Paul E. Black, ed., U.S. National Institute of Standards and Technology. 16 November 2009.
- [7] J.-R. Sack and T. Strothotte “An Algorithm for Merging Heaps”, Acta Informatica 22, 171-186 (1985).
- [8] J.-R. Sack and T. Strothotte “A characterization of heaps and its applications” Information and Computation Volume 86, Issue 1, May 1990, Pages 69–86.
- [9] Atkinson, M.D., J.-R. Sack, N. Santoro, and T. Strothotte (1 October 1986). “Min-max heaps and generalized priority queues.” (PDF). Programming techniques and Data structures. Comm. ACM, 29(10): 996–1000.

## 7.2.8 External links

- Binary Heap Applet by Kubo Kovac
- Using Binary Heaps in A\* Pathfinding
- Open Data Structures - Section 10.1 - BinaryHeap: An Implicit Binary Tree
- Implementation of binary max heap in C by Robin Thomas
- Implementation of binary min heap in C by Robin Thomas

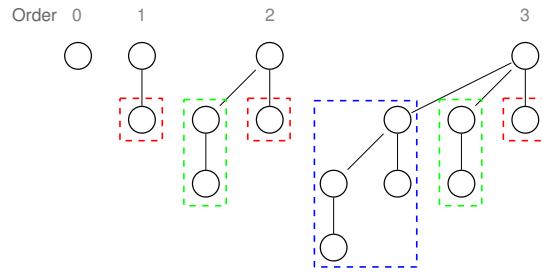
## 7.3 Binomial heap

In computer science, a **binomial heap** is a heap similar to a binary heap but also supports quick merging of two heaps. This is achieved by using a special tree structure. It is important as an implementation of the **mergeable heap abstract data type** (also called **meldable heap**), which is a priority queue supporting merge operation.

### 7.3.1 Binomial heap

A binomial heap is implemented as a collection of binomial trees (compare with a **binary heap**, which has a shape of a single **binary tree**). A **binomial tree** is defined recursively:

- A binomial tree of order 0 is a single node
- A binomial tree of order  $k$  has a root node whose children are roots of binomial trees of orders  $k-1, k-2, \dots, 2, 1, 0$  (in this order).



*Binomial trees of order 0 to 3: Each tree has a root node with subtrees of all lower ordered binomial trees, which have been highlighted. For example, the order 3 binomial tree is connected to an order 2, 1, and 0 (highlighted as blue, green and red respectively) binomial tree.*

A binomial tree of order  $k$  has  $2^k$  nodes, height  $k$ .

Because of its unique structure, a binomial tree of order  $k$  can be constructed from two trees of order  $k-1$  trivially by attaching one of them as the leftmost child of the root of the other tree. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps.

The name comes from the shape: a binomial tree of order  $n$  has  $\binom{n}{d}$  nodes at depth  $d$ . (See **Binomial coefficient**.)

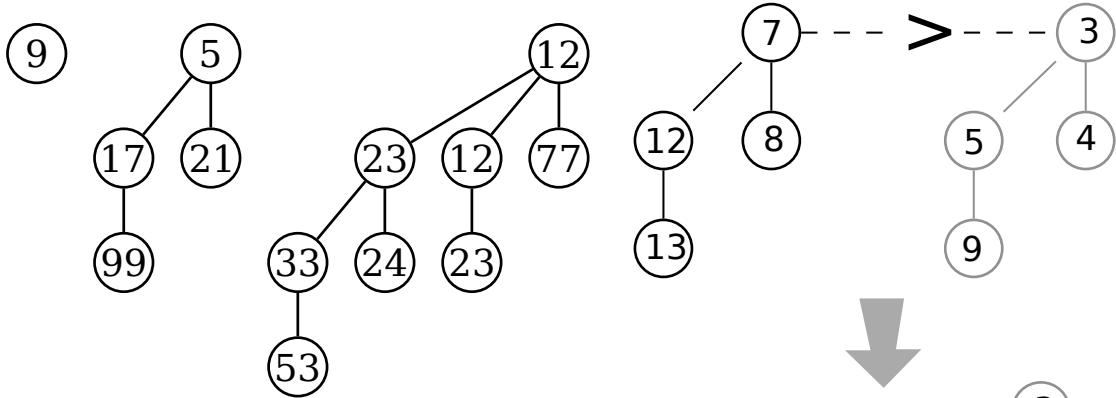
### 7.3.2 Structure of a binomial heap

A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:

- Each binomial tree in a heap obeys the *minimum-heap property*: the key of a node is greater than or equal to the key of its parent.
- There can only be either *one* or *zero* binomial trees for each order, including zero order.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap.

The second property implies that a binomial heap with  $n$  nodes consists of at most  $\log n + 1$  binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of nodes  $n$ : each binomial tree corresponds to one digit in the **binary representation** of number  $n$ . For example number 13 is 1101 in binary,  $2^3 + 2^2 + 2^0$ , and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0 (see figure below).



Example of a binomial heap containing 13 nodes with distinct keys.

The heap consists of three binomial trees with orders 0, 2, and 3.

### 7.3.3 Implementation

Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a [linked list](#), ordered by increasing order of the tree.

#### Merge

As mentioned above, the simplest and most important operation is the merging of two binomial trees of the same order within a binomial heap. Due to the structure of binomial trees, they can be merged trivially. As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes the new root node. Then the other tree becomes a subtree of the combined tree. This operation is basic to the complete merging of two binomial heaps.

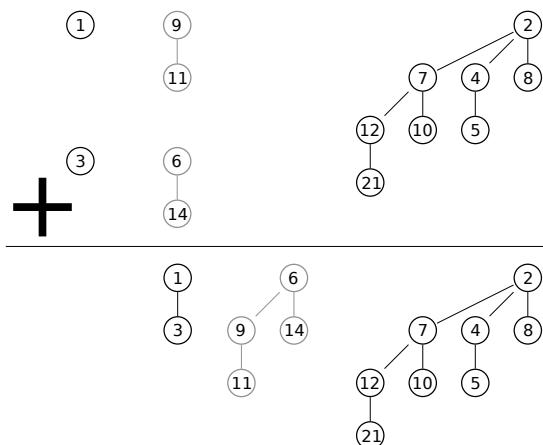
```
function mergeTree(p, q) if p.root.key <= q.root.key return p.addSubTree(q) else return q.addSubTree(p)
```

The operation of **merging** two heaps is perhaps the most interesting and can be used as a subroutine in most other operations. The lists of roots of both heaps are traversed simultaneously, similarly as in the [merge algorithm](#).

If only one of the heaps contains a tree of order  $j$ , this tree is moved to the merged heap. If both heaps contain a tree of order  $j$ , the two trees are merged to one tree of order  $j+1$  so that the minimum-heap property is satisfied. Note that it may later be necessary to merge this tree with some other tree of order  $j+1$  present in one of the heaps. In the course of the algorithm, we need to examine at most three trees of any order (two from the two heaps we merge and one composed of two smaller trees).

Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size,

To merge two binomial trees of the same order, first compare the root key. Since  $7 > 3$ , the black tree on the left (with root node 7) is attached to the grey tree on the right (with root node 3) as a subtree. The result is a tree of order 3.



This shows the merger of two binomial heaps. This is accomplished by merging two binomial trees of the same order one by one. If the resulting merged tree has the same order as one binomial tree in one of the two heaps, then those two are merged again.

there is an analogy between the merging of two heaps and the binary addition of the [sizes](#) of the two heaps, from right-to-left. Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge.

Each tree has order at most  $\log n$  and therefore the running time is  $O(\log n)$ .

```
function merge(p, q) while not (p.end() and q.end())
tree = mergeTree(p.currentTree(), q.currentTree())
if not heap.currentTree().empty() tree = mergeTree(tree,
heap.currentTree()) heap.addTree(tree) heap.next();
p.next(); q.next()
```

### Insert

**Inserting** a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes  $O(\log n)$  time. However, across a series of  $n$  consecutive insertions, **insert** has an *amortized* time of  $O(1)$  (i.e. constant).

### Find minimum

To find the **minimum** element of the heap, find the minimum among the roots of the binomial trees. This can again be done easily in  $O(\log n)$  time, as there are just  $O(\log n)$  trees and hence roots to examine.

By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to  $O(1)$ . The pointer must be updated when performing any operation other than Find minimum. This can be done in  $O(\log n)$  without raising the running time of any operation.

### Delete minimum

To **delete the minimum element** from the heap, first find this element, remove it from its binomial tree, and obtain a list of its subtrees. Then transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order. Then merge this heap with the original heap. Since each tree has at most  $\log n$  children, creating this new heap is  $O(\log n)$ . Merging heaps is  $O(\log n)$ , so the entire delete minimum operation is  $O(\log n)$ .

```
function deleteMin(heap) min = heap.trees().first() for
each current in heap.trees() if current.root < min.root
then min = current for each tree in min.subTrees()
tmp.addTree(tree) heap.removeTree(min) merge(heap,
tmp)
```

### Decrease key

After **decreasing** the key of an element, it may become smaller than the key of its parent, violating the minimum-heap property. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated. Each binomial tree has height at most  $\log n$ , so this takes  $O(\log n)$  time.

### Delete

To **delete** an element from the heap, decrease its key to negative infinity (that is, some value lower than any element in the heap) and then delete the minimum in the heap.

### 7.3.4 Summary of running times

In the following time complexities<sup>[1]</sup>  $O(f)$  is an asymptotic upper bound and  $\Theta(f)$  is an asymptotically tight bound (see **Big O notation**). Function names assume a min-heap.

- [1] Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with  $n$  elements can be constructed bottom-up in  $O(n)$ .<sup>[4]</sup>
- [2] Amortized time.
- [3] Bounded by  $\Omega(\log \log n)$ ,  $O(2^{2\sqrt{\log \log n}})$  <sup>[7][8]</sup>
- [4]  $n$  is the size of the larger heap and  $m$  is the size of the smaller heap.
- [5]  $n$  is the size of the larger heap.

### 7.3.5 Applications

- Discrete event simulation
- Priority queues

### 7.3.6 See also

- Fibonacci heap
- Soft heap
- Skew binomial heap

### 7.3.7 References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 19: Binomial Heaps, pp. 455–475.
- Vuillemin, J. (1978). A data structure for manipulating priority queues. *Communications of the ACM* **21**, 309–314.

### 7.3.8 External links

- Java applet simulation of binomial heap
- Python implementation of binomial heap
- Two C implementations of binomial heap (a generic one and one optimized for integer keys)
- Haskell implementation of binomial heap
- Common Lisp implementation of binomial heap

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest (1990): Introduction to algorithms. MIT Press / McGraw-Hill.
- [2] Iacono, John (2000), “Improved upper bounds for pairing heaps”, *Proc. 7th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science **1851**, Springer-Verlag, pp. 63–77, doi:10.1007/3-540-44985-X\_5
- [3] Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 52–58, 1996
- [4] Goodrich, Michael T.; Tamassia, Roberto (2004). “7.3.6. Bottom-Up Heap Construction”. *Data Structures and Algorithms in Java* (3rd ed.). pp. 338–341.
- [5] Haeupler, Bernhard; Sen, Siddhartha; Tarjan, Robert E. (2009). “Rank-pairing heaps” (PDF). *SIAM J. Computing*: 1463–1485.
- [6] Brodal, G. S. L.; Lagogiannis, G.; Tarjan, R. E. (2012). *Strict Fibonacci heaps* (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12. p. 1177. doi:10.1145/2213977.2214082. ISBN 9781450312455.
- [7] Fredman, Michael Lawrence; Tarjan, Robert E. (1987). “Fibonacci heaps and their uses in improved network optimization algorithms” (PDF). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi:10.1145/28869.28874.
- [8] Pettie, Seth (2005). “Towards a Final Analysis of Pairing Heaps” (PDF). *Max Planck Institut für Informatik*.

## 7.4 Fibonacci heap

In computer science, a **Fibonacci heap** is a **heap data structure** consisting of a collection of **trees**. It has a better amortized running time than a **binomial heap**. Fibonacci heaps were developed by **Michael L. Fredman** and **Robert E. Tarjan** in 1984 and first published in a scientific journal in 1987. The name of Fibonacci heap comes from **Fibonacci numbers** which are used in the running time analysis.

Find-minimum is  $O(1)$  amortized time.<sup>[1]</sup> Operations insert, decrease key, and merge (union) work in constant amortized time.<sup>[2]</sup> Operations delete and delete minimum

work in  $O(\log n)$  amortized time.<sup>[2]</sup> This means that starting from an empty data structure, any sequence of  $a$  operations from the first group and  $b$  operations from the second group would take  $O(a + b \log n)$  time. In a **binomial heap** such a sequence of operations would take  $O((a + b) \log n)$  time. A Fibonacci heap is thus better than a binomial heap when  $b$  is asymptotically smaller than  $a$ .

Using Fibonacci heaps for **priority queues** improves the asymptotic running time of important algorithms, such as **Dijkstra’s algorithm** for computing the **shortest path** between two nodes in a graph.

### 7.4.1 Structure

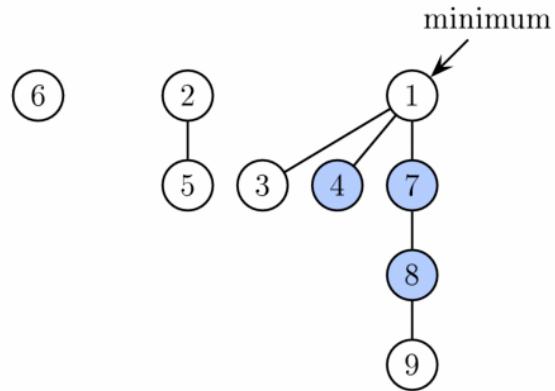


Figure 1. Example of a Fibonacci heap. It has three trees of degrees 0, 1 and 3. Three vertices are marked (shown in blue). Therefore the potential of the heap is 9 (3 trees + 2 \* marked-vertices).

A Fibonacci heap is a collection of **trees** satisfying the **minimum-heap property**, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a “lazy” manner, postponing the work for later operations. For example merging heaps is done simply by concatenating the two lists of trees, and operation *decrease key* sometimes cuts a node from its parent and forms a new tree.

However at some point some order needs to be introduced to the heap to achieve the desired running time. In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most  $O(\log n)$  and the size of a subtree rooted in a node of degree  $k$  is at least  $F_{k+2}$ , where  $F_k$  is the  $k$ th **Fibonacci number**. This is achieved by the rule that we can cut at most one child of each non-root node. When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree (see Proof of degree bounds, below). The number of trees is decreased

in the operation *delete minimum*, where trees are linked together.

As a result of a relaxed structure, some operations can take a long time while others are done very quickly. For the amortized running time analysis we use the **potential method**, in that we pretend that very fast operations take a little bit longer than they actually do. This additional time is then later combined and subtracted from the actual running time of slow operations. The amount of time saved for later use is measured at any given moment by a potential function. The potential of a Fibonacci heap is given by

$$\text{Potential} = t + 2m$$

where  $t$  is the number of trees in the Fibonacci heap, and  $m$  is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked). The amortized time for an operation is given by the sum of the actual time and  $c$  times the difference in potential, where  $c$  is a constant (chosen to match the constant factors in the  $O$  notation for the actual time).

Thus, the root of each tree in a heap has one unit of time stored. This unit of time can be used later to link this tree with another tree at amortized time 0. Also, each marked node has two units of time stored. One can be used to cut the node from its parent. If this happens, the node becomes a root and the second unit of time will remain stored in it as in any other root.

#### 7.4.2 Implementation of operations

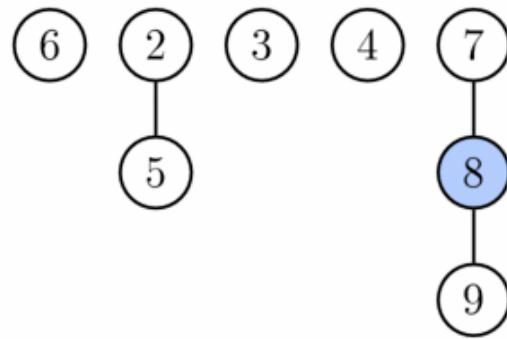
To allow fast deletion and concatenation, the roots of all trees are linked using a circular, **doubly linked list**. The children of each node are also linked using such a list. For each node, we maintain its number of children and whether the node is marked. Moreover we maintain a pointer to the root containing the minimum key.

Operation **find minimum** is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost is constant.

As mentioned above, **merge** is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time.

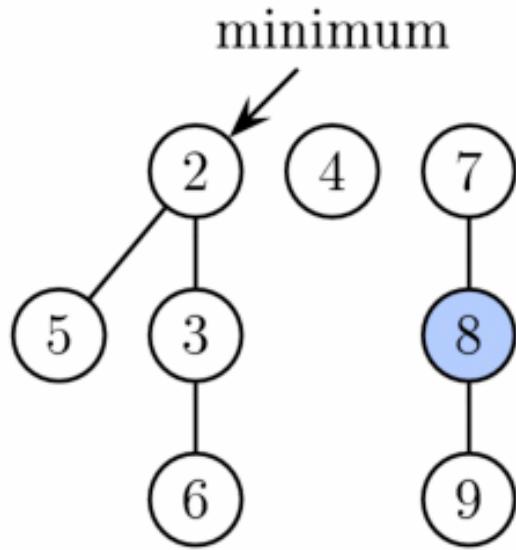
Operation **insert** works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant.

Operation **extract minimum** (same as *delete minimum*) operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was  $d$ ,



Fibonacci heap from Figure 1 after first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.

it takes time  $O(d)$  to process all new roots and the potential increases by  $d-1$ . Therefore the amortized running time of this phase is  $O(d) = O(\log n)$ .

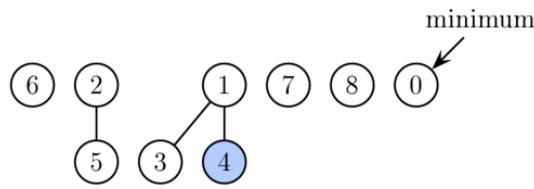


Fibonacci heap from Figure 1 after extract minimum is completed. First, nodes 3 and 6 are linked together. Then the result is linked with tree rooted at node 2. Finally, the new minimum is found.

However to complete the extract minimum operation, we need to update the pointer to the root with minimum key. Unfortunately there may be up to  $n$  roots we need to check. In the second phase we therefore decrease the number of roots by successively linking together roots of the same degree. When two roots  $u$  and  $v$  have the same degree, we make one of them a child of the other so that the one with the smaller key remains the root. Its degree will increase by one. This is repeated until every root has a different degree. To find trees of the same degree efficiently we use an array of length  $O(\log n)$  in which we keep a pointer to one root of each degree. When a second root is found of the same degree, the two are linked and

the array is updated. The actual running time is  $O(\log n + m)$  where  $m$  is the number of roots at the beginning of the second phase. At the end we will have at most  $O(\log n)$  roots (because each has a different degree). Therefore the difference in the potential function from before this phase to after it is:  $O(\log n) - m$ , and the amortized running time is then at most  $O(\log n + m) + c(O(\log n) - m)$ . With a sufficiently large choice of  $c$ , this simplifies to  $O(\log n)$ .

In the third phase we check each of the remaining roots and find the minimum. This takes  $O(\log n)$  time and the potential does not change. The overall amortized running time of extract minimum is therefore  $O(\log n)$ .



Fibonacci heap from Figure 1 after decreasing key of node 9 to 0. This node as well as its two marked ancestors are cut from the tree rooted at 1 and placed as new roots.

Operation **decrease key** will take the node, decrease the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent. If the parent is not a root, it is marked. If it has been marked already, it is cut as well and its parent is marked. We continue upwards until we reach either the root or an unmarked node. In the process we create some number, say  $k$ , of new trees. Each of these new trees except possibly the first one was marked originally but as a root it will become unmarked. One node can become marked. Therefore the number of marked nodes changes by  $-(k - 1) + 1 = -k + 2$ . Combining these 2 changes, the potential changes by  $2(-k + 2) + k = -k + 4$ . The actual time to perform the cutting was  $O(k)$ , therefore (again with a sufficiently large choice of  $c$ ) the amortized running time is constant.

Finally, operation **delete** can be implemented simply by decreasing the key of the element to be deleted to minus infinity, thus turning it into the minimum of the whole heap. Then we call extract minimum to remove it. The amortized running time of this operation is  $O(\log n)$ .

### 7.4.3 Proof of degree bounds

The amortized performance of a Fibonacci heap depends on the degree (number of children) of any tree root being  $O(\log n)$ , where  $n$  is the size of the heap. Here we show that the size of the (sub)tree rooted at any node  $x$  of degree  $d$  in the heap must have size at least  $F_{d+2}$ , where  $F_k$  is the  $k$ th Fibonacci number. The degree bound follows from this and the fact (easily proved by induction) that  $F_{d+2} \geq \varphi^d$  for all integers  $d \geq 0$ , where

$\varphi = (1 + \sqrt{5})/2 \doteq 1.618$ . (We then have  $n \geq F_{d+2} \geq \varphi^d$ , and taking the log to base  $\varphi$  of both sides gives  $d \leq \log_\varphi n$  as required.)

Consider any node  $x$  somewhere in the heap ( $x$  need not be the root of one of the main trees). Define **size**( $x$ ) to be the size of the tree rooted at  $x$  (the number of descendants of  $x$ , including  $x$  itself). We prove by induction on the height of  $x$  (the length of a longest simple path from  $x$  to a descendant leaf), that **size**( $x$ )  $\geq F_{d+2}$ , where  $d$  is the degree of  $x$ .

**Base case:** If  $x$  has height 0, then  $d = 0$ , and **size**( $x$ ) = 1 =  $F_2$ .

**Inductive case:** Suppose  $x$  has positive height and degree  $d > 0$ . Let  $y_1, y_2, \dots, y_d$  be the children of  $x$ , indexed in order of the times they were most recently made children of  $x$  ( $y_1$  being the earliest and  $y_d$  the latest), and let  $c_1, c_2, \dots, c_d$  be their respective degrees. We **claim** that  $c_i \geq i - 2$  for each  $i$  with  $2 \leq i \leq d$ : Just before  $y_i$  was made a child of  $x$ ,  $y_1, \dots, y_{i-1}$  were already children of  $x$ , and so  $x$  had degree at least  $i - 1$  at that time. Since trees are combined only when the degrees of their roots are equal, it must have been that  $y_i$  also had degree at least  $i - 1$  at the time it became a child of  $x$ . From that time to the present,  $y_i$  can only have lost at most one child (as guaranteed by the marking process), and so its current degree  $c_i$  is at least  $i - 2$ . This proves the **claim**.

Since the heights of all the  $y_i$  are strictly less than that of  $x$ , we can apply the inductive hypothesis to them to get **size**( $y_i$ )  $\geq F_{c_i+2} \geq F_{(i-2)+2} = F_i$ . The nodes  $x$  and  $y_1$  each contribute at least 1 to **size**( $x$ ), and so we have

$$\text{size}(x) \geq 2 + \sum_{i=2}^d \text{size}(y_i) \geq 2 + \sum_{i=2}^d F_i = 1 + \sum_{i=0}^d F_i.$$

A routine induction proves that  $1 + \sum_{i=0}^d F_i = F_{d+2}$  for any  $d \geq 0$ , which gives the desired lower bound on **size**( $x$ ).

### 7.4.4 Worst case

Although the total running time of a sequence of operations starting with an empty structure is bounded by the bounds given above, some (very few) operations in the sequence can take very long to complete (in particular delete and delete minimum have linear running time in the worst case). For this reason Fibonacci heaps and other amortized data structures may not be appropriate for real-time systems. It is possible to create a data structure which has the same worst-case performance as the Fibonacci heap has amortized performance.<sup>[3][4]</sup> One such structure, the **Brodal queue**, is, in the words of the creator, "quite complicated" and "[not] applicable in practice." Created in 2012, the strict Fibonacci heap is a simpler (compared to Brodal's) structure with the same worst-case bounds. It is unknown whether the strict Fibonacci heap is efficient in practice. The run-relaxed heaps of Driscoll et al. give good worst-case performance for all

Fibonacci heap operations except merge.

### 7.4.5 Summary of running times

In the following time complexities<sup>[1]</sup>  $O(f)$  is an asymptotic upper bound and  $\Theta(f)$  is an asymptotically tight bound (see Big O notation). Function names assume a min-heap.

- [1] Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with  $n$  elements can be constructed bottom-up in  $O(n)$ .<sup>[7]</sup>
- [2] Amortized time.
- [3] Bounded by  $\Omega(\log \log n)$ ,  $O(2^{2\sqrt{\log \log n}})$ <sup>[2][10]</sup>
- [4]  $n$  is the size of the larger heap and  $m$  is the size of the smaller heap.
- [5]  $n$  is the size of the larger heap.

### 7.4.6 Practical considerations

Fibonacci heaps have a reputation for being slow in practice<sup>[11]</sup> due to large memory consumption per node and high constant factors on all operations.<sup>[12]</sup>

### 7.4.7 References

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 20: Fibonacci Heaps, pp.476–497. Third edition p518.
- [2] Fredman, Michael Lawrence; Tarjan, Robert E. (1987). “Fibonacci heaps and their uses in improved network optimization algorithms” (PDF). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi:10.1145/28869.28874.
- [3] Gerth Stølting Brodal (1996), “Worst-Case Efficient Priority Queues”, *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms* (Society for Industrial and Applied Mathematics): 52–58, doi:10.1145/313852.313883, ISBN 0-89871-366-8, CiteSeerX: 10.1.1.43.8133
- [4] Brodal, G. S. L.; Lagogiannis, G.; Tarjan, R. E. (2012). *Strict Fibonacci heaps* (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12. p. 1177. doi:10.1145/2213977.2214082. ISBN 9781450312455.
- [5] Iacono, John (2000), “Improved upper bounds for pairing heaps”, *Proc. 7th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science **1851**, Springer-Verlag, pp. 63–77, doi:10.1007/3-540-44985-X\_5

- [6] Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 52-58, 1996

- [7] Goodrich, Michael T.; Tamassia, Roberto (2004). “7.3.6. Bottom-Up Heap Construction”. *Data Structures and Algorithms in Java* (3rd ed.). pp. 338–341.
- [8] Haeupler, Bernhard; Sen, Siddhartha; Tarjan, Robert E. (2009). “Rank-pairing heaps” (PDF). *SIAM J. Computing*: 1463–1485.
- [9] Brodal, G. S. L.; Lagogiannis, G.; Tarjan, R. E. (2012). *Strict Fibonacci heaps* (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12. p. 1177. doi:10.1145/2213977.2214082. ISBN 9781450312455.
- [10] Pettie, Seth (2005). “Towards a Final Analysis of Pairing Heaps” (PDF). *Max Planck Institut für Informatik*.
- [11] <http://www.cs.princeton.edu/~{}wayne/kleinberg-tardos/pdf/FibonacciHeaps.pdf>, p. 79
- [12] <http://web.stanford.edu/class/cs166/lectures/07/Small07.pdf>, p. 72

### 7.4.8 External links

- Java applet simulation of a Fibonacci heap
- MATLAB implementation of Fibonacci heap
- De-recursive and memory efficient C implementation of Fibonacci heap (free/libre software, CeCILL-B license)
- Ruby implementation of the Fibonacci heap (with tests)
- Pseudocode of the Fibonacci heap algorithm
- Various Java Implementations for Fibonacci heap

## 7.5 2-3 heap

In computer science, a **2-3 heap** is a data structure, a variation on the heap, designed by Tadao Takaoka in 1999. The structure is similar to the Fibonacci heap, and borrows from the **2-3 tree**.

Time costs for some common heap operations are:

- *Delete-min* takes  $O(\log(n))$  amortized time.
- *Decrease-key* takes constant amortized time.
- *Insertion* takes constant amortized time.

### 7.5.1 References

- Tadao Takaoka. *Theory of 2-3 Heaps*, Cocoon (1999).

## 7.6 Pairing heap

A **pairing heap** is a type of **heap** data structure with relatively simple implementation and excellent practical amortized performance. Pairing heaps are **heap-ordered** multiway **tree structures**, and can be considered simplified **Fibonacci heaps**. They are considered a “robust choice” for implementing such algorithms as Prim’s **MST algorithm**,<sup>[1]:231</sup> and support the following operations (assuming a min-heap):

- *find-min*: simply return the top element of the heap.
- *merge*: compare the two root elements, the smaller remains the root of the result, the larger element and its subtree is appended as a child of this root.
- *insert*: create a new heap for the inserted element and *merge* into the original heap.
- *decrease-key* (optional): remove the subtree rooted at the key to be decreased, replace the key with a smaller key, then *merge* the result back into the heap.
- *delete-min*: remove the root and *merge* its subtrees. Various strategies are employed.

The analysis of pairing heaps’ time complexity was initially inspired by that of **splay trees**.<sup>[2]</sup> The amortized time per *delete-min* is  $O(\log n)$ .<sup>[2]</sup> The operations *find-min*, *merge*, and *insert* run in constant time,  $O(1)$ .<sup>[3]</sup>

Determining the precise asymptotic running time of pairing heaps when a *decrease-key* operation is needed has turned out to be difficult. Initially, the time complexity of this operation was conjectured on empirical grounds to be  $O(1)$ ,<sup>[4]</sup> but **Fredman** proved that the amortized time per *decrease-key* is at least  $\Omega(\log \log n)$  for some sequences of operations.<sup>[5]</sup> **Pettie** then derived an upper bound of  $O(2^{2\sqrt{\log \log n}})$  amortized time for *decrease-key*, which is  $o(\log n)$ .<sup>[6]</sup> No tight  $\Theta(\log \log n)$  bound is known.<sup>[6]</sup>

Although this is worse than other priority queue algorithms such as **Fibonacci heaps**, which perform *decrease-key* in  $O(1)$  amortized time, the performance in practice is excellent. **Stasko** and **Vitter**<sup>[4]</sup> and **Moret** and **Shapiro**<sup>[7]</sup> conducted experiments on pairing heaps and other heap data structures. They concluded that the pairing heap is as fast as, and often faster than, other efficient data structures like the **binary heaps**.

### 7.6.1 Structure

A pairing heap is either an empty heap, or a pair consisting of a root element and a possibly empty list of pairing heaps. The heap ordering property requires that all the root elements of the subheaps in the list are not smaller than the root element of the heap. The following description assumes a purely functional heap that does not support the *decrease-key* operation.

**type** PairingHeap[Elem] = Empty | Heap(elem: Elem, subheaps: List[PairingHeap[Elem]])

A pointer-based implementation for **RAM** machines, supporting *decrease-key*, can be achieved using three pointers per node, by representing the children of a node by a **singly-linked list**: a pointer to the node’s first child, one to its next sibling, and one to the parent. Alternatively, the parent pointer can be omitted by letting the last child point back to the parent, if a single boolean flag is added to indicate “end of list”. This achieves a more compact structure at the expense of a constant overhead factor per operation.<sup>[2]</sup>

### 7.6.2 Operations

#### find-min

The function *find-min* simply returns the root element of the heap:

```
function find-min(heap) if heap == Empty error else return heap.elem
```

#### merge

Merging with an empty heap returns the other heap, otherwise a new heap is returned that has the minimum of the two root elements as its root element and just adds the heap with the larger root to the list of subheaps:

```
function merge(heap1, heap2) if heap1 == Empty return heap2 elseif heap2 == Empty return heap1 elseif heap1.elem < heap2.elem return Heap(heap1.elem, heap2 :: heap1.subheaps) else return Heap(heap2.elem, heap1 :: heap2.subheaps)
```

#### insert

The easiest way to insert an element into a heap is to merge the heap with a new heap containing just this element and an empty list of subheaps:

```
function insert(elem, heap) return merge(Heap(elem, []), heap)
```

#### delete-min

The only non-trivial fundamental operation is the deletion of the minimum element from the heap. The standard strategy first merges the subheaps in pairs (this is the step that gave this datastructure its name) from left to right and then merges the resulting list of heaps from right to left:

```
function delete-min(heap) if heap == Empty error else return merge-pairs(heap.subheaps)
```

This uses the auxiliary function *merge-pairs*:

```

function merge-pairs(l) if length(l) == 0 return
Empty elsif length(l) == 1 return l[0] else return
merge(merge(l[0], l[1]), merge-pairs(l[2..]))

```

That this does indeed implement the described two-pass left-to-right then right-to-left merging strategy can be seen from this reduction:

```

merge-pairs([H1, H2, H3, H4, H5, H6, H7]) =>
merge(merge(H1, H2), merge-pairs([H3, H4, H5, H6, H7])) # merge H1 and H2 to H12, then the rest of
the list => merge(H12, merge(merge(H3, H4), merge-
pairs([H5, H6, H7]))) # merge H3 and H4 to H34,
then the rest of the list => merge(H12, merge(H34,
merge(merge(H5, H6), merge-pairs([H7])))) # merge H5
and H6 to H56, then the rest of the list => merge(H12,
merge(H34, merge(H56, H7))) # switch direction, merge
the last two resulting heaps, giving H567 => merge(H12,
merge(H34, H567)) # merge the last two resulting heaps,
giving H34567 => merge(H12, H34567) # finally, merge
the first merged pair with the result of merging the rest
=> H1234567

```

### 7.6.3 Summary of running times

In the following time complexities<sup>[8]</sup>  $O(f)$  is an asymptotic upper bound and  $\Theta(f)$  is an asymptotically tight bound (see Big O notation). Function names assume a min-heap.

- [1] Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with  $n$  elements can be constructed bottom-up in  $O(n)$ .<sup>[10]</sup>
- [2] Amortized time.
- [3] Bounded by  $\Omega(\log \log n)$ ,  $O(2^2 \sqrt{\log \log n})$ <sup>[13][14]</sup>
- [4]  $n$  is the size of the larger heap and  $m$  is the size of the smaller heap.
- [5]  $n$  is the size of the larger heap.

### 7.6.4 References

- [1] Mehlhorn, Kurt; Sanders, Peter (2008). *Algorithms and Data Structures: The Basic Toolbox*. Springer.
- [2] Fredman, Michael L.; Sedgewick, Robert; Sleator, Daniel D.; Tarjan, Robert E. (1986). “The pairing heap: a new form of self-adjusting heap” (PDF). *Algorithmica* **1** (1): 111–129. doi:10.1007/BF01840439.
- [3] Iacono, John (2000). *Improved upper bounds for pairing heaps* (PDF). Proc. 7th Scandinavian Workshop on Algorithm Theory. Lecture Notes in Computer Science **1851**. Springer-Verlag. pp. 63–77. doi:10.1007/3-540-44985-X\_5. ISBN 978-3-540-67690-4.

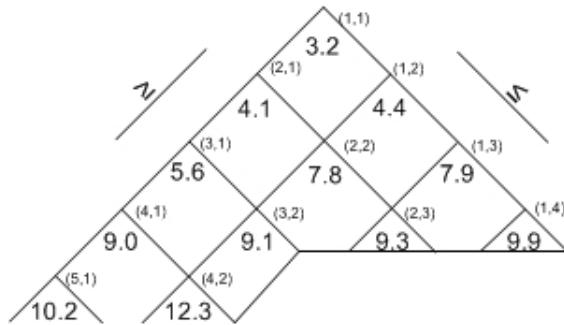
- [4] Stasko, John T.; Vitter, Jeffrey S. (1987), “Pairing heaps: experiments and analysis”, *Communications of the ACM* **30** (3): 234–249, doi:10.1145/214748.214759, CiteSeerX: 10.1.1.106.2988
- [5] Fredman, Michael L. (1999). “On the efficiency of pairing heaps and related data structures” (PDF). *Journal of the ACM* **46** (4): 473–501. doi:10.1145/320211.320214.
- [6] Pettie, Seth (2005), “Towards a final analysis of pairing heaps” (PDF), *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science*, pp. 174–183, doi:10.1109/SFCS.2005.75, ISBN 0-7695-2468-0
- [7] Moret, Bernard M. E.; Shapiro, Henry D. (1991), “An empirical analysis of algorithms for constructing a minimum spanning tree”, *Proc. 2nd Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **519**, Springer-Verlag, pp. 400–411, doi:10.1007/BFb0028279, ISBN 3-540-54343-0, CiteSeerX: 10.1.1.53.5960
- [8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest (1990): Introduction to algorithms. MIT Press / McGraw-Hill.
- [9] Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 52-58, 1996
- [10] Goodrich, Michael T.; Tamassia, Roberto (2004). “7.3.6. Bottom-Up Heap Construction”. *Data Structures and Algorithms in Java* (3rd ed.). pp. 338–341.
- [11] Haeupler, Bernhard; Sen, Siddhartha; Tarjan, Robert E. (2009). “Rank-pairing heaps” (PDF). *SIAM J. Computing*: 1463–1485.
- [12] Brodal, G. S. L.; Lagogiannis, G.; Tarjan, R. E. (2012). *Strict Fibonacci heaps* (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12. p. 1177. doi:10.1145/2213977.2214082. ISBN 9781450312455.
- [13] Fredman, Michael Lawrence; Tarjan, Robert E. (1987). “Fibonacci heaps and their uses in improved network optimization algorithms” (PDF). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi:10.1145/28869.28874.
- [14] Pettie, Seth (2005). “Towards a Final Analysis of Pairing Heaps” (PDF). *Max Planck Institut für Informatik*.

### 7.6.5 External links

- Louis Wasserman discusses pairing heaps and their implementation in Haskell in *The Monad Reader*, Issue 16 (pp. 37–52).
- pairing heaps, Sartaj Sahni
- Amr Elmasry (2009), “Pairing Heaps with  $O(\log \log n)$  decrease Cost” (PDF), *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)* (New York): 471–476

## 7.7 Beap

**Beap**, or **bi-parental heap**, is a data structure where a node usually has two parents (unless it is the first or last on a level) and two children (unless it is on the last level). Unlike a heap, a beap allows sublinear search. The beap was introduced by Ian Munro and Hendra Suwanda. A related data structure is the Young tableau.



Beap

### 7.7.1 Performance

The height of the structure is approximately  $\sqrt{n}$ . Also, assuming the last level is full, the number of elements on that level is also  $\sqrt{n}$ . In fact, because of these properties all basic operations (insert, remove, find) run in  $O(\sqrt{n})$  time on average. Find operations in the heap can be  $O(n)$  in the worst case. Removal and insertion of new elements involves propagation of elements up or down (much like in a heap) in order to restore the beap invariant. An additional perk is that beap provides constant time access to the smallest element and  $O(\sqrt{n})$  time for the maximum element.

Actually, a  $O(\sqrt{n})$  find operation can be implemented if parent pointers at each node are maintained. You would start at the absolute bottom-most element of the top node (similar to the left-most child in a heap) and move either up or right to find the element of interest.

### 7.7.2 References

- J. Ian Munro and Hendra Suwanda. “Implicit data structures for fast search and update”. *Journal of Computer and System Sciences*, 21(2):236-250, Oct.1980.
- Williams, J. W. J. (Jun 1964). “Algorithm 232 - Heapsort”. *Communications of the ACM* 7 (6): 347-348.

## 7.8 Leftist tree

In computer science, a **leftist tree** or **leftist heap** is a priority queue implemented with a variant of a binary heap. Every node has an *s-value* which is the distance to the nearest leaf. In contrast to a *binary heap*, a leftist tree attempts to be very unbalanced. In addition to the heap property, leftist trees are maintained so the right descendant of each node has the lower s-value.

The height-biased leftist tree was invented by Clark Allan Crane.<sup>[1]</sup> The name comes from the fact that the left subtree is usually taller than the right subtree.

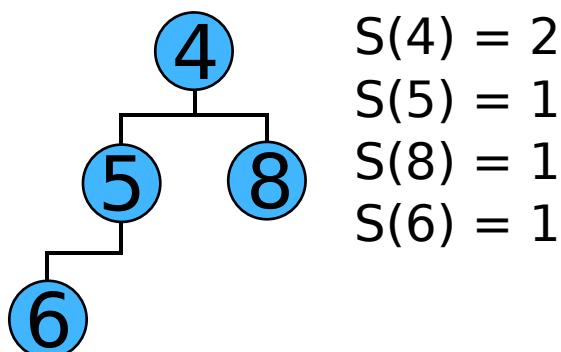
When inserting a new node into a tree, a new one-node tree is created and merged into the existing tree. To delete a minimum item, we remove the root and the left and right sub-trees are then merged. Both these operations take  $O(\log n)$  time. For insertions, this is slower than binomial heaps which support insertion in amortized constant time,  $O(1)$  and  $O(\log n)$  worst-case.

Leftist trees are advantageous because of their ability to merge quickly, compared to binary heaps which take  $\Theta(n)$ . In almost all cases, the merging of **skew heaps** has better performance. However merging leftist heaps has worst-case  $O(\log n)$  complexity while merging skew heaps has only amortized  $O(\log n)$  complexity.

### 7.8.1 Bias

The usual leftist tree is a *height-biased* leftist tree.<sup>[1]</sup> However, other biases can exist, such as in the *weight-biased* leftist tree.<sup>[2]</sup>

### 7.8.2 S-value



S-values of a leftist tree

The **s-value** (or **rank**) of a node is the distance from that node to the nearest leaf of the extended binary representation of the tree. The extended representation (not shown) fills out the tree so that each node has 2 children (adding a total of 5 leaves here). The minimum distance to these leaves are marked in the diagram. Thus s-value of 4 is 2, since the closest leaf is that of 8 --if 8 were extended. The

s-value of 5 is 1 since its extended representation would have one leaf itself.

### 7.8.3 Merging height biased leftist trees

Merging two nodes together depends on whether the tree is a min or max height biased leftist tree. For a min height biased leftist tree, set the higher valued node as the right child of the lower valued node. If the lower valued node already has a right child, then merge the higher valued node with the sub-tree rooted by the right child of the lower valued node.

After merging, the s-value of the lower valued node must be updated (see above section, s-value). Now check if the lower valued node has a left child. If it does not, then move the right child to the left. If it does have a left child, then the child with the highest s-value should go on the left.

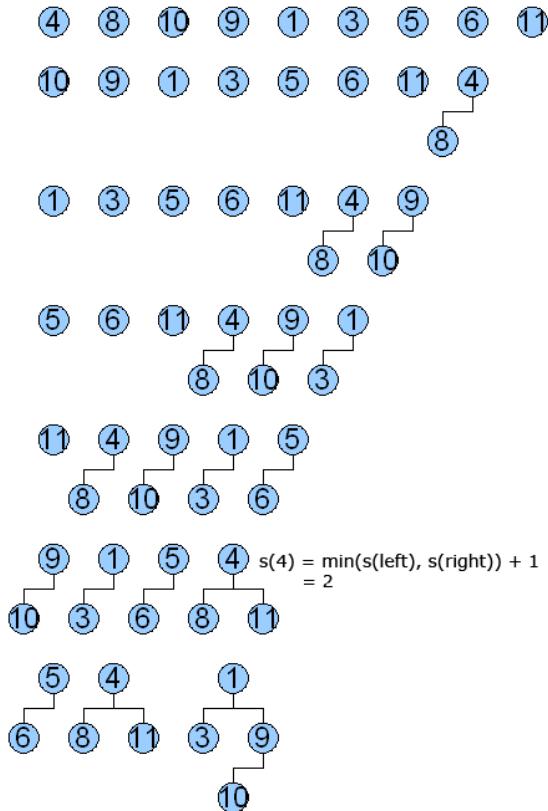
#### Java code for merging a min height biased leftist tree

```
public Node merge(Node x, Node y) { if(x == null) return y; if(y == null) return x; // if this was a max height biased leftist tree, then the // next line would be: if(x.element < y.element) if(x.element.compareTo(y.element) > 0) { // x.element > y.element Node temp = x; x = y; y = temp; } x.rightChild = merge(x.rightChild, y); if(x.leftChild == null) { // left child doesn't exist, so move right child to the left side x.leftChild = x.rightChild; x.rightChild = null; } else { // left child does exist, so compare s-values if(x.leftChild.s < x.rightChild.s) { Node temp = x.leftChild; x.leftChild = x.rightChild; x.rightChild = temp; } // since we know the right child has the lower s-value, we can just // add one to its s-value x.s = x.rightChild.s + 1; } return x; }
```

### 7.8.4 Initializing a height biased leftist tree

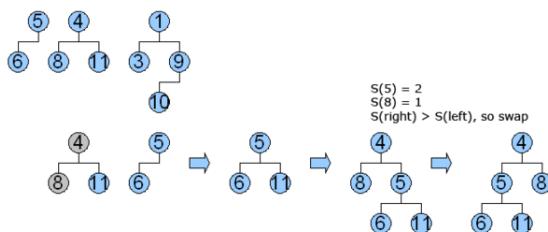
Initializing a height biased leftist tree is primarily done in one of two ways. The first is to merge each node one at a time into one HBLT. This process is inefficient and takes  $O(n \log n)$  time. The other approach is to use a queue to store each node and resulting tree. The first two items in the queue are removed, merged, and placed back into the queue. This can initialize a HBLT in  $O(n)$  time. This approach is detailed in the three diagrams supplied. A min height biased leftist tree is shown.

To initialize a min HBLT, place each element to be added to the tree into a queue. In the example (see Part 1 to the left), the set of numbers [4, 8, 10, 9, 1, 3, 5, 6, 11] are initialized. Each line of the diagram represents another cycle of the algorithm, depicting the contents of the queue. The first five steps are easy to follow. Notice that the freshly created HBLT is added to the end



Initializing a min HBLT - Part 1

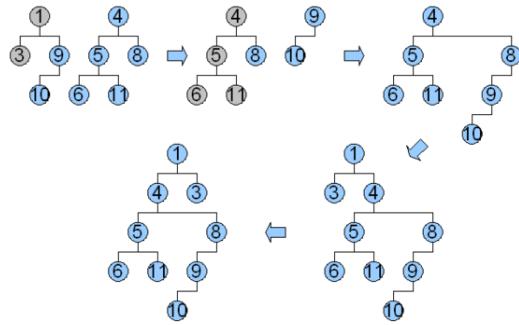
of the queue. In the fifth step, the first occurrence of an s-value greater than 1 occurs. The sixth step shows two trees merged with each other, with predictable results.



Initializing a min HBLT - Part 2

In part 2 a slightly more complex merge happens. The tree with the lower value (tree x) has a right child, so merge must be called again on the subtree rooted by tree x's right child and the other tree. After the merge with the subtree, the resulting tree is put back into tree x. The s-value of the right child (s=2) is now greater than the s-value of the left child (s=1), so they must be swapped. The s-value of the root node 4 is also now 2.

Part 3 is the most complex. Here, we recursively call merge twice (each time with the right child's subtree that is not grayed out). This uses the same process described for part 2.



Initializing a min HBLT - Part 3

## 7.8.5 References

- [1] Clark A. Crane (1972), *Linear Lists and Priority Queues as Balanced Binary Trees*, Department of Computer Science, Stanford University.
- [2] Seonghun Cho and Sartaj Sahni (1996), “Weight Biased Leftist Trees and Modified Skip Lists”, *Journal of Experimental Algorithms* 3

## 7.8.6 External links

- [Leftist Trees, Sartaj Sahni](#)

## 7.8.7 Further reading

- Robert E. Tarjan (1983). *Data Structures and Network Algorithms*. SIAM. pp. 38–42. ISBN 978-0-89871-187-5.
- Dinesh P. Mehta; Sartaj Sahni (28 October 2004). *Handbook of Data Structures and Applications. Chapter 5: Leftist trees*. CRC Press. ISBN 978-1-4200-3517-9.

## 7.9 Skew heap

A **skew heap** (or **self-adjusting heap**) is a heap data structure implemented as a **binary tree**. Skew heaps are advantageous because of their ability to merge more quickly than binary heaps. In contrast with **binary heaps**, there are no structural constraints, so there is no guarantee that the height of the tree is logarithmic. Only two conditions must be satisfied:

- The general heap order must be enforced
- Every operation (add, remove\_min, merge) on two skew heaps must be done using a special **skew heap merge**.

A skew heap is a self-adjusting form of a leftist heap which attempts to maintain balance by unconditionally

swapping all nodes in the merge path when merging two heaps. (The merge operation is also used when adding and removing values.)

With no structural constraints, it may seem that a skew heap would be horribly inefficient. However, **amortized complexity analysis** can be used to demonstrate that all operations on a skew heap can be done in  $O(\log n)$ .<sup>[1]</sup>

### 7.9.1 Definition

Skew heaps may be described with the following recursive definition:

- A heap with only one element is a skew heap.
- The result of **skew merging** two skew heaps  $sh_1$  and  $sh_2$  is also a skew heap.

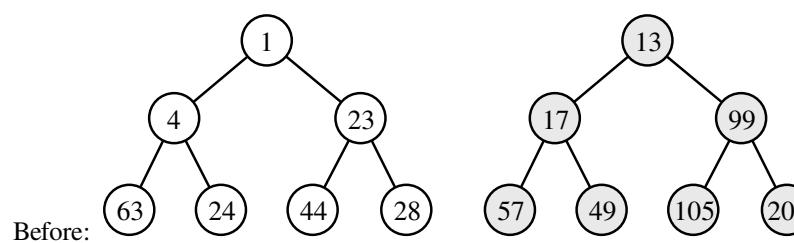
### 7.9.2 Operations

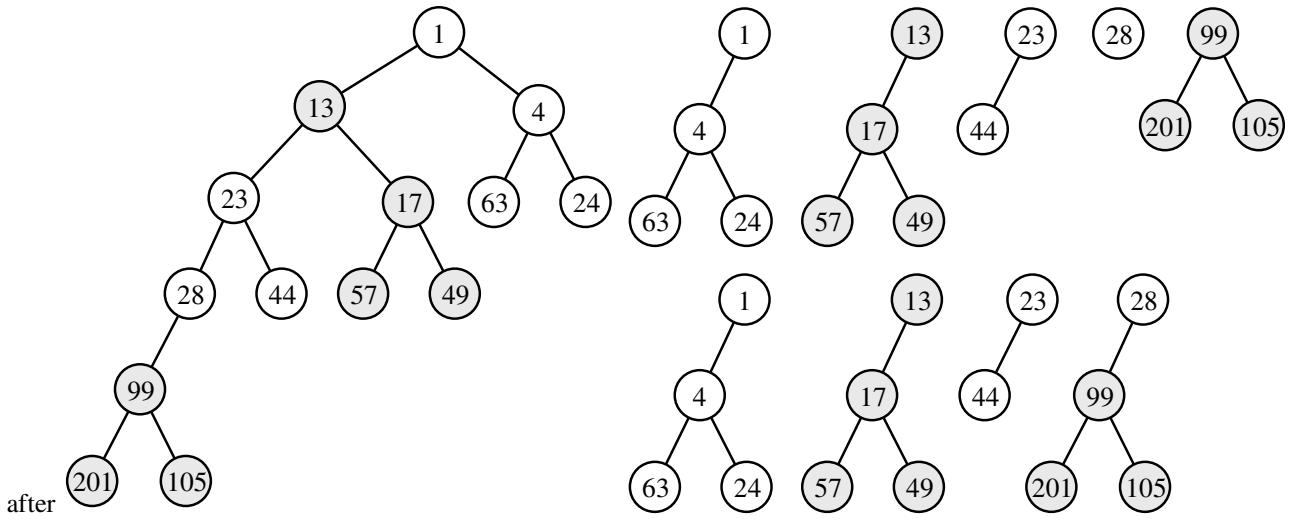
#### Merging two heaps

When two skew heaps are to be merged, we can use a similar process as the merge of two leftist heaps:

- Compare roots of two heaps; let  $p$  be the heap with the smaller root, and  $q$  be the other heap. Let  $r$  be the name of the resulting new heap.
- Let the root of  $r$  be the root of  $p$  (the smaller root), and let  $r$ 's right subtree be  $p$ 's left subtree.
- Now, compute  $r$ 's left subtree by recursively merging  $p$ 's right subtree with  $q$ .

```
template<class T, class CompareFunction>
SkewNode<T>* CSkewHeap<T, CompareFunction>::_Merge(SkewNode<T>* root_1, SkewNode<T>* root_2) {
 SkewNode<T>* firstRoot = root_1;
 SkewNode<T>* secondRoot = root_2;
 if(firstRoot == NULL) return secondRoot;
 else if(secondRoot == NULL) return firstRoot;
 if(sh_compare->Less(firstRoot->key, secondRoot->key)) {
 SkewNode<T>* tempHeap = firstRoot->rightNode;
 firstRoot->rightNode = firstRoot->leftNode;
 firstRoot->leftNode = _Merge(secondRoot, tempHeap);
 return firstRoot;
 } else return _Merge(secondRoot, firstRoot);
}
```

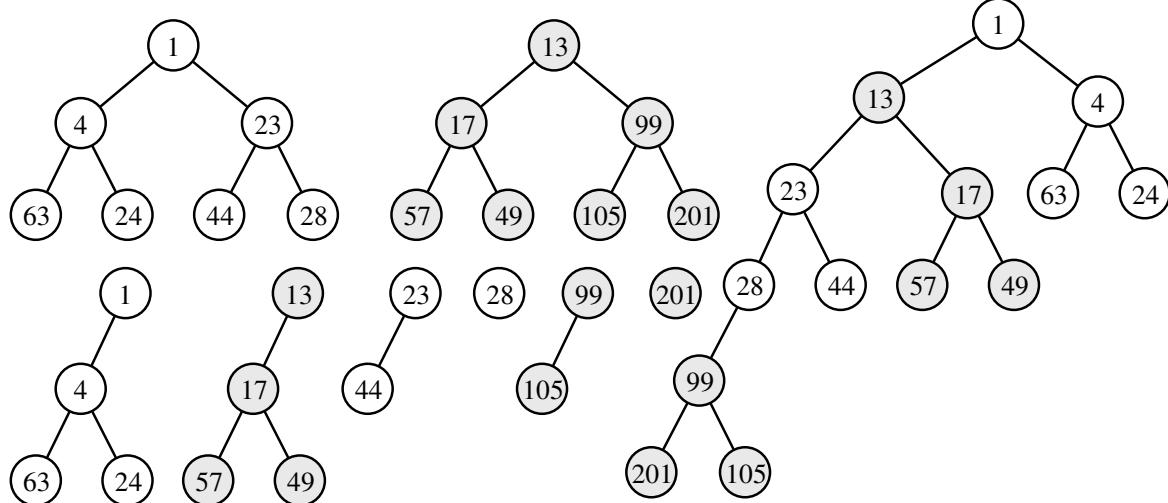
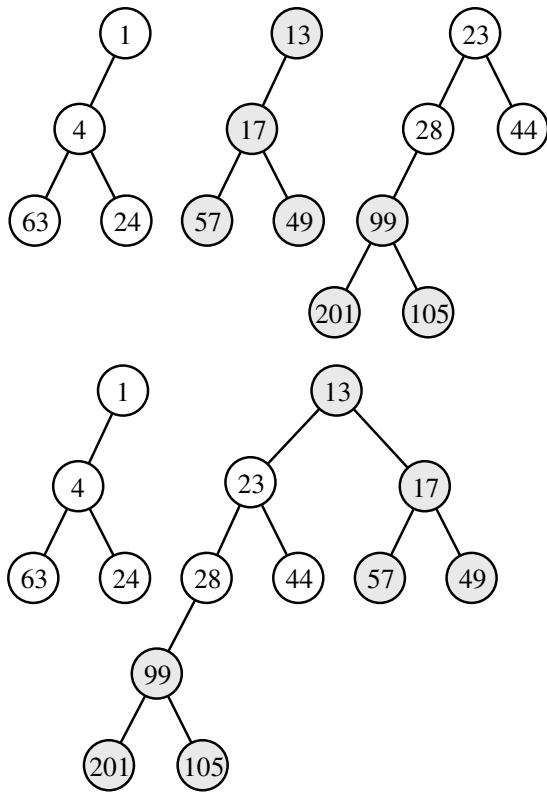




### Non-recursive merging

Alternatively, there is a non-recursive approach which is more wordy, and does require some sorting at the outset.

- Split each heap into subtrees by cutting every right-most path. (From the root node, sever the right child and make the right child its own subtree.) This will result in a set of trees in which the root either only has a left child or no children at all.
- Sort the subtrees in ascending order based on the value of the root node of each subtree.
- While there are still multiple subtrees, iteratively recombine the last two (from right to left).
  - If the root of the second-to-last subtree has a left child, swap it to be the right child.
  - Link the root of the last subtree as the left child of the second-to-last subtree.



### Adding values

Adding a value to a skew heap is like merging a tree with one node together with the original tree.

### Removing values

Removing the first value in a heap can be accomplished by removing the root and merging its child subtrees.

### Implementation

In many functional languages, skew heaps become extremely simple to implement. Here is a complete sample implementation in Haskell.

```
data SkewHeap a = Empty | Node a (SkewHeap a)
(SkewHeap a) singleton :: Ord a => a -> SkewHeap a
singleton x = Node x Empty Empty union :: Ord a =>
SkewHeap a -> SkewHeap a -> SkewHeap a
Empty `union` t2 = t2 t1 `union` Empty = t1 t1@(Node x1 l1
r1) `union` t2@(Node x2 l2 r2) | x1 <= x2 = Node x1
(t2 `union` r1) l1 | otherwise = Node x2 (t1 `union` r2)
l2 insert :: Ord a => a -> SkewHeap a -> SkewHeap a
insert x heap = singleton x `union` heap extractMin :: Ord a => SkewHeap a -> Maybe (a, SkewHeap a)
extractMin Empty = Nothing extractMin (Node x l r) = Just (x, l `union` r)
```

### 7.9.3 References

- Sleator, Daniel Dominic; Tarjan, Robert Endre (1986). “Self-Adjusting Heaps”. *SIAM Journal on Computing* **15** (1): 52–69. doi:10.1137/0215004. ISSN 0097-5397.
- CSE 4101 lecture notes, York University

[1] <http://www.cse.yorku.ca/~{}andy/courses/4101/lecture-notes/LN5.pdf>

### 7.9.4 External links

- Animations comparing leftist heaps and skew heaps, York University
- Java applet for simulating heaps, Kansas State University

## 7.10 Soft heap

For the Canterbury scene band, see Soft Heap.

In computer science, a **soft heap** is a variant on the simple **heap data structure** that has constant **amortized** time for 5 types of operations. This is achieved by carefully “corrupting” (increasing) the keys of at most a certain number of values in the heap. The constant time operations are:

- **create( $S$ )**: Create a new soft heap
- **insert( $S, x$ )**: Insert an element into a soft heap
- **meld( $S, S'$ )**: Combine the contents of two soft heaps into one, destroying both
- **delete( $S, x$ )**: Delete an element from a soft heap
- **findmin( $S$ )**: Get the element with minimum key in the soft heap

Other heaps such as **Fibonacci heaps** achieve most of these bounds without any corruption, but cannot provide a constant-time bound on the critical *delete* operation. The amount of corruption can be controlled by the choice of a parameter  $\epsilon$ , but the lower this is set, the more time insertions require  $O(\log 1/\epsilon)$  for an error rate of  $\epsilon$ .

More precisely, the guarantee offered by the soft heap is the following: for a fixed value  $\epsilon$  between 0 and 1/2, at any point in time there will be at most  $\epsilon * n$  corrupted keys in the heap, where  $n$  is the number of elements inserted so far. Note that this does not guarantee that only a fixed percentage of the keys *currently* in the heap are corrupted: in an unlucky sequence of insertions and deletions, it can happen that all elements in the heap will have corrupted keys. Similarly, we have no guarantee that in a sequence of elements extracted from the heap with *findmin* and *delete*, only a fixed percentage will have corrupted keys: in an unlucky scenario only corrupted elements are extracted from the heap.

The soft heap was designed by Bernard Chazelle in 2000. The term “corruption” in the structure is the result of what Chazelle called “carpooling” in a soft heap. Each node in the soft heap contains a linked-list of keys and one common key. The common key is an upper bound on the values of the keys in the linked-list. Once a key is added to the linked-list, it is considered corrupted because its value is never again relevant in any of the soft heap operations: only the common keys are compared. This is what makes soft heaps “soft”; you can’t be sure whether or not any particular value you put into it will be corrupted. The purpose of these corruptions is effectively to lower the **information entropy** of the data, enabling the data structure to break through **information-theoretic** barriers regarding heaps.

### 7.10.1 Applications

Surprisingly, despite its limitations and its unpredictable nature, soft heaps are useful in the design of deterministic

algorithms. They were used to achieve the best complexity to date for finding a *minimum spanning tree*. They can also be used to easily build an optimal *selection algorithm*, as well as *near-sorting* algorithms, which are algorithms that place every element near its final position, a situation in which *insertion sort* is fast.

One of the simplest examples is the selection algorithm. Say we want to find the  $k$ th largest of a group of  $n$  numbers. First, we choose an error rate of  $1/3$ ; that is, at most about 33% of the keys we insert will be corrupted. Now, we insert all  $n$  elements into the heap — we call the original values the “correct” keys, and the values stored in the heap the “stored” keys. At this point, at most  $n/3$  keys are corrupted, that is, for at most  $n/3$  keys is the “stored” key larger than the “correct” key, for all the others the stored key equals the correct key.

Next, we delete the minimum element from the heap  $n/3$  times (this is done according to the “stored” key). As the total number of insertions we have made so far is still  $n$ , there are still at most  $n/3$  corrupted keys in the heap. Accordingly, at least  $2n/3 - n/3 = n/3$  of the keys remaining in the heap are not corrupted.

Let  $L$  be the element with the largest correct key among the elements we removed. The stored key of  $L$  is possibly larger than its correct key (if  $L$  was corrupted), and even this larger value is smaller than all the stored keys of the remaining elements in the heap (as we were removing minimums). Therefore, the correct key of  $L$  is smaller than the remaining  $n/3$  uncorrupted elements in the soft heap. Thus,  $L$  divides the elements somewhere between 33%/66% and 66%/33%. We then partition the set about  $L$  using the *partition* algorithm from *quicksort* and apply the same algorithm again to either the set of numbers less than  $L$  or the set of numbers greater than  $L$ , neither of which can exceed  $2n/3$  elements. Since each insertion and deletion requires  $O(1)$  amortized time, the total deterministic time is  $T(n) = T(2n/3) + O(n)$ . Using case 3 of the *master theorem* (with  $\varepsilon=1$  and  $c=2/3$ ), we know that  $T(n) = \Theta(n)$ .

The final algorithm looks like this:

```
function softHeapSelect(a[1..n], k) if k = 1 then return
minimum(a[1..n]) create(S) for i from 1 to n insert(S,
a[i]) for i from 1 to n/3 x := findmin(S) delete(S, x) xIndex
:= partition(a, x) // Returns new index of pivot x if
k < xIndex softHeapSelect(a[1..xIndex-1], k) else soft-
HeapSelect(a[xIndex..n], k-xIndex+1)
```

## 7.10.2 References

- Chazelle, B. 2000. The soft heap: an approximate priority queue with optimal error rate. *J. ACM* 47, 6 (Nov. 2000), 1012-1027.
- Kaplan, H. and Zwick, U. 2009. A simpler implementation and analysis of Chazelle’s soft heaps. In *Proceedings of the Nineteenth Annual ACM -SIAM*

*Symposium on Discrete Algorithms* (New York, New York, January 4—6, 2009). *Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 477-485.

## 7.11 d-ary heap

The  **$d$ -ary heap** or  **$d$ -heap** is a priority queue data structure, a generalization of the *binary heap* in which the nodes have  $d$  children instead of 2.<sup>[1][2][3]</sup> Thus, a binary heap is a 2-heap, and a **ternary heap** is a 3-heap. According to Tarjan<sup>[2]</sup> and Jensen et al.,<sup>[4]</sup>  $d$ -ary heaps were invented by Donald B. Johnson in 1975.<sup>[1]</sup>

This data structure allows decrease priority operations to be performed more quickly than binary heaps, at the expense of slower delete minimum operations. This trade-off leads to better running times for algorithms such as *Dijkstra’s algorithm* in which decrease priority operations are more common than delete min operations.<sup>[1][5]</sup> Additionally,  $d$ -ary heaps have better *memory cache* behavior than a binary heap, allowing them to run more quickly in practice despite having a theoretically larger worst-case running time.<sup>[6][7]</sup> Like binary heaps,  $d$ -ary heaps are an *in-place* data structure that uses no additional storage beyond that needed to store the array of items in the heap.<sup>[2][8]</sup>

### 7.11.1 Data structure

The  $d$ -ary heap consists of an array of  $n$  items, each of which has a priority associated with it. These items may be viewed as the nodes in a complete  $d$ -ary tree, listed in *breadth first traversal order*: the item at position 0 of the array forms the root of the tree, the items at positions  $1-d$  are its children, the next  $d^2$  items are its grandchildren, etc. Thus, the parent of the item at position  $i$  (for any  $i > 0$ ) is the item at position  $\text{floor}((i-1)/d)$  and its children are the items at positions  $di+1$  through  $di+d$ . According to the *heap property*, in a min-heap, each item has a priority that is at least as large as its parent; in a max-heap, each item has a priority that is no larger than its parent.<sup>[2][3]</sup>

The minimum priority item in a min-heap (or the maximum priority item in a max-heap) may always be found at position 0 of the array. To remove this item from the priority queue, the last item  $x$  in the array is moved into its place, and the length of the array is decreased by one. Then, while item  $x$  and its children do not satisfy the heap property, item  $x$  is swapped with one of its children (the one with the smallest priority in a min-heap, or the one with the largest priority in a max-heap), moving it downward in the tree and later in the array, until eventually the heap property is satisfied. The same downward swapping procedure may be used to increase the priority of an item in a min-heap, or to decrease the priority of an item in a

max-heap.<sup>[2][3]</sup>

To insert a new item into the heap, the item is appended to the end of the array, and then while the heap property is violated it is swapped with its parent, moving it upward in the tree and earlier in the array, until eventually the heap property is satisfied. The same upward-swapping procedure may be used to decrease the priority of an item in a min-heap, or to increase the priority of an item in a max-heap.<sup>[2][3]</sup>

To create a new heap from an array of  $n$  items, one may loop over the items in reverse order, starting from the item at position  $n - 1$  and ending at the item at position 0, applying the downward-swapping procedure for each item.<sup>[2][3]</sup>

### 7.11.2 Analysis

In a  $d$ -ary heap with  $n$  items in it, both the upward-swapping procedure and the downward-swapping procedure may perform as many as  $\log_d n = \log n / \log d$  swaps. In the upward-swapping procedure, each swap involves a single comparison of an item with its parent, and takes constant time. Therefore, the time to insert a new item into the heap, to decrease the priority of an item in a min-heap, or to increase the priority of an item in a max-heap, is  $O(\log n / \log d)$ . In the downward-swapping procedure, each swap involves  $d$  comparisons and takes  $O(d)$  time: it takes  $d - 1$  comparisons to determine the minimum or maximum of the children and then one more comparison against the parent to determine whether a swap is needed. Therefore, the time to delete the root item, to increase the priority of an item in a min-heap, or to decrease the priority of an item in a max-heap, is  $O(d \log n / \log d)$ .<sup>[2][3]</sup>

When creating a  $d$ -ary heap from a set of  $n$  items, most of the items are in positions that will eventually hold leaves of the  $d$ -ary tree, and no downward swapping is performed for those items. At most  $n/d + 1$  items are non-leaves, and may be swapped downwards at least once, at a cost of  $O(d)$  time to find the child to swap them with. At most  $n/d^2 + 1$  nodes may be swapped downward two times, incurring an additional  $O(d)$  cost for the second swap beyond the cost already counted in the first term, etc. Therefore, the total amount of time to create a heap in this way is

$$\sum_{i=1}^{\log_d n} \left( \frac{n}{d^i} + 1 \right) O(d) = O(n). \quad [2][3]$$

The exact value of the above (the worst-case number of comparisons during the construction of  $d$ -ary heap) is known to be equal to:

$$\frac{d}{d-1} (n - s_d(n)) - (d - 1 - (n \bmod d)) (e_d(\lfloor \frac{n}{d} \rfloor) + 1), \quad [9]$$

where  $s_d(n)$  is the sum of all digits of the standard base- $d$

representation of  $n$  and  $e_d(n)$  is the exponent of  $d$  in the factorization of  $n$ . This reduces to

$$2n - 2s_2(n) - e_2(n), \quad [9]$$

for  $d = 2$ , and to

$$\frac{3}{2}(n - s_3(n)) - 2e_3(n) - e_3(n - 1), \quad [9]$$

for  $d = 3$ .

The space usage of the  $d$ -ary heap, with insert and delete-min operations, is linear, as it uses no extra storage other than an array containing a list of the items in the heap.<sup>[2][8]</sup> If changes to the priorities of existing items need to be supported, then one must also maintain pointers from the items to their positions in the heap, which again uses only linear storage.<sup>[2]</sup>

### 7.11.3 Applications

Dijkstra's algorithm for shortest paths in graphs and Prim's algorithm for minimum spanning trees both use a min-heap in which there are  $n$  delete-min operations and as many as  $m$  decrease-priority operations, where  $n$  is the number of vertices in the graph and  $m$  is the number of edges. By using a  $d$ -ary heap with  $d = m/n$ , the total times for these two types of operations may be balanced against each other, leading to a total time of  $O(m \log m/n)$  for the algorithm, an improvement over the  $O(m \log n)$  running time of binary heap versions of these algorithms whenever the number of edges is significantly larger than the number of vertices.<sup>[1][5]</sup> An alternative priority queue data structure, the Fibonacci heap, gives an even better theoretical running time of  $O(m + n \log n)$ , but in practice  $d$ -ary heaps are generally at least as fast, and often faster, than Fibonacci heaps for this application.<sup>[10]</sup>

4-heaps may perform better than binary heaps in practice, even for delete-min operations.<sup>[2][3]</sup> Additionally, a  $d$ -ary heap typically runs much faster than a binary heap for heap sizes that exceed the size of the computer's cache memory: A binary heap typically requires more cache misses and virtual memory page faults than a  $d$ -ary heap, each one taking far more time than the extra work incurred by the additional comparisons a  $d$ -ary heap makes compared to a binary heap.<sup>[6][7]</sup>

### 7.11.4 References

- [1] Johnson, D. B. (1975), "Priority queues with update and finding minimum spanning trees", *Information Processing Letters* **4** (3): 53–57, doi:10.1016/0020-0190(75)90001-0.
- [2] Tarjan, R. E. (1983), "3.2.  $d$ -heaps", *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics **44**, Society for Industrial and Applied Mathematics, pp. 34–38.

- [3] Weiss, M. A. (2007), "d-heaps", *Data Structures and Algorithm Analysis* (2nd ed.), Addison-Wesley, p. 216, ISBN 0-321-37013-9.
- [4] Jensen, C.; Katajainen, J.; Vitale, F. (2004), *An extended truth about heaps* (PDF).
- [5] Tarjan (1983), pp. 77 and 91.
- [6] Naor, D.; Martel, C. U.; Matloff, N. S. (1991), "Performance of priority queue structures in a virtual memory environment", *Computer Journal* **34** (5): 428–437, doi:10.1093/comjnl/34.5.428.
- [7] Kamp, Poul-Henning (2010), "You're doing it wrong", *ACM Queue* **8** (6).
- [8] Mortensen, C. W.; Pettie, S. (2005), "The complexity of implicit and space efficient priority queues", *Algorithms and Data Structures: 9th International Workshop, WADS 2005, Waterloo, Canada, August 15–17, 2005, Proceedings*, Lecture Notes in Computer Science **3608**, Springer-Verlag, pp. 49–60, doi:10.1007/11534273\_6, ISBN 978-3-540-28101-6.
- [9] Suchenek, Marek A. (2012), "Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program", *Fundamenta Informaticae* (IOS Press) **120** (1): 75–92, doi:10.3233/FI-2012-751.
- [10] Cherkassky, B. V.; Goldberg, A. V.; Radzik, T. (1996), "Shortest paths algorithms: Theory and experimental evaluation", *Mathematical Programming* **73** (2): 129–174, doi:10.1007/BF02592101.

### 7.11.5 External links

- C++ implementation of generalized heap with D-Heap support

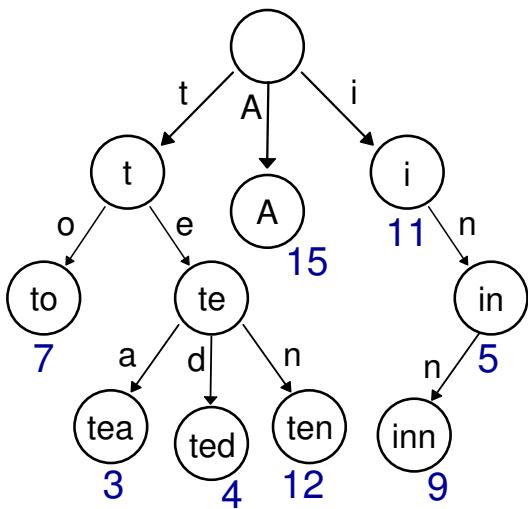
# Chapter 8

## Tries

### 8.1 Trie

This article is about a tree data structure. For the French commune, see [Trie-sur-Baïse](#).

In computer science, a **trie**, also called **digital tree**



A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".

and sometimes **radix tree** or **prefix tree** (as they can be searched by prefixes), is an **ordered tree** data structure that is used to store a **dynamic set** or **associative array** where the keys are usually **strings**. Unlike a **binary search tree**, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common **prefix** of the string associated with that node, and the root is associated with the **empty string**. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest. For the space-optimized presentation of prefix tree, see [compact prefix tree](#).

The term **trie** comes from **retrieval**. This term was coined by [Edward Fredkin](#), who pronounces it /'tri:/ "tree" as in the word **retrieval**.<sup>[1][2]</sup> However, other authors pronounce it /'trai/ "try", in an attempt to distinguish it verbally from "tree".<sup>[1][2][3]</sup>

In the example shown, keys are listed in the nodes and val-

ues below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a **deterministic finite automaton** without loops. Each **finite language** is generated by a trie automaton, and each trie can be compressed into a **DAFSA**.

It is not necessary for keys to be explicitly stored in nodes. (In the figure, words are shown only to illustrate how the trie works.)

Though tries are most commonly keyed by character strings, they don't need to be. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes. In particular, a **bitwise trie** is keyed on the individual bits making up a short, fixed size of bits such as an integer number or memory address.

#### 8.1.1 Applications

##### As a replacement for other data structures

As discussed below, a trie has a number of advantages over binary search trees.<sup>[4]</sup> A trie can also be used to replace a **hash table**, over which it has the following advantages:

- Looking up data in a trie is faster in the worst case,  $O(m)$  time (where  $m$  is the length of a search string), compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect hash table is  $O(N)$  time, but far more typically is  $O(1)$ , with  $O(m)$  time spent evaluating the hash.
- There are no collisions of different keys in a trie.
- Buckets in a trie, which are analogous to hash table buckets that store key collisions, are necessary only if a single key is associated with more than one value.
- There is no need to provide a hash function or to change hash functions as more keys are added to a trie.

- A trie can provide an alphabetical ordering of the entries by key.

Tries do have some drawbacks as well:

- Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random-access time is high compared to main memory.<sup>[5]</sup>
- Some keys, such as floating point numbers, can lead to long chains and prefixes that are not particularly meaningful. Nevertheless a bitwise trie can handle standard IEEE single and double format floating point numbers.
- Some tries can require more space than a hash table, as memory may be allocated for each character in the search string, rather than a single chunk of memory for the whole entry, as in most hash tables.

## Dictionary representation

A common application of a trie is storing a **predictive text** or **autocomplete** dictionary, such as found on a mobile telephone. Such applications take advantage of a trie's ability to quickly search for, insert, and delete entries; however, if storing dictionary words is all that is required (i.e., storage of information auxiliary to each word is not required), a minimal **deterministic acyclic finite state automaton** would use less space than a trie. This is because an acyclic deterministic finite automaton can compress identical branches from the trie which correspond to the same suffixes (or parts) of different words being stored.

Tries are also well suited for implementing approximate matching algorithms,<sup>[6]</sup> including those used in spell checking and hyphenation<sup>[2]</sup> software.

## Algorithms

We can describe lookup (and membership) easily. Given a recursive trie type, storing an optional value at each node, and a list of children tries, indexed by the next character (here, represented as a **Haskell** data type):

```
import Prelude hiding (lookup)
import Data.Map (Map, lookup)
data Trie a = Trie { value :: Maybe a, children :: Map Char (Trie a) }
```

We can look up a value in the trie as follows:

```
find :: String -> Trie a -> Maybe a
find [] t = value t
find (k:ks) t = do ct <- lookup k (children t)
 find ks ct
```

In an imperative style, and assuming an appropriate data type in place, we can describe the same algorithm in

Python (here, specifically for testing membership). Note that `children` is a map of a node's children; and we say that a “terminal” node is one which contains a valid word.

```
def find(node, key):
 for char in key:
 if char not in node.children:
 return None
 else:
 node = node.children[char]
 return node.value
```

Insertion proceeds by walking the trie according to the string to be inserted, then appending new nodes for the suffix of the string that is not contained in the trie. In imperative pseudocode,

```
algorithm insert(root : node, s : string, value : any):
 node = root
 i = 0
 n = length(s)
 while i < n:
 if node.child(s[i]) != nil:
 node = node.child(s[i])
 i = i + 1
 else:
 break
 (* append new nodes, if necessary *)
 while i < n:
 node.child(s[i]) = new node
 node = node.child(s[i])
 i = i + 1
 node.value = value
```

## Sorting

Lexicographic sorting of a set of keys can be accomplished with a simple trie-based algorithm as follows:

- Insert all keys in a trie.
- Output all keys in the trie by means of **pre-order traversal**, which results in output that is in lexicographically increasing order. Pre-order traversal is a kind of **depth-first traversal**.

This algorithm is a form of **radix sort**.

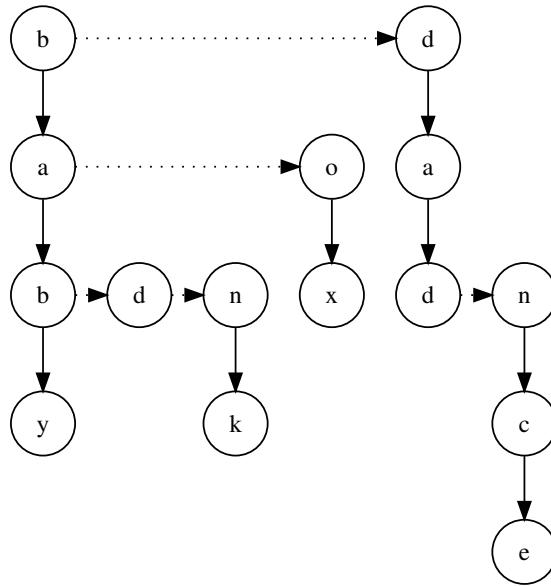
A trie forms the fundamental data structure of **Burstsort**, which (in 2007) was the fastest known string sorting algorithm.<sup>[7]</sup> However, now there are faster string sorting algorithms.<sup>[8]</sup>

## Full text search

A special kind of trie, called a **suffix tree**, can be used to index all suffixes in a text in order to carry out fast full text searches.

### 8.1.2 Implementation strategies

There are several ways to represent tries, corresponding to different trade-offs between memory use and speed of the operations. The basic form is that of a linked set of nodes, where each node contains an array of child pointers, one for each symbol in the **alphabet** (so for the **English alphabet**, one would store 26 child pointers and for the alphabet of bytes, 256 pointers). This is simple but wasteful in terms of memory if the alphabet is sizeable; the nodes near the bottom of the tree tend to have



A trie implemented as a *doubly-chained tree*: vertical arrows are child pointers, dashed horizontal arrows are next pointers. The set of strings stored in this trie is {baby, bad, bank, box, dad, dance}. The lists are sorted to allow traversal in lexicographic order.

few children and there are many of them.<sup>[9]</sup> An alternative implementation represents a node as a triple (symbol, child, next) and links the children of a node together as a *singly-linked list*: child points to the node's first child, next to the parent node's next child.<sup>[9][10]</sup> The set of children can also be represented as a *binary search tree*, in which case the trie is called a *ternary search tree*.

### Bitwise tries

Bitwise tries are much the same as a normal character based trie except that individual bits are used to traverse what effectively becomes a form of binary tree. Generally, implementations use a special CPU instruction to very quickly find the first set bit in a fixed length key (e.g., GCC's `__builtin_clz()` intrinsic). This value is then used to index a 32- or 64-entry table which points to the first item in the bitwise trie with that number of leading zero bits. The search then proceeds by testing each subsequent bit in the key and choosing `child[0]` or `child[1]` appropriately until the item is found.

Although this process might sound slow, it is very cache-local and highly parallelizable due to the lack of register dependencies and therefore in fact has excellent performance on modern out-of-order execution CPUs. A *red-black tree* for example performs much better on paper, but is highly cache-unfriendly and causes multiple pipeline and *TLB* stalls on modern CPUs which makes that algorithm bound by memory latency rather than CPU speed. In comparison, a bitwise trie rarely accesses memory and when it does it does so only to read, thus avoiding

SMP cache coherency overhead, and hence is becoming increasingly the algorithm of choice for code which does a lot of insertions and deletions such as memory allocators (e.g., recent versions of the famous *Doug Lea's allocator* (`dlmalloc`) and its descendants).

### Compressing tries

When the trie is mostly static, i.e., all insertions or deletions of keys from a prefilled trie are disabled and only lookups are needed, and when the trie nodes are not keyed by node specific data (or if the node's data is common) it is possible to compress the trie representation by merging the common branches.<sup>[11]</sup> This application is typically used for compressing lookup tables when the total set of stored keys is very sparse within their representation space.

For example it may be used to represent sparse *bitsets* (i.e., subsets of a much larger fixed enumerable set) using a trie keyed by the bit element position within the full set, with the key created from the string of bits needed to encode the integral position of each element. The trie will then have a very degenerate form with many missing branches, and compression becomes possible by storing the leaf nodes (set segments with fixed length) and combining them after detecting the repetition of common patterns or by filling the unused gaps.

Such compression is also typically used in the implementation of the various fast lookup tables needed to retrieve *Unicode* character properties (for example to represent case mapping tables, or lookup tables containing the combination of base and combining characters needed to support *Unicode* normalization). For such application, the representation is similar to transforming a very large unidimensional sparse table into a multidimensional matrix, and then using the coordinates in the hyper-matrix as the string key of an uncompressed trie. The compression will then consist of detecting and merging the common columns within the hyper-matrix to compress the last dimension in the key; each dimension of the hypermatrix stores the start position within a storage vector of the next dimension for each coordinate value, and the resulting vector is itself compressible when it is also sparse, so each dimension (associated to a layer level in the trie) is compressed separately.

Some implementations do support such data compression within dynamic sparse tries and allow insertions and deletions in compressed tries, but generally this has a significant cost when compressed segments need to be split or merged, and some tradeoff has to be made between the smallest size of the compressed trie and the speed of updates, by limiting the range of global lookups for comparing the common branches in the sparse trie.

The result of such compression may look similar to trying to transform the trie into a *directed acyclic graph* (DAG), because the reverse transform from a DAG to a trie is

obvious and always possible, however it is constrained by the form of the key chosen to index the nodes.

Another compression approach is to “unravel” the data structure into a single byte array.<sup>[12]</sup> This approach eliminates the need for node pointers which reduces the memory requirements substantially and makes memory mapping possible which allows the virtual memory manager to load the data into memory very efficiently.

Another compression approach is to “pack” the trie.<sup>[2]</sup> Liang describes a space-efficient implementation of a sparse packed trie applied to hyphenation, in which the descendants of each node may be interleaved in memory.

### 8.1.3 See also

- Suffix tree
- Radix tree
- Directed acyclic word graph (aka DAWG)
- Acyclic deterministic finite automata
- Hash trie
- Deterministic finite automata
- Judy array
- Search algorithm
- Extendible hashing
- Hash array mapped trie
- Prefix Hash Tree
- Burstsrt
- Luleå algorithm
- Huffman coding
- Ctrie
- HAT-trie

### 8.1.4 Notes

- [1] Black, Paul E. (2009-11-16). “trie”. *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. Archived from the original on 2010-05-19.
- [2] Franklin Mark Liang (1983). *Word Hy-phen-a-tion By Com-put-er* (Doctor of Philosophy thesis). Stanford University. Archived from the original (PDF) on 2010-05-19. Retrieved 2010-03-28.
- [3] Knuth, Donald (1997). “6.3: Digital Searching”. *The Art of Computer Programming Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley. p. 492. ISBN 0-201-89685-0.

- [4] Bentley, Jon; Sedgewick, Robert (1998-04-01). “Ternary Search Trees”. *Dr. Dobb’s Journal* (Dr Dobb’s). Archived from the original on 2008-06-23.
- [5] Edward Fredkin (1960). “Trie Memory”. *Communications of the ACM* **3** (9): 490–499. doi:10.1145/367390.367400.
- [6] Aho, Alfred V.; Corasick, Margaret J. (Jun 1975). “Efficient String Matching: An Aid to Bibliographic Search” (PDF). *Communications of the ACM* **18** (6): 333–340.
- [7] “Cache-Efficient String Sorting Using Copying” (PDF). Retrieved 2008-11-15.
- [8] “Engineering Radix Sort for Strings.” (PDF). Retrieved 2013-03-11.
- [9] Allison, Lloyd. “Tries”. Retrieved 18 February 2014.
- [10] Sahni, Sartaj. “Tries”. *Data Structures, Algorithms, & Applications in Java*. University of Florida. Retrieved 18 February 2014.
- [11] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, Richard E. Watson (2000). “Incremental Construction of Minimal Acyclic Finite-State Automata”. *Computational Linguistics (Association for Computational Linguistics)* **26**: 3. doi:10.1162/089120100561601. Archived from the original on 2006-03-13. Retrieved 2009-05-28. This paper presents a method for direct building of minimal acyclic finite states automaton which recognizes a given finite list of words in lexicographical order. Our approach is to construct a minimal automaton in a single phase by adding new strings one by one and minimizing the resulting automaton on-the-fly
- [12] Ulrich Germann, Eric Joanis, Samuel Larkin (2009). “Tightly packed tries: how to fit large models into memory, and make them load fast, too” (PDF). *ACL Workshops: Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*. Association for Computational Linguistics. pp. 31–39. We present Tightly Packed Tries (TPTs), a compact implementation of read-only, compressed trie structures with fast on-demand paging and short load times. We demonstrate the benefits of TPTs for storing n-gram back-off language models and phrase tables for statistical machine translation. Encoded as TPTs, these databases require less space than flat text file representations of the same data compressed with the gzip utility. At the same time, they can be mapped into memory quickly and be searched directly in time linear in the length of the key, without the need to decompress the entire file. The overhead for local decompression during search is marginal.

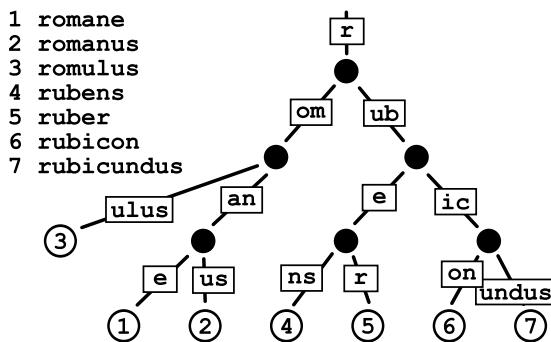
### 8.1.5 References

- de la Briandais, R. (1959). “File Searching Using Variable Length Keys”. *Proceedings of the Western Joint Computer Conference*: 295–298.

### 8.1.6 External links

- NIST's Dictionary of Algorithms and Data Structures: Trie
- Tries by Lloyd Allison
- Comparison and Analysis

## 8.2 Radix tree



An example of a radix tree

In computer science, a **radix tree** (also **patricia trie**, **radix trie** or **compact prefix tree**) is a **data structure** that represents a space-optimized trie in which each node with only one child is merged with its parent. The result is that every internal node has up to the number of children of the **radix**  $r$  of the radix trie, where  $r$  is a positive integer and a power  $x$  of 2, having  $x \geq 1$ . Unlike in regular tries, edges can be labeled with sequences of elements as well as single elements. This makes them much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes.

Unlike regular trees (where whole keys are compared *en masse* from their beginning up to the point of inequality), the key at each node is compared chunk-of-bits by chunk-of-bits, where the quantity of bits in that chunk at that node is the radix  $r$  of the radix trie. When the  $r$  is 2, the radix trie is binary (i.e., compare that node's 1-bit portion of the key), which minimizes sparseness at the expense of maximizing trie depth—i.e., maximizing up to conflation of nondiverging bit-strings in the key. When  $r$  is an integer power of 2 greater or equal to 4, then the radix trie is an  $r$ -ary trie, which lessens the depth of the radix trie at the expense of potential sparseness.

As an optimization, edge labels can be stored in constant size by using two pointers to a string (for the first and last elements).<sup>[1]</sup>

Note that although the examples in this article show strings as sequences of characters, the type of the string elements can be chosen arbitrarily; for example, as a bit or byte of the string representation when using multibyte character encodings or **Unicode**.

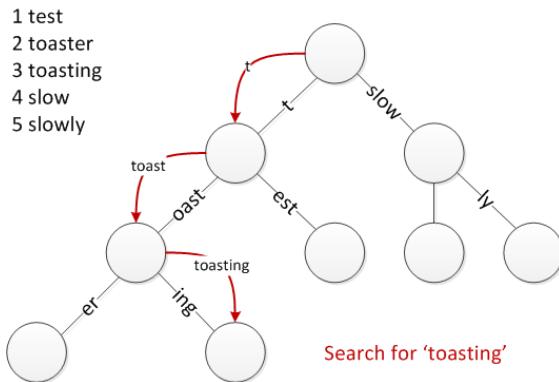
### 8.2.1 Applications

Radix trees are useful for constructing associative arrays with keys that can be expressed as strings. They find particular application in the area of **IP routing**, where the ability to contain large ranges of values with a few exceptions is particularly suited to the hierarchical organization of **IP addresses**.<sup>[2]</sup> They are also used for inverted indexes of text documents in information retrieval.

### 8.2.2 Operations

Radix trees support insertion, deletion, and searching operations. Insertion adds a new string to the trie while trying to minimize the amount of data stored. Deletion removes a string from the trie. Searching operations include (but are not necessarily limited to) exact lookup, find predecessor, find successor, and find all strings with a prefix. All of these operations are  $O(k)$  where  $k$  is the maximum length of all strings in the set, where length is measured in the quantity of bits equal to the radix of the radix trie.

#### Lookup



Finding a string in a Patricia trie

The lookup operation determines if a string exists in a trie. Most operations modify this approach in some way to handle their specific tasks. For instance, the node where a string terminates may be of importance. This operation is similar to tries except that some edges consume multiple elements.

The following pseudo code assumes that these classes exist.

#### Edge

- *Node* targetNode
- *string* label

#### Node

- *Array of Edges* edges

- function isLeaf()

```

function lookup(string x) { // Begin at the root with
 no elements found Node traverseNode := root; int
 elementsFound := 0; // Traverse until a leaf is found
 or it is not possible to continue while (traverseNode != null && !traverseNode.isLeaf() && elementsFound < x.length) { //Get the next edge to explore based on the
 elements not yet found in x Edge nextEdge := select edge
 from traverseNode.edges where edge.label is a prefix
 of x.suffix(elementsFound) //x.suffix(elementsFound) re-
 turns the last (x.length - elementsFound) elements of x //Was an edge found? if (nextEdge != null) { //Set the next
 node to explore traverseNode := nextEdge.targetNode; //Increment elements found based on the label stored at the
 edge elementsFound += nextEdge.label.length; } else {
 //Terminate loop traverseNode := null; } } //A match is
 found if we arrive at a leaf node and have used up ex-
 actly x.length elements return (traverseNode != null &&
 traverseNode.isLeaf() && elementsFound == x.length);
}

```

## Insertion

To insert a string, we search the tree until we can make no further progress. At this point we either add a new outgoing edge labeled with all remaining elements in the input string, or if there is already an outgoing edge sharing a prefix with the remaining input string, we split it into two edges (the first labeled with the common prefix) and proceed. This splitting step ensures that no node has more children than there are possible string elements.

Several cases of insertion are shown below, though more may exist. Note that *r* simply represents the root. It is assumed that edges can be labelled with empty strings to terminate strings where necessary and that the root has no incoming edge. (The lookup algorithm described above will not work when using empty-string edges.)

- Insert 'water' at the root
- Insert 'slower' while keeping 'slow'
- Insert 'test' which is a prefix of 'tester'
- Insert 'team' while splitting 'test' and creating a new edge label 'st'
- Insert 'toast' while splitting 'te' and moving previous strings a level lower

## Deletion

To delete a string *x* from a tree, we first locate the leaf representing *x*. Then, assuming *x* exists, we remove the corresponding leaf node. If the parent of our leaf node has only one other child, then that child's incoming label is appended to the parent's incoming label and the child is removed.

## Additional operations

- Find all strings with common prefix: Returns an array of strings which begin with the same prefix.
- Find predecessor: Locates the largest string less than a given string, by lexicographic order.
- Find successor: Locates the smallest string greater than a given string, by lexicographic order.

### 8.2.3 History

Donald R. Morrison first described what he called “Patricia trees” in 1968;<sup>[3]</sup> the name comes from the acronym **PATRICIA**, which stands for “*Practical Algorithm To Retrieve Information Coded In Alphanumeric*”. Gernot Gwehenberger independently invented and described the data structure at about the same time.<sup>[4]</sup> PATRICIA tries are radix tries with radix equals 2, which means that each bit of the key is compared individually and each node is a two-way (i.e., left versus right) branch.

### 8.2.4 Comparison to other data structures

(In the following comparisons, it is assumed that the keys are of length *k* and the data structure contains *n* members.)

Unlike **balanced trees**, radix trees permit lookup, insertion, and deletion in  $O(k)$  time rather than  $O(\log n)$ . This doesn't seem like an advantage, since normally  $k \geq \log n$ , but in a balanced tree every comparison is a string comparison requiring  $O(k)$  worst-case time, many of which are slow in practice due to long common prefixes (in the case where comparisons begin at the start of the string). In a trie, all comparisons require constant time, but it takes *m* comparisons to look up a string of length *m*. Radix trees can perform these operations with fewer comparisons, and require many fewer nodes.

Radix trees also share the disadvantages of tries, however: as they can only be applied to strings of elements or elements with an efficiently reversible mapping to strings, they lack the full generality of balanced search trees, which apply to any data type with a **total ordering**. A reversible mapping to strings can be used to produce the required total ordering for balanced search trees, but not the other way around. This can also be problematic if a data type only **provides** a comparison operation, but not a (de)serialization operation.

**Hash tables** are commonly said to have expected  $O(1)$  insertion and deletion times, but this is only true when considering computation of the hash of the key to be a constant time operation. When hashing the key is taken into account, hash tables have expected  $O(k)$  insertion and deletion times, but may take longer in the worst-case depending on how collisions are handled. Radix trees

have worst-case  $O(k)$  insertion and deletion. The successor/predecessor operations of radix trees are also not implemented by hash tables.

## 8.2.5 Variants

A common extension of radix trees uses two colors of nodes, 'black' and 'white'. To check if a given string is stored in the tree, the search starts from the top and follows the edges of the input string until no further progress can be made. If the search-string is consumed and the final node is a black node, the search has failed; if it is white, the search has succeeded. This enables us to add a large range of strings with a common prefix to the tree, using white nodes, then remove a small set of "exceptions" in a space-efficient manner by inserting them using black nodes.

The **HAT-trie** is a radix tree based cache-conscious data structure that offers efficient string storage and retrieval, and ordered iterations. Performance, with respect to both time and space, is comparable to the cache-conscious hashtable.<sup>[5][6]</sup> See HAT trie implementation notes at

The **adaptive radix tree** is a radix tree variant that integrates adaptive node sizes to the radix tree. One major drawbacks of the usual radix trees is the use of space, because it uses a constant node size in every level. The major difference between the radix tree and the adaptive radix tree is its variable size for each node based on the number of child-elements, which grows while adding new entries. Hence, the adaptive radix tree leads to a better use of space without reducing its speed.<sup>[7][8][9]</sup>

## 8.2.6 See also

- Prefix tree (also known as a Trie)
- Directed acyclic word graph (aka DAWG)
- Ternary search tries
- Acyclic deterministic finite automata
- Hash trie
- Deterministic finite automata
- Judy array
- Search algorithm
- Extendible hashing
- Hash array mapped trie
- Prefix Hash Tree
- Burstsrt
- Luleå algorithm
- Huffman coding

## 8.2.7 References

- [1] Morin, Patrick. "Data Structures for Strings" (PDF). Retrieved 15 April 2012.
- [2] Knizhnik, Konstantin. "Patricia Tries: A Better Index For Prefix Searches", *Dr. Dobb's Journal*, June, 2008.
- [3] Morrison, Donald R. Practical Algorithm to Retrieve Information Coded in Alphanumeric
- [4] G. Gwehenberger, Anwendung einer binären Verweiskettentmethode beim Aufbau von Listen. *Elektronische Rechenanlagen* 10 (1968), pp. 223–226
- [5] Askitis, Nikolas; Sinha, Ranjan (2007). *HAT-trie: A Cache-conscious Trie-based Data Structure for Strings*. *Proceedings of the 30th Australasian Conference on Computer science* **62**. pp. 97–105. ISBN 1-920682-43-0.
- [6] Askitis, Nikolas; Sinha, Ranjan (October 2010). "Engineering scalable, cache and space efficient tries for strings". *The VLDB Journal* **19** (5): 633–660. doi:10.1007/s00778-010-0183-9. ISSN 1066-8888. ISSN 0949-877X (online).
- [7] Kemper, Alfons; Eickler, André (2013). *Datenbanksysteme, Eine Einführung* **9**. pp. 604–605. ISBN 978-3-486-72139-3.
- [8] "armon/libart · GitHub". *GitHub*. Retrieved 17 September 2014.
- [9] <http://www-db.in.tum.de/~{}leis/papers/ART.pdf>

## 8.2.8 External links

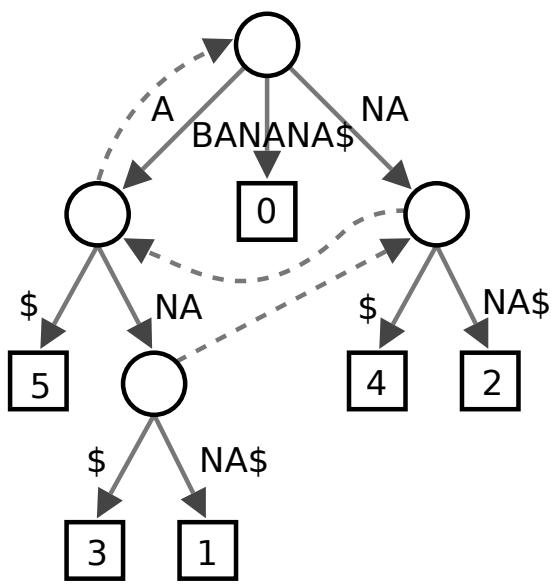
- Algorithms and Data Structures Research & Reference Material: PATRICIA, by Lloyd Allison, Monash University
- Patricia Tree, NIST Dictionary of Algorithms and Data Structures
- Crit-bit trees, by Daniel J. Bernstein
- Radix Tree API in the Linux Kernel, by Jonathan Corbet
- Kart (key alteration radix tree), by Paul Jarc

## Implementations

- Linux Kernel Implementation, used for the page cache, among other things.
- GNU C++ Standard library has a trie implementation
- Java implementation of Concurrent Radix Tree, by Niall Gallagher
- C# implementation of a Radix Tree

- Practical Algorithm Template Library, a C++ library on PATRICIA tries (VC++ >=2003, GCC G++ 3.x), by Roman S. Klyujkov
- Patricia Trie C++ template class implementation, by Radu Gruian
- Haskell standard library implementation “based on big-endian patricia trees”. Web-browsable source code.
- Patricia Trie implementation in Java, by Roger Kapsi and Sam Berlin
- Crit-bit trees forked from C code by Daniel J. Bernstein
- Patricia Trie implementation in C, in libcprops
- Patricia Trees : efficient sets and maps over integers in OCaml, by Jean-Christophe Filliatre

## 8.3 Suffix tree



Suffix tree for the text BANANA. Each substring is terminated with special character \$. The six paths from the root to a leaf (shown as boxes) correspond to the six suffixes A\$, NA\$, ANA\$, NANA\$, ANANA\$ and BANANA\$. The numbers in the leaves give the start position of the corresponding suffix. Suffix links, drawn dashed, are used during construction.

In computer science, a **suffix tree** (also called **PAT tree** or, in an earlier form, **position tree**) is a compressed **trie** containing all the suffixes of the given text as their keys and positions in the text as their values. Suffix trees allow particularly fast implementations of many important string operations.

The construction of such a tree for the string  $S$  takes time and space linear in the length of  $S$ . Once constructed,

several operations can be performed quickly, for instance locating a substring in  $S$ , locating a substring if a certain number of mistakes are allowed, locating matches for a regular expression pattern etc. Suffix trees also provide one of the first linear-time solutions for the **longest common substring problem**. These speedups come at a cost: storing a string's suffix tree typically requires significantly more space than storing the string itself.

### 8.3.1 History

The concept was first introduced by Weiner (1973), which Donald Knuth subsequently characterized as “Algorithm of the Year 1973”. The construction was greatly simplified by McCreight (1976), and also by Ukkonen (1995).<sup>[1]</sup> Ukkonen provided the first online-construction of suffix trees, now known as Ukkonen's algorithm, with running time that matched the then fastest algorithms. These algorithms are all linear-time for a constant-size alphabet, and have worst-case running time of  $O(n \log n)$  in general.

Farach (1997) gave the first suffix tree construction algorithm that is optimal for all alphabets. In particular, this is the first linear-time algorithm for strings drawn from an alphabet of integers in a polynomial range. Farach's algorithm has become the basis for new algorithms for constructing both suffix trees and **suffix arrays**, for example, in external memory, compressed, succinct, etc.

### 8.3.2 Definition

The suffix tree for the string  $S$  of length  $n$  is defined as a tree such that:<sup>[2]</sup>

- The tree has exactly  $n$  leaves numbered from 1 to  $n$ .
- Except for the root, every internal node has at least two children.
- Each edge is labeled with a non-empty substring of  $S$ .
- No two edges starting out of a node can have string-labels beginning with the same character.
- The string obtained by concatenating all the string-labels found on the path from the root to leaf  $i$  spells out suffix  $S[i..n]$ , for  $i$  from 1 to  $n$ .

Since such a tree does not exist for all strings,  $S$  is padded with a terminal symbol not seen in the string (usually denoted \$). This ensures that no suffix is a prefix of another, and that there will be  $n$  leaf nodes, one for each of the  $n$  suffixes of  $S$ . Since all internal non-root nodes are branching, there can be at most  $n - 1$  such nodes, and  $n + (n - 1) + 1 = 2n$  nodes in total ( $n$  leaves,  $n - 1$  internal non-root nodes, 1 root).

**Suffix links** are a key feature for older linear-time construction algorithms, although most newer algorithms, which are based on Farach's algorithm, dispense with suffix links. In a complete suffix tree, all internal non-root nodes have a suffix link to another internal node. If the path from the root to a node spells the string  $\chi\alpha$ , where  $\chi$  is a single character and  $\alpha$  is a string (possibly empty), it has a suffix link to the internal node representing  $\alpha$ . See for example the suffix link from the node for ANA to the node for NA in the figure above. Suffix links are also used in some algorithms running on the tree.

### 8.3.3 Generalized suffix tree

A **generalized suffix tree** is a suffix tree made for a set of words instead of only for a single word. It represents all suffixes from this set of words. Each word must be terminated by a different termination symbol or word.

### 8.3.4 Functionality

A suffix tree for a string  $S$  of length  $n$  can be built in  $\Theta(n)$  time, if the letters come from an alphabet of integers in a polynomial range (in particular, this is true for constant-sized alphabets).<sup>[3]</sup> For larger alphabets, the running time is dominated by first **sorting** the letters to bring them into a range of size  $O(n)$ ; in general, this takes  $O(n \log n)$  time. The costs below are given under the assumption that the alphabet is constant.

Assume that a suffix tree has been built for the string  $S$  of length  $n$ , or that a **generalised suffix tree** has been built for the set of strings  $D = \{S_1, S_2, \dots, S_K\}$  of total length  $n = |n_1| + |n_2| + \dots + |n_K|$ . You can:

- Search for strings:
  - Check if a string  $P$  of length  $m$  is a substring in  $O(m)$  time.<sup>[4]</sup>
  - Find the first occurrence of the patterns  $P_1, \dots, P_q$  of total length  $m$  as substrings in  $O(m)$  time.
  - Find all  $z$  occurrences of the patterns  $P_1, \dots, P_q$  of total length  $m$  as substrings in  $O(m + z)$  time.<sup>[5]</sup>
  - Search for a **regular expression**  $P$  in time expected **sublinear** in  $n$ .<sup>[6]</sup>
  - Find for each suffix of a pattern  $P$ , the length of the longest match between a prefix of  $P[i \dots m]$  and a substring in  $D$  in  $\Theta(m)$  time.<sup>[7]</sup> This is termed the **matching statistics** for  $P$ .
- Find properties of the strings:
  - Find the **longest common substrings** of the string  $S_i$  and  $S_j$  in  $\Theta(n_i + n_j)$  time.<sup>[8]</sup>

- Find all maximal pairs, maximal repeats or supermaximal repeats in  $\Theta(n + z)$  time.<sup>[9]</sup>
- Find the **Lempel-Ziv** decomposition in  $\Theta(n)$  time.<sup>[10]</sup>
- Find the **longest repeated substrings** in  $\Theta(n)$  time.
- Find the most frequently occurring substrings of a minimum length in  $\Theta(n)$  time.
- Find the shortest strings from  $\Sigma$  that do not occur in  $D$ , in  $O(n + z)$  time, if there are  $z$  such strings.
- Find the shortest substrings occurring only once in  $\Theta(n)$  time.
- Find, for each  $i$ , the shortest substrings of  $S_i$  not occurring elsewhere in  $D$  in  $\Theta(n)$  time.

The suffix tree can be prepared for constant time **lowest common ancestor** retrieval between nodes in  $\Theta(n)$  time.<sup>[11]</sup> One can then also:

- Find the longest common prefix between the suffixes  $S_i[p..n_i]$  and  $S_j[q..n_j]$  in  $\Theta(1)$ .<sup>[12]</sup>
- Search for a pattern  $P$  of length  $m$  with at most  $k$  mismatches in  $O(kn + z)$  time, where  $z$  is the number of hits.<sup>[13]</sup>
- Find all  $z$  maximal **palindromes** in  $\Theta(n)$ ,<sup>[14]</sup> or  $\Theta(gn)$  time if gaps of length  $g$  are allowed, or  $\Theta(kn)$  if  $k$  mismatches are allowed.<sup>[15]</sup>
- Find all  $z$  tandem repeats in  $O(n \log n + z)$ , and  $k$ -mismatch tandem repeats in  $O(kn \log(n/k) + z)$ .<sup>[16]</sup>
- Find the **longest common substrings** to at least  $k$  strings in  $D$  for  $k = 2, \dots, K$  in  $\Theta(n)$  time.<sup>[17]</sup>
- Find the **longest palindromic substring** of a given string (using the generalized suffix tree of the string and its reverse) in linear time.<sup>[18]</sup>

### 8.3.5 Applications

Suffix trees can be used to solve a large number of string problems that occur in text-editing, free-text search, computational biology and other application areas.<sup>[19]</sup> Primary applications include:<sup>[19]</sup>

- **String search**, in  $O(m)$  complexity, where  $m$  is the length of the sub-string (but with initial  $O(n)$  time required to build the suffix tree for the string)
- Finding the longest repeated substring
- Finding the longest common substring
- Finding the longest **palindrome** in a string

Suffix trees are often used in **bioinformatics** applications, searching for patterns in **DNA** or **protein** sequences (which can be viewed as long strings of characters). The ability to search efficiently with mismatches might be considered their greatest strength. Suffix trees are also used in **data compression**; they can be used to find repeated data, and can be used for the sorting stage of the **Burrows–Wheeler transform**. Variants of the **LZW** compression schemes use suffix trees (**LZSS**). A suffix tree is also used in **suffix tree clustering**, a **data clustering** algorithm used in some search engines.<sup>[20]</sup>

### 8.3.6 Implementation

If each node and edge can be represented in  $\Theta(1)$  space, the entire tree can be represented in  $\Theta(n)$  space. The total length of all the strings on all of the edges in the tree is  $O(n^2)$ , but each edge can be stored as the position and length of a substring of  $S$ , giving a total space usage of  $\Theta(n)$  computer words. The worst-case space usage of a suffix tree is seen with a **fibonacci word**, giving the full  $2n$  nodes.

An important choice when making a suffix tree implementation is the parent-child relationships between nodes. The most common is using **linked lists** called **sibling lists**. Each node has a pointer to its first child, and to the next node in the child list it is a part of. Other implementations with efficient running time properties use **hash maps**, **sorted** or **unsorted arrays** (with array doubling), or **balanced search trees**. We are interested in:

- The cost of finding the child on a given character.
- The cost of inserting a child.
- The cost of enlisting all children of a node (divided by the number of children in the table below).

Let  $\sigma$  be the size of the alphabet. Then you have the following costs:

Note that the insertion cost is amortised, and that the costs for hashing are given for perfect hashing.

The large amount of information in each edge and node makes the suffix tree very expensive, consuming about 10 to 20 times the memory size of the source text in good implementations. The **suffix array** reduces this requirement to a factor of 8 (for array including **LCP** values built within 32-bit address space and 8-bit characters.) This factor depends on the properties and may reach 2 with usage of 4-byte wide characters (needed to contain any symbol in some UNIX-like systems, see **wchar t**) on 32-bit systems. Researchers have continued to find smaller indexing structures.

### 8.3.7 External construction

Though linear, the memory usage of a suffix tree is significantly higher than the actual size of the sequence collection. For a large text, construction may require external memory approaches.

There are theoretical results for constructing suffix trees in external memory. The algorithm by **Farach-Colton, Ferragina & Muthukrishnan (2000)** is theoretically optimal, with an I/O complexity equal to that of sorting. However the overall intricacy of this algorithm has prevented, so far, its practical implementation.<sup>[21]</sup>

On the other hand, there have been practical works for constructing disk-based suffix trees which scale to (few) GB/hours. The state of the art methods are **TDD**,<sup>[22]</sup> **TRELLIS**,<sup>[23]</sup> **DiGeST**,<sup>[24]</sup> and **B<sup>2</sup>ST**.<sup>[25]</sup>

**TDD** and **TRELLIS** scale up to the entire human genome – approximately 3GB – resulting in a disk-based suffix tree of a size in the tens of gigabytes.<sup>[22][23]</sup> However, these methods cannot handle efficiently collections of sequences exceeding 3GB.<sup>[24]</sup> **DiGeST** performs significantly better and is able to handle collections of sequences in the order of 6GB in about 6 hours.<sup>[24]</sup> All these methods can efficiently build suffix trees for the case when the tree does not fit in main memory, but the input does. The most recent method, **B<sup>2</sup>ST**,<sup>[25]</sup> scales to handle inputs that do not fit in main memory. **ERA** is a recent parallel suffix tree construction method that is significantly faster. **ERA** can index the entire human genome in 19 minutes on an 8-core desktop computer with 16GB RAM. On a simple Linux cluster with 16 nodes (4GB RAM per node), **ERA** can index the entire human genome in less than 9 minutes.<sup>[26]</sup>

### 8.3.8 See also

- **Suffix array**
- **Generalised suffix tree**
- **Trie**

### 8.3.9 Notes

- [1] **Giegerich & Kurtz (1997).**
- [2] [http://www.cs.uoi.gr/~{}kblekas/courses/bioinformatics/Suffix\\_Trees1.pdf](http://www.cs.uoi.gr/~{}kblekas/courses/bioinformatics/Suffix_Trees1.pdf)
- [3] **Farach (1997).**
- [4] **Gusfield (1999), p.92.**
- [5] **Gusfield (1999), p.123.**
- [6] **Baeza-Yates & Gonnet (1996).**
- [7] **Gusfield (1999), p.132.**
- [8] **Gusfield (1999), p.125.**

- [9] Gusfield (1999), p.144.
- [10] Gusfield (1999), p.166.
- [11] Gusfield (1999), Chapter 8.
- [12] Gusfield (1999), p.196.
- [13] Gusfield (1999), p.200.
- [14] Gusfield (1999), p.198.
- [15] Gusfield (1999), p.201.
- [16] Gusfield (1999), p.204.
- [17] Gusfield (1999), p.205.
- [18] Gusfield (1999), pp.197–199.
- [19] Allison, L. “Suffix Trees”. Retrieved 2008-10-14.
- [20] First introduced by Zamir & Etzioni (1998).
- [21] Smyth (2003).
- [22] Tata, Hankins & Patel (2003).
- [23] Phoophakdee & Zaki (2007).
- [24] Barsky et al. (2008).
- [25] Barsky et al. (2009).
- [26] Mansour et al. (2011).
- Giegerich, R.; Kurtz, S. (1997), “From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction” (PDF), *Algorithmica* **19** (3): 331–353, doi:10.1007/PL00009177.
- Gusfield, Dan (1999), *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, ISBN 0-521-58519-8.
- Mansour, Essam; Allam, Amin; Skiadopoulos, Spiros; Kalnis, Panos (2011), “ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings” (PDF), *PVLDB* **5** (1): 49–60.
- McCreight, Edward M. (1976), “A Space-Economical Suffix Tree Construction Algorithm”, *Journal of the ACM* **23** (2): 262–272, doi:10.1145/321941.321946.
- Phoophakdee, Benjarath; Zaki, Mohammed J. (2007), “Genome-scale disk-based suffix tree indexing”, *SIGMOD '07: Proceedings of the ACM SIGMOD International Conference on Management of Data*, New York, NY, USA: ACM, pp. 833–844.
- Smyth, William (2003), *Computing Patterns in Strings*, Addison-Wesley.
- Tata, Sandeep; Hankins, Richard A.; Patel, Jignesh M. (2003), “Practical Suffix Tree Construction”, *VLDB '03: Proceedings of the 30th International Conference on Very Large Data Bases*, Morgan Kaufmann, pp. 36–47.
- Ukkonen, E. (1995), “On-line construction of suffix trees” (PDF), *Algorithmica* **14** (3): 249–260, doi:10.1007/BF01206331.
- Weiner, P. (1973), “Linear pattern matching algorithms” (PDF), *14th Annual IEEE Symposium on Switching and Automata Theory*, pp. 1–11, doi:10.1109/SWAT.1973.13.
- Zamir, Oren; Etzioni, Oren (1998), “Web document clustering: a feasibility demonstration”, *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, New York, NY, USA: ACM, pp. 46–54.

### 8.3.10 References

- Baeza-Yates, Ricardo A.; Gonnet, Gaston H. (1996), “Fast text searching for regular expressions or automaton searching on tries”, *Journal of the ACM* **43** (6): 915–936, doi:10.1145/235809.235810.
- Barsky, Marina; Stege, Ulrike; Thomo, Alex; Upton, Chris (2008), “A new method for indexing genomes using on-disk suffix trees”, *CIKM '08: Proceedings of the 17th ACM Conference on Information and Knowledge Management*, New York, NY, USA: ACM, pp. 649–658.
- Barsky, Marina; Stege, Ulrike; Thomo, Alex; Upton, Chris (2009), “Suffix trees for very large genomic sequences”, *CIKM '09: Proceedings of the 18th ACM Conference on Information and Knowledge Management*, New York, NY, USA: ACM.
- Farach, Martin (1997), “Optimal Suffix Tree Construction with Large Alphabets” (PDF), *38th IEEE Symposium on Foundations of Computer Science (FOCS '97)*, pp. 137–143.
- Farach-Colton, Martin; Ferragina, Paolo; Muthukrishnan, S. (2000), “On the sorting-complexity of suffix tree construction.”, *Journal of the ACM* **47** (6): 987–1011, doi:10.1145/355541.355547.

### 8.3.11 External links

- Suffix Trees by Sartaj Sahni
- Suffix Trees by Lloyd Allison
- NIST's Dictionary of Algorithms and Data Structures: Suffix Tree
- suffix\_tree ANSI C implementation of a Suffix Tree

- libstree, a generic suffix tree library written in C
- Tree::Suffix, a Perl binding to libstree
- Strmat a faster generic suffix tree library written in C (uses arrays instead of linked lists)
- SuffixTree a Python binding to Strmat
- Universal Data Compression Based on the Burrows-Wheeler Transformation: Theory and Practice, application of suffix trees in the BWT
- Theory and Practice of Succinct Data Structures, C++ implementation of a compressed suffix tree
- Practical Algorithm Template Library, a C++ library with suffix tree implementation on PATRICIA trie, by Roman S. Klyujkov
- A Java implementation
- A Java implementation of Concurrent Suffix Tree
- Text-Indexing project (linear-time construction of suffix trees, suffix arrays, LCP array and Burrows-Wheeler Transform)
- Ukkonen's Suffix Tree Implementation in C Part 1 Part 2 Part 3 Part 4 Part 5 Part 6

## 8.4 Suffix array

In computer science, a **suffix array** is a sorted array of all suffixes of a string. It is a data structure used, among others, in full text indices, data compression algorithms and within the field of bioinformatics.<sup>[1]</sup>

Suffix arrays were introduced by Manber & Myers (1990) as a simple, space efficient alternative to suffix trees. They have independently been discovered by Gonnet, Baeza-Yates & Snider (1992) under the name *PAT array*.

### 8.4.1 Definition

Let  $S = S[1]S[2]\dots S[n]$  be a string and let  $S[i, j]$  denote the substring of  $S$  ranging from  $i$  to  $j$ .

The suffix array  $A$  of  $S$  is now defined to be an array of integers providing the starting positions of **suffixes** of  $S$  in lexicographical order. This means, an entry  $A[i]$  contains the starting position of the  $i$ -th smallest suffix in  $S$  and thus for all  $1 < i \leq n : S[A[i-1], n] < S[A[i], n]$ .

### 8.4.2 Example

Consider the text  $S = \text{banana\$}$  to be indexed:

The text ends with the special sentinel letter  $\$$  that is unique and lexicographically smaller than any other character. The text has the following suffixes:

These suffixes can be sorted:

The suffix array  $A$  contains the starting positions of these sorted suffixes:

The suffix array with the suffixes written out vertically underneath for clarity:

So for example,  $A[3]$  contains the value 4, and therefore refers to the suffix starting at position 4 within  $S$ , which is the suffix  $\text{ana\$}$ .

### 8.4.3 Correspondence to suffix trees

Suffix arrays are closely related to **suffix trees**:

- Suffix arrays can be constructed by performing a depth-first traversal of a suffix tree. The suffix array corresponds to the leaf-labels given in the order in which these are visited during the traversal, if edges are visited in the lexicographical order of their first character.
- A suffix tree can be constructed in linear time by using a combination of suffix and **LCP array**. For a description of the algorithm, see the corresponding section in the **LCP array** article.

It has been shown that every suffix tree algorithm can be systematically replaced with an algorithm that uses a suffix array enhanced with additional information (such as the **LCP array**) and solves the same problem in the same time complexity.<sup>[2]</sup> Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality.<sup>[1]</sup>

### 8.4.4 Space Efficiency

Suffix arrays were introduced by Manber & Myers (1990) in order to improve over the space requirements of **suffix trees**: Suffix arrays store  $n$  integers. Assuming an integer requires 4 bytes, a suffix array requires  $4n$  bytes in total. This is significantly less than the  $20n$  bytes which are required by a careful suffix tree implementation.<sup>[3]</sup>

However, in certain applications, the space requirements of suffix arrays may still be prohibitive. Analyzed in bits, a suffix array requires  $\mathcal{O}(n \log n)$  space, whereas the original text over an alphabet of size  $\sigma$  only requires  $\mathcal{O}(n \log \sigma)$  bits. For a human genome with  $\sigma = 4$  and  $n = 3.4 \times 10^9$  the suffix array would therefore occupy about 16 times more memory than the genome itself.

Such discrepancies motivated a trend towards **compressed suffix arrays** and **BWT**-based compressed full-text indices such as the **FM-index**. These data structures require only space within the size of the text or even less.

### 8.4.5 Construction Algorithms

A suffix tree can be built in  $\mathcal{O}(n)$  and can be converted into a suffix array by traversing the tree depth-first also in  $\mathcal{O}(n)$ , so there exist algorithms that can build a suffix array in  $\mathcal{O}(n)$ .

A naive approach to construct a suffix array is to use a comparison-based sorting algorithm. These algorithms require  $\mathcal{O}(n \log n)$  suffix comparisons, but a suffix comparison runs in  $\mathcal{O}(n)$  time, so the overall runtime of this approach is  $\mathcal{O}(n^2 \log n)$ .

More advanced algorithms take advantage of the fact that the suffixes to be sorted are not arbitrary strings but related to each other. These algorithms strive to achieve the following goals:<sup>[4]</sup>

- minimal asymptotic complexity  $\Theta(n)$
- lightweight in space, meaning little or no working memory beside the text and the suffix array itself is needed
- fast in practice

One of the first algorithms to achieve all goals is the SA-IS algorithm of Nong, Zhang & Chan (2009). The algorithm is also rather simple (< 100 LOC) and can be enhanced to simultaneously construct the LCP array.<sup>[5]</sup> The SA-IS algorithm is one of the fastest known suffix array construction algorithms. A careful implementation by Yuta Mori outperforms most other linear or super-linear construction approaches.

Beside time and space requirements, suffix array construction algorithms are also differentiated by their supported **alphabet**: *constant alphabets* where the alphabet size is bound by a constant, *integer alphabets* where characters are integers in a range depending on  $n$  and *general alphabets* where only character comparisons are allowed.<sup>[6]</sup>

Most suffix array construction algorithms are based on one of the following approaches:<sup>[4]</sup>

- *Prefix doubling* algorithms are based on a strategy of Karp, Miller & Rosenberg (1972). The idea is to find prefixes that honor the lexicographic ordering of suffixes. The assessed prefix length doubles in each iteration of the algorithm until a prefix is unique and provides the rank of the associated suffix.
- *Recursive* algorithms follow the approach of the suffix tree construction algorithm by Farach (1997) to recursively sort a subset of suffixes. This subset is then used to infer a suffix array of the remaining suffixes. Both of these suffix arrays are then merged to compute the final suffix array.
- *Induced copying* algorithms are similar to recursive algorithms in the sense that they use an already

sorted subset to induce a fast sort of the remaining suffixes. The difference is that these algorithms favor iteration over recursion to sort the selected suffix subset. A survey of this diverse group of algorithms has been put together by Puglisi, Smyth & Turpin (2007).

A well-known recursive algorithm for integer alphabets is the *DC3 / skew* algorithm of Kärkkäinen & Sanders (2003). It runs in linear time and has successfully been used as the basis for parallel<sup>[7]</sup> and external memory<sup>[8]</sup> suffix array construction algorithms.

Recent work by Salson et al. (2009) proposes an algorithm for updating the suffix array of a text that has been edited instead of rebuilding a new suffix array from scratch. Even if the theoretical worst-case time complexity is  $\mathcal{O}(n \log n)$ , it appears to perform well in practice: experimental results from the authors showed that their implementation of dynamic suffix arrays is generally more efficient than rebuilding when considering the insertion of a reasonable number of letters in the original text.

### 8.4.6 Applications

The suffix array of a string can be used as an **index** to quickly locate every occurrence of a substring pattern  $P$  within the string  $S$ . Finding every occurrence of the pattern is equivalent to finding every suffix that begins with the substring. Thanks to the lexicographical ordering, these suffixes will be grouped together in the suffix array and can be found efficiently with two binary searches. The first search locates the starting position of the interval, and the second one determines the end position:

```
def search(P): l = 0; r = n while l < r: mid = (l+r) / 2 if P > suffixAt(A[mid]): l = mid + 1 else: r = mid s = l; r = n while l < r: mid = (l+r) / 2 if P < suffixAt(A[mid]): r = mid else: l = mid + 1 return (s, r)
```

Finding the substring pattern  $P$  of length  $m$  in the string  $S$  of length  $n$  takes  $\mathcal{O}(m \log n)$  time, given that a single suffix comparison needs to compare  $m$  characters. Manber & Myers (1990) describe how this bound can be improved to  $\mathcal{O}(m + \log n)$  time using LCP information. The idea is that a pattern comparison does not need to re-compare certain characters, when it is already known that these are part of the longest common prefix of the pattern and the current search interval. Abouelhoda, Kurtz & Ohlebusch (2004) improve the bound even further and achieve a search time of  $\mathcal{O}(m)$  as known from suffix trees.

Suffix sorting algorithms can be used to compute the **Burrows–Wheeler transform (BWT)**. The BWT requires sorting of all cyclic permutations of a string. If this string ends in a special end-of-string character that is lexicographically smaller than all other character (i.e., \$), then

the order of the sorted rotated **BWT** matrix corresponds to the order of suffixes in a suffix array. The **BWT** can therefore be computed in linear time by first constructing a suffix array of the text and then deducing the **BWT** string:  $BWT[i] = S[A[i] - 1]$ .

Suffix arrays can also be used to look up substrings in **Example-Based Machine Translation**, demanding much less storage than a full phrase table as used in **Statistical machine translation**.

Many additional applications of the suffix array require the **LCP** array. Some of these are detailed in the application section of the latter.

#### 8.4.7 Notes

- [1] Abouelhoda, Kurtz & Ohlebusch 2002.
- [2] Abouelhoda, Kurtz & Ohlebusch 2004.
- [3] Kurtz 1999.
- [4] Puglisi, Smyth & Turpin 2007.
- [5] Fischer 2011.
- [6] Burkhardt & Kärkkäinen 2003.
- [7] Kulla & Sanders 2007.
- [8] Dementiev et al. 2008.

#### 8.4.8 References

- Abouelhoda, Mohamed Ibrahim; Kurtz, Stefan; Ohlebusch, Enno (2004). “Replacing suffix trees with enhanced suffix arrays”. *Journal of Discrete Algorithms* **2**: 53. doi:10.1016/S1570-8667(03)00065-0.
- Manber, Udi; Myers, Gene (1990). *Suffix arrays: a new method for on-line string searches*. First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327.
- Manber, Udi; Myers, Gene (1993). “Suffix arrays: a new method for on-line string searches”. *SIAM Journal on Computing* **22**: 935–948. doi:10.1137/0222058.
- Gonnet, G.H; Baeza-Yates, R.A; Snider, T (1992). “New indices for text: PAT trees and PAT arrays”. *Information retrieval: data structures and algorithms*.
- Kurtz, S (1999). “Reducing the space requirement of suffix trees”. *Software-Practice and Experience* **29** (13): 1149. doi:10.1002/(SICI)1097-024X(199911)29:13<1149::AID-SPE274>3.0.CO;2-O.
- Abouelhoda, Mohamed Ibrahim; Kurtz, Stefan; Ohlebusch, Enno (2002). *The Enhanced Suffix Array and Its Applications to Genome Analysis*. Algorithms in Bioinformatics. Lecture Notes in Computer Science **2452**. p. 449. doi:10.1007/3-540-45784-4\_35. ISBN 978-3-540-44211-0.
- Puglisi, Simon J.; Smyth, W. F.; Turpin, Andrew H. (2007). “A taxonomy of suffix array construction algorithms”. *ACM Computing Surveys* **39** (2): 4. doi:10.1145/1242471.1242472.
- Nong, Ge; Zhang, Sen; Chan, Wai Hong (2009). *Linear Suffix Array Construction by Almost Pure Induced-Sorting*. 2009 Data Compression Conference. p. 193. doi:10.1109/DCC.2009.42. ISBN 978-0-7695-3592-0.
- Fischer, Johannes (2011). *Inducing the LCP-Array*. Algorithms and Data Structures. Lecture Notes in Computer Science **6844**. p. 374. doi:10.1007/978-3-642-22300-6\_32. ISBN 978-3-642-22299-3.
- Salson, M.; Lecroq, T.; Léonard, M.; Mouchard, L. (2010). “Dynamic extended suffix arrays”. *Journal of Discrete Algorithms* **8** (2): 241. doi:10.1016/j.jda.2009.02.007.
- Burkhardt, Stefan; Kärkkäinen, Juha (2003). *Fast Lightweight Suffix Array Construction and Checking*. Combinatorial Pattern Matching. Lecture Notes in Computer Science **2676**. p. 55. doi:10.1007/3-540-44888-8\_5. ISBN 978-3-540-40311-1.
- Karp, Richard M.; Miller, Raymond E.; Rosenberg, Arnold L. (1972). *Rapid identification of repeated patterns in strings, trees and arrays*. Proceedings of the fourth annual ACM symposium on Theory of computing - STOC '72. p. 125. doi:10.1145/800152.804905.
- Farach, M. (1997). *Optimal suffix tree construction with large alphabets*. Proceedings 38th Annual Symposium on Foundations of Computer Science. p. 137. doi:10.1109/SFCS.1997.646102. ISBN 0-8186-8197-7.
- Kärkkäinen, Juha; Sanders, Peter (2003). *Simple Linear Work Suffix Array Construction*. Automata, Languages and Programming. Lecture Notes in Computer Science **2719**. p. 943. doi:10.1007/3-540-45061-0\_73. ISBN 978-3-540-40493-4.
- Dementiev, Roman; Kärkkäinen, Juha; Mehnert, Jens; Sanders, Peter (2008). “Better external memory suffix array construction”. *Journal of Experimental Algorithms* **12**: 1. doi:10.1145/1227161.1402296.
- Kulla, Fabian; Sanders, Peter (2007). “Scalable parallel suffix array construction”. *Parallel Computing* **33** (9): 605. doi:10.1016/j.parco.2007.06.004.

### 8.4.9 External links

- Suffix Array in Java
- Suffix sorting module for BWT in C code
- Suffix Array Implementation in Ruby
- Suffix array library and tools
- Project containing various Suffix Array c/c++ Implementations with a unified interface
- A fast, lightweight, and robust C API library to construct the suffix array
- Suffix Array implementation in Python
- Linear Time Suffix Array implementation in C using suffix tree

## 8.5 Compressed suffix array

In computer science, a **compressed suffix array**<sup>[1][2][3]</sup> is a **compressed data structure** for **pattern matching**. Compressed suffix arrays are a general class of data structure that improve on the suffix array.<sup>[1][2]</sup> These data structures enable quick search for an arbitrary string with a comparatively small index.

Given a text  $T$  of  $n$  characters from an alphabet  $\Sigma$ , a compressed suffix array supports searching for arbitrary patterns in  $T$ . For an input pattern  $P$  of  $m$  characters, the search time is typically  $O(m)$  or  $O(m + \log(n))$ . The space used is typically  $O(nH_k(T)) + o(n)$ , where  $H_k(T)$  is the  $k$ -th order empirical entropy of the text  $T$ . The time and space to construct a compressed suffix array are normally  $O(n)$ .

The original instantiation of the compressed suffix array<sup>[1]</sup> solved a long-standing open problem by showing that fast pattern matching was possible using only a linear-space data structure, namely, one proportional to the size of the text  $T$ , which takes  $O(n \log |\Sigma|)$  bits. The conventional suffix array and suffix tree use  $\Omega(n \log n)$  bits, which is substantially larger. The basis for the data structure is a recursive decomposition using the “neighbor function,” which allows a suffix array to be represented by one of half its length. The construction is repeated multiple times until the resulting suffix array uses a linear number of bits. Following work showed that the actual storage space was related to the zeroth-order entropy and that the index supports self-indexing.<sup>[4]</sup> The space bound was further improved achieving the ultimate goal of higher-order entropy; the compression is obtained by partitioning the neighbor function by high-order contexts, and compressing each partition with a **wavelet tree**.<sup>[3]</sup> The space usage is extremely competitive in practice with other state-of-the-art compressors,<sup>[5]</sup> and it also supports fast pattern matching.

The memory accesses made by compressed suffix arrays and other compressed data structures for pattern matching are typically not localized, and thus these data structures have been notoriously hard to design efficiently for use in **external memory**. Recent progress using geometric duality takes advantage of the block access provided by disks to speed up the I/O time significantly<sup>[6]</sup> In addition, potentially practical search performance for a compressed suffix array in external-memory has been demonstrated.<sup>[7]</sup>

### 8.5.1 Open Source Implementations

There are several open source implementations of compressed suffix arrays available (see **External Links** below). Bowtie and Bowtie2 are open-source compressed suffix array implementations of **read alignment** for use in **bioinformatics**. The Succinct Data Structure Library (SDSL) is a library containing a variety of compressed data structures including compressed suffix arrays. FEMTO is an implementation of compressed suffix arrays for external memory. In addition, a variety of implementations, including the original **FM-index** implementations, are available from the Pizza & Chili Website.

### 8.5.2 See also

FM-index

Suffix Array

### 8.5.3 References

- [1] R. Grossi and J. S. Vitter, Compressed Suffix Arrays and Suffix Trees, with Applications to Text Indexing and String Matching, *SIAM Journal on Computing*, 35(2), 2005, 378-407. An earlier version appeared in *Proceedings of the 32nd ACM Symposium on Theory of Computing, May 2000*, 397-406.
- [2] Paolo Ferragina and Giovanni Manzini (2000). “Opportunistic Data Structures with Applications”. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. p.390.
- [3] R. Grossi, A. Gupta, and J. S. Vitter, High-Order Entropy-Compressed Text Indexes, *Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms*, January 2003, 841-850.
- [4] K. Sadakane, Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Arrays, *Proceedings of the International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science, vol. 1969, Springer, December 2000, 410-421.
- [5] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, Indexing Equals Compression: Experiments on Suffix Arrays and Trees, *ACM Transactions on Algorithms*, 2(4), 2006, 611-639.

- [6] W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter, On Entropy-Compressed Text Indexing in External Memory, *Proceedings of the Conference on String Processing and Information Retrieval*, August 2009.
- [7] M. P. Ferguson, *FEMTO: fast search of large sequence collections*, *Proceedings of the 23rd Annual Conference on Combinatorial Pattern Matching*, July 2012

### 8.5.4 External links

Implementations:

- Bowtie and Bowtie2
- Succinct Data Structure Library (SDSL)
- FEMTO
- Pizza&Chili website.

## 8.6 FM-index

In computer science, an **FM-index** is a compressed full-text substring index based on the **Burrows-Wheeler transform**, with some similarities to the **suffix array**. It was created by Paolo Ferragina and Giovanni Manzini,<sup>[1]</sup> who describe it as an opportunistic data structure as it allows compression of the input text while still permitting fast substring queries. The name stands for Full-text index in Minute space.<sup>[2]</sup>

It can be used to efficiently find the number of occurrences of a pattern within the compressed text, as well as locate the position of each occurrence. Both the query time and storage space requirements are sublinear with respect to the size of the input data.

The original authors have devised improvements to their original approach and dubbed it “FM-Index version 2”.<sup>[3]</sup> A further improvement, the alphabet-friendly FM-index, combines the use of compression boosting and **wavelet trees**<sup>[4]</sup> to significantly reduce the space usage for large alphabets.

The FM-index has found use in, among other places, **bioinformatics**.<sup>[5]</sup>

### 8.6.1 Background

Using an index is a common strategy to efficiently search a large body of text. When the text is larger than what reasonably fits within a computer’s main memory, there is a need to compress not only the text but also the index. When the FM-index was introduced, there were several suggested solutions that were based on traditional compression methods and tried to solve the compressed matching problem. In contrast, the FM-index is a compressed self-index, which means that it compresses the data and indexes it at the same time.

### 8.6.2 FM-index data structure

An FM-index is created by first taking the **Burrows-Wheeler transform** (BWT) of the input text. For example, the BWT of the string  $T = \text{"abracadabra"}$  is “ard\$rcaaaabb”, and here it is represented by the matrix  $M$  where each row is a rotation of the text that has been sorted. The transform corresponds to the last column labeled  $L$ .

The BWT in itself allows for some compression with, for instance, **move to front** and **Huffman encoding**, but the transform has even more uses. The rows in the matrix are essentially the sorted suffixes of the text and the first column  $F$  of the matrix shares similarities with **suffix arrays**. How the suffix array relates to the BWT lies at the heart of the FM-index.

### 8.6.3 Count

The operation *count* takes a pattern  $P[1..p]$  and returns the number of occurrences of that pattern in the original text  $T$ . Since the rows of matrix  $M$  are sorted, and it contains every suffix of  $T$ , the occurrences of pattern  $P$  will be next to each other in a single continuous range. The operation iterates backwards over the pattern. For every character in the pattern, the range that has the character as a suffix is found. For example, the count of the pattern “bra” in “abracadabra” follows these steps:

1. The first character we look for is a, the last character in the pattern. The initial range is set to  $[C[a] + 1..C[a+1] = [2..6]$ . This range over  $L$  represents every character of  $T$  that has a suffix beginning with  $a$ .
2. The next character to look for is r. The new range is  $[C[r] + \text{Occ}(r, \text{start}-1) + 1..C[r] + \text{Occ}(r, \text{end})] = [10 + 0 + 1..10 + 2] = [11..12]$ , if start is the index of the beginning of the range and end is the end. This range over  $L$  is all the characters of  $T$  that have suffixes beginning with  $ra$ .
3. The last character to look at is b. The new range is  $[C[b] + \text{Occ}(b, \text{start}-1) + 1..C[b] + \text{Occ}(b, \text{end})] = [6 + 0 + 1..6 + 2] = [7..8]$ . This range over  $L$  is all the characters that have a suffix that begins with  $bra$ . Now that the whole pattern has been processed, the count is the same as the size of the range:  $8 - 7 + 1 = 2$ .

If the range at becomes empty or the range boundaries cross each other before the whole pattern has been looked up, the pattern does not occur in  $T$ . Because  $\text{Occ}(c, k)$  can be performed in constant time, *count* can complete in linear time in the length of the pattern:  $O(p)$  time.

### 8.6.4 Locate

The operation *locate* takes as input an index of a character in  $L$  and returns its position  $i$  in  $T$ . For instance  $\text{locate}(7) = 8$ . To locate every occurrence of a pattern, first the range of character is found whose suffix is the pattern in the same way the *count* operation found the range. Then the position of every character in the range can be located.

To map an index in  $L$  to one in  $T$ , a subset of the indices in  $L$  are associated with a position in  $T$ . If  $L[j]$  has a position associated with it,  $\text{locate}(j)$  is trivial. If it's not associated, the string is followed with  $\text{LF}(i)$  until an associated index is found. By associating a suitable number of indices, an upper bound can be found. *Locate* can be implemented to find  $occ$  occurrences of a pattern  $P[1..p]$  in a text  $T[1..u]$  in  $O(p + occ \log^e u)$  time with  $O(H_k(T) + \frac{\log \log u}{\log^e u})$  bits per input symbol for any  $k \geq 0$ .<sup>[1]</sup>

### 8.6.5 Applications

#### DNA read mapping

FM index with Backtracking has been successfully (>2000 citations) applied to approximate string matching/sequence alignment, See Bowtie <http://bowtie-bio.sourceforge.net/index.shtml>

### 8.6.6 See also

Burrows–Wheeler transform

Suffix array

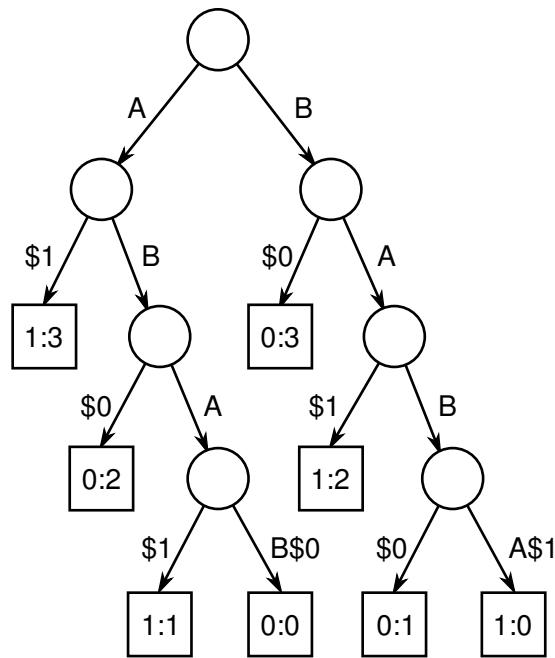
Compressed suffix array

Sequence alignment

### 8.6.7 References

- [1] Paolo Ferragina and Giovanni Manzini (2000). “Opportunistic Data Structures with Applications”. Proceedings of the 41st Annual Symposium on Foundations of Computer Science. p.390.
- [2] Paolo Ferragina and Giovanni Manzini (2005). “Indexing Compressed Text”. Journal of the ACM (JACM), 52, 4 (Jul. 2005). p. 553
- [3] Paolo Ferragina and Rossano Venturini “FM-Index version 2”
- [4] P. Ferragina, G. Manzini, V. Mäkinen and G. Navarro. An Alphabet-Friendly FM-index. In Proc. SPIRE'04, pages 150-160. LNCS 3246.
- [5] Simpson, Jared T. and Durbin, Richard (2010). “Efficient construction of an assembly string graph using the FM-index”. Bioinformatics, 26, 12 (Jun. 17). p. i367

## 8.7 Generalized suffix tree



suffix tree for the strings ABAB and BABA. Suffix links not shown.

In computer science, a **generalized suffix tree** is a suffix tree for a set of strings. Given the set of strings  $D = S_1, S_2, \dots, S_d$  of total length  $n$ , it is a **Patricia tree** containing all  $n$  suffixes of the strings. It is mostly used in bioinformatics.

### 8.7.1 Functionality

It can be built in  $\Theta(n)$  time and space, and can be used to find all  $z$  occurrences of a string  $P$  of length  $m$  in  $O(m + z)$  time, which is asymptotically optimal (assuming the size of the alphabet is constant, see page 119).

When constructing such a tree, each string should be padded with a unique out-of-alphabet marker symbol (or string) to ensure no suffix is a substring of another, guaranteeing each suffix is represented by a unique leaf node.

Algorithms for constructing a GST include Ukkonen's algorithm (1995) and McCreight's algorithm (1976).

### 8.7.2 Example

A suffix tree for the strings ABAB and BABA is shown in a figure above. They are padded with the unique terminator strings  $\$0$  and  $\$1$ . The numbers in the leaf nodes are string number and starting position. Notice how a left to right traversal of the leaf nodes corresponds to the sorted order of the suffixes. The terminators might be strings or unique single symbols. Edges on  $\$$  from the root are left out in this example.

### 8.7.3 Alternatives

An alternative to building a generalised suffix tree is to concatenate the strings, and build a regular suffix tree or **suffix array** for the resulting string. When hits are evaluated after a search, global positions are mapped into documents and local positions with some algorithm and/or data structure, such as a binary search in the starting/ending positions of the documents.

### 8.7.4 References

- ^ Lucas Chi Kwong Hui (1992). “Color Set Size Problem with Applications to String Matching”. *Combinatorial Pattern Matching, Lecture Notes in Computer Science*, 644. pp. 230–243.
- ^ Paul Bieganski, John Riedl, John Carlis, and Ernest F. Retzel (1994). “Generalized Suffix Trees for Biological Sequence Data”. *Biotechnology Computing, Proceedings of the Twenty-Seventh Hawaii International Conference on*. pp. 35–44.
- ^ Gusfield, Dan (1999) [1997]. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. USA: Cambridge University Press. ISBN 0-521-58519-8.

### 8.7.5 External links

- A C implementation of Generalized Suffix Tree for two strings

## 8.8 B-trie

The **B-trie** is a **trie-based data structure** that can store and retrieve variable-length strings efficiently on disk.<sup>[1]</sup>

The B-trie was compared against several high-performance variants of **B-tree** that were designed for string keys. It was shown to offer superior performance, particularly under skew access (i.e., many repeated searches). It is currently a leading choice for maintaining a string dictionary on disk, along with other disk-based tasks, such as maintaining an index to a string database or for accumulating the vocabulary of a large text collection.

### 8.8.1 References

- [1] Askitis, Nikolas; Zobel, Justin (2008), “B-tries for Disk-based String Management”, *VLDB Journal*: 1–26, ISSN 1066-8888

## 8.9 Judy array

In computer science and software engineering, a **Judy array** is a data structure that has high performance, low memory usage and implements an **associative array**. Unlike normal arrays, Judy arrays may be sparse, that is, they may have large ranges of unassigned indices. They can be used for storing and looking up values using integer or string keys. The key benefits of using a Judy array is its scalability, high performance, memory efficiency and ease of use.<sup>[1]</sup>

Judy arrays are both speed- and memory-efficient, with no tuning or configuration required and therefore they can sometime replace common in-memory dictionary implementations (like red-black trees or hash tables) and work better with very large data sets.

Roughly speaking, Judy arrays are highly optimised 256-ary radix trees.<sup>[2]</sup> Judy arrays use over 20 different compression techniques on **trie** nodes to reduce memory usage.

The Judy array was invented by Douglas Baskins and named after his sister.<sup>[3]</sup>

### 8.9.1 Terminology

1. *Expanse* is a range of possible keys, e.g., 200...300, etc.<sup>[4]</sup>
2. *Population* is the expanse’s total number of keys, e.g., a population of 5 could be the keys 200, 360, 400, 512, and 720<sup>[4]</sup>
3. *Density* equals Population/Expanse and is an estimate of sparseness<sup>[4]</sup>

### 8.9.2 Benefits

#### Memory allocation

Judy arrays are **dynamic** and can grow or shrink as elements are added to, or removed from, the array. The maximum size of a Judy array is bounded by machine memory.<sup>[5]</sup> The memory used by Judy arrays is nearly proportional to the number of elements (population) in the Judy array.

#### Speed

Judy arrays are designed to keep the number of processor **cache-line** fills as low as possible, and the algorithm is internally complex in an attempt to satisfy this goal as often as possible. Due to these **cache** optimizations, Judy arrays are fast, sometimes even faster than a **hash table**, especially for very big datasets. Despite Judy arrays being a type of **trie**, they consume much less memory than hash

tables. Also because a Judy array is a trie, it is possible to do an ordered sequential traversal of keys, which is not possible in hash tables.

### 8.9.3 Drawbacks

The HP (SourceForge) implementation of Judy arrays appears to be the subject of US patent 6735595.<sup>[6]</sup>

### 8.9.4 References

- [1] <http://packages.debian.org/wheezy/libjudy-dev>
- [2] Alan Silverstein, "Judy IV Shop Manual", 2002
- [3] <http://judy.sourceforge.net/>
- [4] Baskins, Doug (October 16, 2001). "A 10-MINUTE DESCRIPTION OF HOW JUDY ARRAYS WORK AND WHY THEY ARE SO FAST".
- [5] Advances in databases: concepts, systems and applications : By Kotagiri Ramamohanarao
- [6] <http://www.google.com/patents/US6735595>

### 8.9.5 External links

- Main Judy arrays site
- How Judy arrays work and why they are so fast
- A complete technical description of Judy arrays
- An independent performance comparison of Judy to Hash Tables
- A compact implementation of Judy arrays in 1250 lines of C code

## 8.10 Ctrie

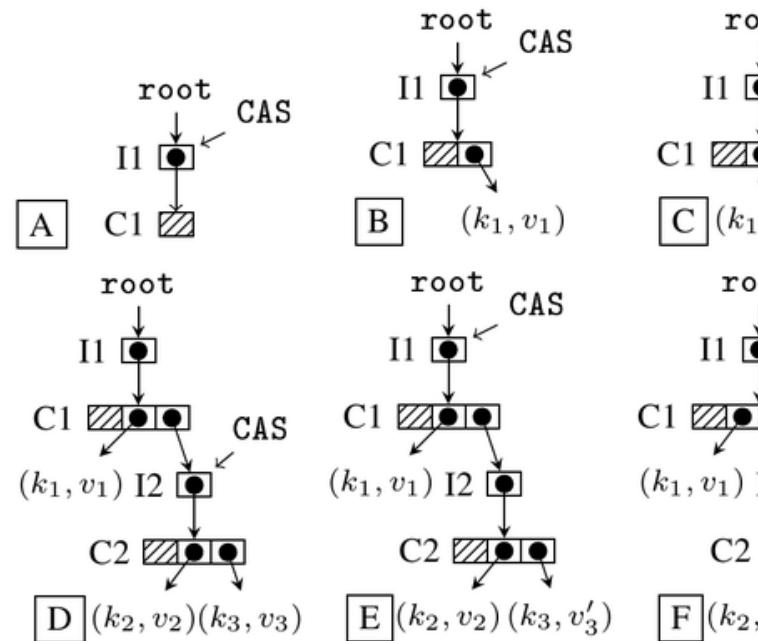
Not to be confused with C-trie.

A **concurrent hash-trie** or **Ctrie**<sup>[1][2]</sup> is a concurrent thread-safe lock-free implementation of a hash array mapped trie. It is used to implement the concurrent map abstraction. It has particularly scalable concurrent insert and remove operations and is memory-efficient.<sup>[3]</sup> It is the first known concurrent data-structure that supports O(1), atomic, lock-free snapshots.<sup>[2][4]</sup>

### 8.10.1 Operation

The Ctrie data structure is a non-blocking concurrent hash array mapped trie based on single-word compare-and-swap instructions in a shared-memory system. It supports concurrent lookup, insert and remove operations. Just like the hash array mapped trie, it uses the entire 32-bit space for hash values thus having low risk of hashcode collisions. Each node may branch to up to 32 sub tries. To conserve memory, each node contains a 32 bits bitmap where each bit indicates the presence of a branch followed by an array of length equal to the Hamming weight of the bitmap.

Keys are inserted by doing an atomic compare-and-swap operation on the node which needs to be modified. To ensure that updates are done independently and in a proper order, a special indirection node (an I-node) is inserted between each regular node and its subtrees.



The figure above illustrates the Ctrie insert operation. Trie A is empty - an atomic CAS instruction is used to swap the old node C1 with the new version of C1 which has the new key  $k1$ . If the CAS is not successful, the operation is restarted. If the CAS is successful, we obtain the trie B. This procedure is repeated when a new key  $k2$  is added (trie C). If two hashcodes of the keys in the Ctrie collide as is the case with  $k2$  and  $k3$ , the Ctrie must be extended with at least one more level - trie D has a new indirection node I2 with a new node C2 which holds both colliding keys. Further CAS instructions are done on the contents of the indirection nodes I1 and I2 - such CAS instructions can be done independently of each other, thus enabling concurrent updates with less contention.

The Ctrie is defined by the pointer to the root indirection

node (or a root I-node). The following types of nodes are defined for the Ctrie:

```
structure INode { main: CNode } structure CNode {
 bmp: integer array: Branch[2^W] } Branch: INode |
 SNode structure SNode { k: KeyType v: ValueType }
```

A C-node is a branching node. It typically contains up to 32 branches, so  $W$  above is 5. Each branch may either be a key-value pair (represented with an S-node) or another I-node. To avoid wasting 32 entries in the branching array when some branches may be empty, an integer bitmap is used to denote which bits are full and which are empty. The helper method *flagpos* is used to inspect the relevant hashcode bits for a given level and extract the value of the bit in the bitmap to see if its set or not - denoting whether there is a branch at that position or not. If there is a bit, it also computes its position in the branch array. The formula used to do this is:

```
bit = bmp & (1 << ((hashcode >> level) & 0x1F)) pos =
bitcount((bit - 1) & bmp)
```

Note that the operations treat only the I-nodes as mutable nodes - all other nodes are never changed after being created and added to the Ctrie.

Below is an illustration of the pseudocode of the insert operation:

```
def insert(k, v) r = READ(root) if iinsert(r, k, v, 0, null) =
RESTART insert(k, v) def iinsert(i, k, v, lev, parent) cn =
READ(i.main) flag, pos = flagpos(k.hc, lev, cn.bmp)
if cn.bmp & flag = 0 { ncn = cn.inserted(pos, flag,
SNode(k, v)) if CAS(i.main, cn, ncn) return OK else re-
turn RESTART } cn.array(pos) match { case sin: INode =>
{ return iinsert(sin, k, v, lev + W, i) case sn: SNode =>
if sn.k ≠ k { nsn = SNode(k, v) nin = INode(CNode(sn,
nsn, lev + W)) ncn = cn.updated(pos, nin) if CAS(i.main,
cn, ncn) return OK else return RESTART } else { ncn =
cn.updated(pos, SNode(k, v)) if CAS(i.main, cn, ncn) re-
turn OK else return RESTART } }
```

The *inserted* and *updated* methods on nodes return new versions of the C-node with a value inserted or updated at the specified position, respectively. Note that the insert operation above is tail-recursive, so it can be rewritten as a while loop. Other operations are described in more detail in the original paper on Ctries.<sup>[1][5]</sup>

The data-structure has been proven to be correct<sup>[1]</sup> - Ctrie operations have been shown to have the atomicity, linearizability and lock-freedom properties. The lookup operation can be modified to guarantee wait-freedom.

### 8.10.2 Advantages of Ctries

Ctries have been shown to be comparable in performance with concurrent skip lists,<sup>[2][4]</sup> concurrent hash tables and similar data structures in terms of the lookup operation, being slightly slower than hash tables and faster than skip lists due to the lower level of indirections. However, they

are far more scalable than most concurrent hash tables where the insertions are concerned.<sup>[1]</sup> Most concurrent hash tables are bad at conserving memory - when the keys are removed from the hash table, the underlying array is not shrunk. Ctries have the property that the allocated memory is always a function of only the current number of keys in the data-structure.<sup>[1]</sup>

Ctries have logarithmic complexity bounds of the basic operations, albeit with a low constant factor due to the high branching level (usually 32).

Ctries support a lock-free, linearizable, constant-time snapshot operation,<sup>[2]</sup> based on the insight obtained from persistent data structures. This is a breakthrough in concurrent data-structure design, since existing concurrent data-structures do not support snapshots. The snapshot operation allows implementing lock-free, linearizable iterator, size and clear operations - existing concurrent data-structures have implementations which either use global locks or are correct only given that there are no concurrent modifications to the data-structure. In particular, Ctries have an O(1) iterator creation operation, O(1) clear operation, O(1) duplicate operation and an amortized O(logn) size retrieval operation.

### 8.10.3 Problems with Ctries

Most concurrent data structures require dynamic memory allocation, and lock-free concurrent data structures rely on garbage collection on most platforms. The current implementation<sup>[4]</sup> of the Ctrie is written for the JVM, where garbage collection is provided by the platform itself. While it's possible to keep a concurrent memory pool for the nodes shared by all instances of Ctries in an application or use reference counting to properly deallocate nodes, the only implementation so far to deal with manual memory management of nodes used in Ctries is the common-lisp implementation cl-ctrie, which implements several stop-and-copy and mark-and-sweep garbage collection techniques for persistent, memory-mapped storage. Hazard pointers are another possible solution for a correct manual management of removed nodes. Such a technique may be viable for managed environments as well, since it could lower the pressure on the GC. A Ctrie implementation in Rust makes use of hazard pointers for this purpose.<sup>[6]</sup>

### 8.10.4 Implementations

A Ctrie implementation<sup>[4]</sup> for Scala 2.9.x is available on GitHub. It is a mutable thread-safe implementation which ensures progress and supports lock-free, linearizable, O(1) snapshots.

- A data-structure similar to Ctries has been used in ScalaSTM,<sup>[7]</sup> a software transactional memory library for the JVM.

- The Scala standard library includes a Ctries implementation since February 2012.<sup>[8]</sup>
  - Haskell implementation is available as a package<sup>[9]</sup> and on GitHub.<sup>[10]</sup>
  - A standalone Java implementation is available on GitHub.<sup>[11]</sup>
  - CL-CTRIE is the Common Lisp implementation is available on GitHub.<sup>[12]</sup>
  - An insert-only Ctrie variant has been used for tabling in Prolog programs.<sup>[13]</sup>
  - A Rust implementation<sup>[6]</sup> uses hazard pointers in its implementation to achieve lock-free synchronization.
- [10] GitHub repo for Haskell Ctrie  
[11] GitHub repo for Java Ctrie  
[12] GitHub repo for Common Lisp Ctrie  
[13] Miguel Areias and Ricardo Rocha, *A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs*

### 8.10.5 History

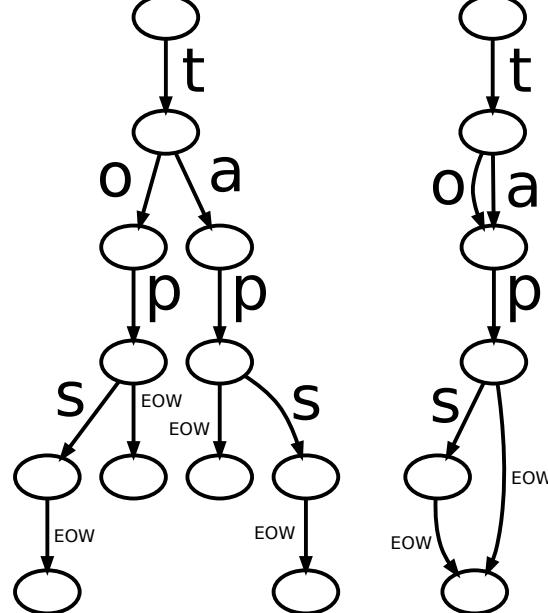
Ctries were first described in 2011 by Aleksandar Prokopec.<sup>[1]</sup> To quote the author:

*Ctrie is a non-blocking concurrent shared-memory hash trie based on single-word compare-and-swap instructions. Insert, lookup and remove operations modifying different parts of the hash trie can be run independent of each other and do not contend. Remove operations ensure that the unneeded memory is freed and that the trie is kept compact.*

In 2012, a revised version of the Ctrie data structure was published,<sup>[2]</sup> simplifying the data structure and introducing an optional constant-time, lock-free, atomic snapshot operation.

### 8.10.6 References

- [1] Prokopec, A. et al. (2011) Cache-Aware Lock-Free Concurrent Hash Tries. Technical Report, 2011.
- [2] Prokopec, A., Bronson N., Bagwell P., Odersky M. (2011) Concurrent Tries with Efficient Non-Blocking Snapshots
- [3] Prokopec, A. et al. (2011) Lock-Free Resizable Concurrent Tries. The 24th International Workshop on Languages and Compilers for Parallel Computing, 2011.
- [4] Prokopec, A. JVM implementation on GitHub
- [5] <http://axel22.github.io/resources/docs/lcpc-ctries.ppt>
- [6] Rust Ctrie implementation at GitHub
- [7] N. Bronson ScalaSTM
- [8] TrieMap.scala
- [9] Haskell ctrie package



The strings “tap”, “taps”, “top”, and “tops” stored in a Trie (left) and a DAFSA (right), EOW stands for End-of-word.

In computer science, a **deterministic acyclic finite state automaton (DAFSA)**,<sup>[1]</sup> also called a **directed acyclic word graph (DAWG)**; though that name also refers to a related data structure that functions as a suffix index<sup>[2]</sup>) is a data structure that represents a set of **strings**, and allows for a query operation that tests whether a given string belongs to the set in time proportional to its length. In these respects, a DAFSA is very similar to a **trie**, but it is much more space efficient.

A DAFSA is a special case of a **finite state recognizer** that takes the form of a **directed acyclic graph** with a single source vertex (a vertex with no incoming edges), in which each edge of the graph is labeled by a letter or symbol, and in which each vertex has at most one outgoing edge for each possible letter or symbol. The strings represented by the DAFSA are formed by the symbols on paths in the graph from the source vertex to any sink vertex (a vertex with no outgoing edges). In fact, a **deterministic finite state automaton** is **acyclic** if and only if it recognizes a finite set of strings.<sup>[1]</sup>

### 8.11.1 Comparison to tries

By allowing the same vertices to be reached by multiple paths, a DAFSA may use significantly fewer vertices than the strongly related trie data structure. Consider, for example, the four English words “tap”, “taps”, “top”, and “tops”. A trie for those four words would have 11 vertices, one for each of the strings formed as a prefix of one of these words, or for one of the words followed by the end-of-string marker. However, a DAFSA can represent these same four words using only six vertices  $v_i$  for  $0 \leq i \leq 5$ , and the following edges: an edge from  $v_0$  to  $v_1$  labeled “t”, two edges from  $v_1$  to  $v_2$  labeled “a” and “o”, an edge from  $v_2$  to  $v_3$  labeled “p”, an edge  $v_3$  to  $v_4$  labeled “s”, and edges from  $v_3$  and  $v_4$  to  $v_5$  labeled with the end-of-string marker. There is a tradeoff between memory and functionality, because a standard DAFSA can tell you if a word exists within it, but it cannot point you to auxiliary information about that word, whereas a trie can.

The primary difference between DAFSA and trie is the elimination of suffix and infix redundancy in storing strings. The trie eliminates prefix redundancy since all common prefixes are shared between strings, such as between *doctors* and *doctorate* the *doctor* prefix is shared. In a DAFSA common suffixes are also shared, for words that have the same set of possible suffixes as each other. For dictionary sets of common English words, this translates into major memory usage reduction.

Because the terminal nodes of a DAFSA can be reached by multiple paths, a DAFSA cannot directly store auxiliary information relating to each path, e.g. a word’s frequency in the English language. However, if for each node we store the number of unique paths through that point in the structure, we can use it to retrieve the index of a word, or a word given its index.<sup>[3]</sup> The auxiliary information can then be stored in an array.

### 8.11.2 References

- [1] Jan Daciuk, Stoyan Mihov, Bruce Watson and Richard Watson (2000). Incremental construction of minimal acyclic finite state automata. *Computational Linguistics* **26**(1):3-16.
- [2] Black, Paul E. “directed acyclic word graph”. *Dictionary of Algorithms and Data Structures*. NIST.
- [3] Kowaltowski, T.; CL Lucchesi (1993). “Applications of finite automata representing large vocabularies”. *Software-Practice and Experience* **1993**: 15–30. CiteSeerX: 10.1.1.56.5272.
- Blumer, A.; Blumer, J.; Haussler, D.; Ehrenfeucht, A.; Chen, M.T.; Seiferas, J. (1985), “The smallest automation recognizing the subwords of a text”, *Theoretical computer science* **40**: 31–55, doi:10.1016/0304-3975(85)90157-4

- Appel, Andrew; Jacobsen, Guy (1988), “The World’s Fastest Scrabble Program” (PDF), *Communications of the ACM*. One of the early mentions of the data structure.

- Jansen, Cees J. A.; Boekee, Dick E. (1990), “On the significance of the directed acyclic word graph in cryptology”, *Advances in Cryptology — AUSCRYPT '90*, Lecture Notes in Computer Science **453**, Springer-Verlag, pp. 318–326, doi:10.1007/BFb0030372, ISBN 3-540-53000-2.

- Epifanio, Chiara; Mignosi, Filippo; Shallit, Jeffrey; Venturini, Ilaria (2004), “Sturmian graphs and a conjecture of Moser”, in Calude, Cristian S.; Calude, Elena; Dineen, Michael J., *Developments in language theory. Proceedings, 8th international conference (DLT 2004)*, Auckland, New Zealand, December 2004, Lecture Notes in Computer Science **3340**, Springer-Verlag, pp. 175–187, ISBN 3-540-24014-4, Zbl 1117.68454

### 8.11.3 External links

- <http://pages.pathcom.com/~{}vadco/dawg.html> - JohnPaul Adamovsky teaches how to construct a DAFSA using an array of integers.
- <http://pages.pathcom.com/~{}vadco/cwg.html> - JohnPaul Adamovsky teaches how to construct a DAFSA hash function using a novel encoding with multiple integer arrays. This encoding is called the Caroline Word Graph (CWG).

# Chapter 9

## Multiway trees

### 9.1 Ternary search tree

In computer science, a **ternary search tree** is a type of tree (sometimes called a *prefix tree*) where nodes are arranged in a manner similar to a **binary search tree**, but with up to three children rather than the binary tree's limit of two. Like other prefix trees, a ternary search tree can be used as an **associative map** structure with the ability for incremental **string search**. However, ternary search trees are more space efficient compared to standard prefix trees, at the cost of speed. Common applications for ternary search trees include **spell-checking** and **auto-completion**.

#### 9.1.1 Description

Each node of a ternary search tree stores a single **character**, an **object** (or a pointer to an object depending on implementation), and pointers to its three children conventionally named “equal kid” “lo kid” and “hi kid.”<sup>[1][2]</sup> A node may also have a pointer to its parent node as well as an indicator as to whether or not the node marks the end of a word.<sup>[1]</sup> The lo kid pointer must point to a node whose character value is less than the current node. The hi kid pointer must point to a node whose character is greater than the current node.<sup>[2]</sup> The figure below shows a ternary search tree with the strings “as”, “at”, “cup”, “cute”, “he”, “i” and “us”:

```
c /| \ a u h ||| \ t t e u //| | s p e i s
```

As with other trie data structures, each node in a ternary search tree represents a prefix of the stored strings. All strings in the middle subtree of a node start with that prefix.

#### 9.1.2 Ternary search tree operations

##### Look up

To look up a particular node or the data associated with a node, a string key is needed. A lookup procedure begins by checking the root node of the tree and determining which of the following conditions has occurred. If the

first character of the string is less than the character in the root node, a recursive lookup can be called on the tree whose root is the lo kid of the current root. Similarly, if the first character is greater than the current node in the tree, then a recursive call can be made to the tree whose root is the hi kid of the current node.<sup>[2]</sup> As a final case, if the first character of the string is equal to the character of the current node then the function returns the node if there are no more characters in the key. If there are more characters in the key then the first character of the key must be removed and a recursive call is made given the equal kid node and the modified key.<sup>[2]</sup> This can also be written in a non-recursive way by using a pointer to the current node and a pointer to the current character of the key.<sup>[2]</sup>

##### Insertion

Inserting a value into a ternary search can be defined recursively much as lookups are defined. This recursive method is continually called on nodes of the tree given a key which gets progressively shorter by pruning characters off the front of the key. If this method reaches a node that has not been created, it creates the node and assigns it the character value of the first character in the key. Whether a new node is created or not, the method checks to see if the first character in the string is greater than or less than the character value in the node and makes a recursive call on the appropriate node as in the lookup operation. If, however, the key's first character is equal to the node's value then the insertion procedure is called on the equal kid and the key's first character is pruned away.<sup>[2]</sup> Like **binary search trees** and other **data structures**, ternary search trees can become degenerate depending on the order of the keys.<sup>[3]</sup> Inserting keys in order is one way to attain the worst possible degenerate tree.<sup>[2]</sup> Inserting the keys in random order often produces a well-balanced tree.<sup>[2]</sup>

##### Running Time

The running time of ternary search trees varies significantly with the input. Ternary search trees run best when given several similar strings, especially when those strings

share a common prefix. Alternatively, ternary search trees are effective when storing a large number of relatively short strings (such as words in a dictionary).<sup>[2]</sup> Running times for ternary search trees are similar to [binary search trees](#) in that they typically run in logarithmic time but can run in linear time in the degenerate case.

Time complexities for ternary search tree operations:<sup>[2]</sup>

### 9.1.3 Comparison to other data structures

#### Tries

While being slower than other [prefix trees](#), ternary search trees can be better suited for larger data sets due to their space-efficiency.<sup>[2]</sup>

#### Hash maps

Hashtables can also be used in place of ternary search trees for mapping strings to values. However, hash maps also frequently use more memory than ternary search trees (but not as much as tries). Additionally, hash maps are typically slower at reporting a string that is not in the same data structure because it must compare the entire string rather than just the first few characters. There is some evidence that shows ternary search trees running faster than hash maps.<sup>[2]</sup> Additionally, hash maps do not allow for many of the uses of ternary search trees such as near-neighbor lookups.

### 9.1.4 Uses

Ternary search trees can be used to solve many problems in which a large number of strings must be stored and retrieved in an arbitrary order. Some of the most common or most useful of these are below:

- Anytime a [trie](#) could be used but a less memory-consuming structure is preferred.<sup>[2]</sup>
- A quick and space-saving data structure for [mapping](#) strings to other data.<sup>[3]</sup>
- To implement an [auto-complete](#) feature.<sup>[1]</sup>
- As a [spell check](#).<sup>[4]</sup>
- [Near-neighbor searching](#) (Of which a spell-check is a special case)<sup>[2]</sup>
- As a [database](#) especially when indexing by several non-key fields is desirable<sup>[4]</sup>
- In place of a [hash table](#).<sup>[4]</sup>

### 9.1.5 See also

- [Three-way radix quicksort](#)

### 9.1.6 References

- [1] Ostrovsky, Igor. “Efficient auto-complete with a ternary search tree”.
- [2] Dobbs. “Ternary Search Trees”.
- [3] Wrobel, Lukasz. “Ternary Search Tree”.
- [4] Flint, Wally. “Plant your data in a ternary search tree”.

### 9.1.7 External links

- [Ternary Search Trees](#)

## 9.2 And-or tree

An [and-or tree](#) is a graphical representation of the reduction of [problems](#) (or [goals](#)) to [conjunctions](#) and [disjunctions](#) of subproblems (or subgoals).

### 9.2.1 Example

The and-or tree:

represents the [search space](#) for solving the problem P, using the goal-reduction methods:

P if Q and R

P if S

Q if T

Q if U

### 9.2.2 Definitions

Given an initial problem P0 and set of problem solving methods of the form:

P if P1 and ... and Pn

the associated and-or tree is a set of labelled nodes such that:

1. The root of the tree is a node labelled by P0.
2. For every node N labelled by a problem or subproblem P and for every method of the form P if P1 and ... and Pn, there exists a set of children nodes N1, ..., Nn of the node N, such that each node Ni is labelled by Pi. The nodes are conjoined by an arc, to distinguish them from children of N that might be associated with other methods.

A node N, labelled by a problem P, is a success node if there is a method of the form P if nothing (i.e., P is a “fact”). The node is a failure node if there is no method for solving P.

If all of the children of a node N, conjoined by the same arc, are success nodes, then the node N is also a success node. Otherwise the node is a failure node.

### 9.2.3 Search strategies

An and-or tree specifies only the search space for solving a problem. Different **search strategies** for searching the space are possible. These include searching the tree depth-first, breadth-first, or best-first using some measure of desirability of solutions. The search strategy can be sequential, searching or generating one node at a time, or parallel, searching or generating several nodes in parallel.

### 9.2.4 Relationship with logic programming

The methods used for generating and-or trees are propositional logic programs (without variables). In the case of logic programs containing variables, the solutions of conjoint sub-problems must be compatible. Subject to this complication, sequential and parallel search strategies for and-or trees provide a computational model for executing logic programs.

### 9.2.5 Relationship with two-player games

And-or trees can also be used to represent the search spaces for two-person games. The root node of such a tree represents the problem of one of the players winning the game, starting from the initial state of the game. Given a node N, labelled by the problem P of the player winning the game from a particular state of play, there exists a single set of conjoint children nodes, corresponding to all of the opponents responding moves. For each of these children nodes, there exists a set of non-conjoint children nodes, corresponding to all of the player's defending moves.

For solving game trees with **proof-number search** family of algorithms, game trees are to be mapped to And/Or trees. MAX-nodes (i.e. maximizing player to move) are represented as OR nodes, MIN-nodes map to AND nodes. The mapping is possible, when the search is done with only a binary goal, which usually is “player to move wins the game”.

### 9.2.6 Bibliography

- Luger, George F.; Stubblefield, William A. (1993). *Artificial intelligence: structures and strategies for complex problem solving* (2 ed.). The Benjamin/Cummings. ISBN 978-0-8053-4785-2. Retrieved 28 February 2013.

Nilsson, Nils J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann. ISBN 978-1-55860-467-4. Retrieved 28 February 2013.

- Nilsson, Nils J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann. ISBN 978-1-55860-467-4. Retrieved 28 February 2013.

## 9.3 (a,b)-tree

In computer science, an **(a,b) tree** is a kind of balanced search tree.

An (a,b)-tree has all of its **leaves** at the same depth, and all internal **nodes** except for the root have between a and b **children**, where a and b are integers such that  $2 \leq a \leq (b+1)/2$ . The root has, if it is not a leaf, between 2 and b children.

### 9.3.1 Definition

Let a, b be positive integers such that  $2 \leq a \leq (b+1)/2$ . Then a rooted tree T is an (a,b)-tree when:

- Every inner node except the root has at least a and at most b children.
- The root has at most b children.
- All paths from the root to the leaves are of the same length.

### 9.3.2 Inner node representation

Every inner node v has the following representation:

- Let  $\rho_v$  be the number of child nodes of node v.
- Let  $S_v[1 \dots \rho_v]$  be pointers to child nodes.
- Let  $H_v[1 \dots \rho_v - 1]$  be an array of keys such that  $H_v[i]$  equals the largest key in the subtree pointed to by  $S_v[i]$ .

### 9.3.3 See also

- B-tree
- 2-3 tree
- 2-4 tree

### 9.3.4 References

- Black, Paul E. "(a,b)-tree". *Dictionary of Algorithms and Data Structures*. NIST.

## 9.4 Link/cut tree

A **link/cut tree** is a data structure for representing a **forest** (= set of trees).

Our ultimate goal is: given a certain node in the forest, find the root of the tree it belongs to (in order to check whether two nodes belong to the same tree). The represented forest is given and it might be unbalanced, so if we represent the forest as a plain collection of its trees, it might take us a long time to find the root of a given node.

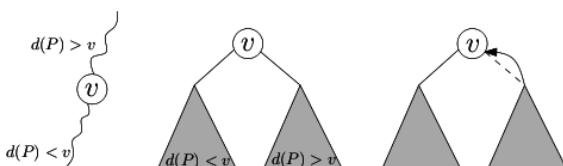
However, if we represent each tree in the forest as a link/cut tree, we can find which tree an element belongs to in  $O(\log(n))$  amortized time. Moreover, we can quickly adjust the collection of link/cut trees to changes in the represented forest. In particular, we can adjust it to merge (link) and split (cut) in  $O(\log(n))$  amortized time.

Link-Cut trees divide each tree in the represented forest into vertex-disjoint paths, where each path is represented by an auxiliary tree (often **splay trees**, though the original paper pre-dates splay trees and thus uses biased binary search trees). The nodes in the auxiliary trees are keyed by their depth in the corresponding represented tree. In one variation, *Naive Partitioning*, the paths are determined by the most recently accessed paths and nodes, similar to *Tango Trees*. In *Partitioning by Size* paths are determined by the heaviest child (child with the most children) of the given node. This gives a more complicated structure, but reduces the cost of the operations from amortized  $O(\log n)$  to worst case  $O(\log n)$ . It has uses in solving a variety of network flow problems and to jive data sets.

In the original publication, Sleator and Tarjan referred to link/cut trees as “dynamic trees”, or “dynamic dyno trees”.

### 9.4.1 Structure

We take a tree where each node has an arbitrary degree of unordered nodes and split it into paths. We call this the *represented tree*. These paths are represented internally by auxiliary trees (here we will use splay trees), where the nodes from left to right represent the path from root to the last node on the path. Nodes that are connected in the represented tree that are not on the same preferred path (and therefore not in the same auxiliary tree) are connected via a *path-parent pointer*. This pointer is stored in the root of the auxiliary tree representing the path.

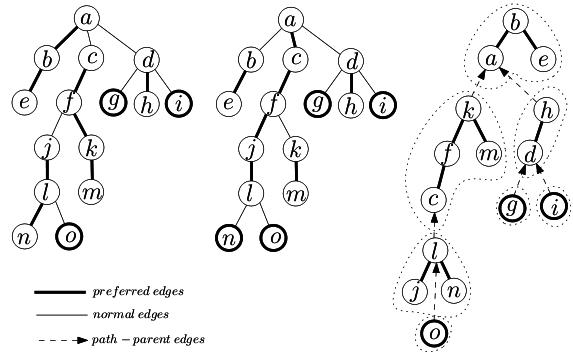


Demonstrating how nodes are stored by depth in the link-cut tree

### Preferred Paths

When an access to a node  $v$  is made on the *represented tree*, the path that is taken becomes the **preferred path**. The **preferred child** of a node is the last child that was on the access path, or null if the last access was to  $v$  or if no accesses were made to this particular branch of the tree. A **preferred edge** is the edge that connects the **preferred child** to  $v$ .

In an alternate version, preferred paths are determined by the heaviest child.



Showing how a link cut tree transforms preferred paths into a forest of trees.

### 9.4.2 Operations

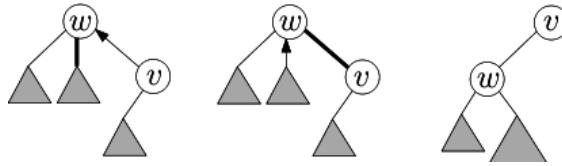
The operations we are interested in are `FindRoot(Node v)`, `Cut(Node v)`, `Link(Node v, Node w)`, and `Path(Node v)`. Every operation is implemented using the `Access(Node v)` subroutine. When we *access* a vertex  $v$ , the preferred path of the represented tree is changed to a path from the root  $R$  of the represented tree to the node  $v$ . If a node on the access path previously had a preferred child  $u$ , and the path now goes to child  $w$ , the old *preferred edge* is deleted (changed to a *path-parent pointer*), and the new path now goes through  $u$ .

#### Access

After performing an access to node  $v$ , it will no longer have any preferred children, and will be at the end of the path. Since nodes in the auxiliary tree are keyed by depth, this means that any nodes to the right of  $v$  in the auxiliary tree must be disconnected. In a splay tree this is a relatively simple procedure; we splay at  $v$ , which brings  $v$  to the root of the auxiliary tree. We then disconnect the right subtree of  $v$ , which is every node that came below it on the previous preferred path. The root of the disconnected tree will have a path-parent pointer, which we point to  $v$ .

We now walk up the represented tree to the root  $R$ , breaking and resetting the preferred path where necessary. To do this we follow the path-parent pointer from  $v$  (since  $v$

is now the root, we have direct access to the path-parent pointer). If the path that  $v$  is on already contains the root  $R$  (since the nodes are keyed by depth, it would be the left most node in the auxiliary tree), the path-parent pointer will be null, and we are done the access. Otherwise we follow the pointer to some node on another path  $w$ . We want to break the old preferred path of  $w$  and reconnect it to the path  $v$  is on. To do this we splay at  $w$ , and disconnect its right subtree, setting its path-parent pointer to  $w$ . Since all nodes are keyed by depth, and every node in the path of  $v$  is deeper than every node in the path of  $w$  (since they are children of  $w$  in the represented tree), we simply connect the tree of  $v$  as the right child of  $w$ . We splay at  $v$  again, which, since  $v$  is a child of the root  $w$ , simply rotates  $v$  to root. We repeat this entire process until the path-parent pointer of  $v$  is null, at which point it is on the same preferred path as the root of the represented tree  $R$ .



*During an access old preferred paths are broken and replaced with path-parent pointers, while the accessed node is splayed to the root of the tree*

### FindRoot

FindRoot refers to finding the root of the represented tree that contains the node  $v$ . Since the *access* subroutine puts  $v$  on the preferred path, we first execute an access. Now the node  $v$  is on the same preferred path, and thus the same auxiliary tree as the root  $R$ . Since the auxiliary trees are keyed by depth, the root  $R$  will be the leftmost node of the auxiliary tree. So we simply choose the left child of  $v$  recursively until we can go no further, and this node is the root  $R$ . The root may be linearly deep (which is worst case for a splay tree), we therefore splay it so that the next access will be quick.

### Cut

Here we would like to cut the represented tree at node  $v$ . First we access  $v$ . This puts all the elements lower than  $v$  in the represented tree as the right child of  $v$  in the auxiliary tree. All the elements now in the left subtree of  $v$  are the nodes higher than  $v$  in the represented tree. We therefore disconnect the left child of  $v$  (which still maintains an attachment to the original represented tree through its path-parent pointer). Now  $v$  is the root of a represented tree. Accessing  $v$  breaks the preferred path below  $v$  as well, but that subtree maintains its connection to  $v$  through its path-parent pointer.

### Link

If  $v$  is a tree root and  $w$  is a vertex in another tree, link the trees containing  $v$  and  $w$  by adding the edge( $v$ ,  $w$ ), making  $w$  the parent of  $v$ . To do this we access both  $v$  and  $w$  in their respective trees, and make  $w$  the left child of  $v$ . Since  $v$  is the root, and nodes are keyed by depth in the auxiliary tree, accessing  $v$  means that  $v$  will have no left child in the auxiliary tree (since as root it is the minimum depth). Adding  $w$  as a left child effectively makes it the parent of  $v$  in the represented tree.

### Path

For this operation we wish to do some aggregate function over all the nodes (or edges) on the path to  $v$  (such as “sum” or “min” or “max” or “increase”, etc.). To do this we access  $v$ , which gives us an auxiliary tree with all the nodes on the path from root  $R$  to node  $v$ . The data structure can be augmented with data we wish to retrieve, such as min or max values, or the sum of the costs in the subtree, which can then be returned from a given path in constant time.

### Switch-Preferred-Child( $x, y$ )

1. path-parent(right( $x$ )) =  $x$
2. right( $x, y$ )

### Access( $v$ )

1. Switch-Path-Prent( $v, \text{null}$ )
2. **while**( $v$  is not root)
3.  $w = \text{path-parent}(v)$
4.  $\text{splay}(w)$
5. Switch-Path-Parent( $w, v$ )
6. path-parent( $v$ ) = **null**
7.  $v = w$
8.  $\text{splay}(v)$

### Link( $v, w$ )

1. Access( $v$ )
2. Access( $w$ )
3. left( $v$ ) =  $w$

### Cut( $x, y$ )

1. Access( $v$ )
2. left( $v$ ) = **null**

*Pseudo-code for link-cut tree operations*

### 9.4.3 Analysis

Cut and link have  $O(1)$  cost, plus that of the access. FindRoot has an  $O(\log n)$  amortized upper bound, plus the cost of the access. The data structure can be augmented with additional information (such as the min or max valued node in its subtrees, or the sum), depending on the implementation. Thus Path can return this information in constant time plus the access bound.

So the it remains to bound the *access* to find our running time.

Access makes use of splaying, which we know has an  $O(\log n)$  amortized upper bound. So the remaining analysis deals with the number of times we need to splay. This is equal to the number of preferred child changes (the number of edges changed in the preferred path) as we traverse up the tree.

We bound *access* by using a technique called **Heavy-Light Decomposition**.

#### Heavy-Light Decomposition

Main article: Heavy path decomposition

This technique calls an edge *heavy* or *light* depending on the number of nodes in the subtree.  $Size(v)$  represents the number of nodes in the subtree of  $v$  in the represented tree. An edge is called *heavy* if  $size(v) > \frac{1}{2} size(parent(v))$ . Thus we can see that each node can have at most 1 *heavy* edge. An edge that is not a *heavy* edge is referred to as a *light* edge.

The *light-depth* refers to the number of light edges on a given path from root to vertex  $v$ .  $Light-depth \leq \lg n$  because each time we traverse a light-edge we decrease the number of nodes by at least a factor of 2 (since it can have at most half the nodes of the parent).

So a given edge in the represented tree can be any of four possibilities: *heavy-preferred*, *heavy-unpreferred*, *light-preferred* or *light-unpreferred*.

First we prove an  $O(\log^2 n)$  upper bound.

**$O(\log^2 n)$  upper bound** The splay operation of the access gives us  $\log n$ , so we need to bound the number of accesses to  $\log n$  to prove the  $O(\log^2 n)$  upper bound.

Every change of preferred edge results in a new preferred edge being formed. So we count the number of preferred edges formed. Since there are at most  $\log n$  edges that are light on any given path, there are at most  $\log n$  light edges changing to preferred.

The number of heavy edges becoming preferred can be  $O(n)$  for any given operation, but it is  $O(\log n)$  amortized. Over a series of executions we can have  $n-1$  heavy edges become preferred (as there are at most  $n-1$  heavy edges

total in the represented tree), but from then on the number of heavy edges that become preferred is equal to the number of heavy edges that became unpreferred on a previous step. For every heavy edge that becomes unpreferred a light edge must become preferred. We have seen already that the number of light edges that can become preferred is at most  $\log n$ . So the number of heavy edges that become preferred for  $m$  operations is  $m(\log n) + (n - 1)$ . Over enough operations ( $m > n-1$ ) this averages to  $O(\log n)$ .

**Improving to  $O(\log n)$  upper bound** We have bound the number of preferred child changes at  $O(\log n)$ , so if we can show that each preferred child change has cost  $O(1)$  amortized we can bound the *access* operation at  $O(\log n)$ . This is done using the potential method.

Let  $s(v)$  be the number of nodes under  $v$  in the tree of auxiliary trees. Then the potential function  $\Phi = \sum_v \log s(v)$ . We know that the amortized cost of splaying is bounded by:

$$cost(splay(v)) \leq 3 (\log s(root(v)) - \log s(v)) + 1$$

We know that after splaying,  $v$  is the child of its path-parent node  $w$ . So we know that:

$$s(v) \leq s(w)$$

We use this inequality and the amortized cost of access to achieve a telescoping sum that is bounded by:

$$3 (\log s(R) - \log s(v)) + O(\# \text{ of preferred child changes})$$

where  $R$  is the root of the represented tree, and we know the number of preferred child changes is  $O(\log n)$ .  $s(R) = n$ , so we have  $O(\log n)$  amortized.

### 9.4.4 Application

Link-Cut trees can be used to solve the **dynamic connectivity** problem for acyclic graphs. Given two nodes  $x$  and  $y$ , they are connected if and only if  $\text{FindRoot}(x)=\text{FindRoot}(y)$ .

Another data structure that can be used for the same purpose is **Euler tour tree**.

### 9.4.5 See also

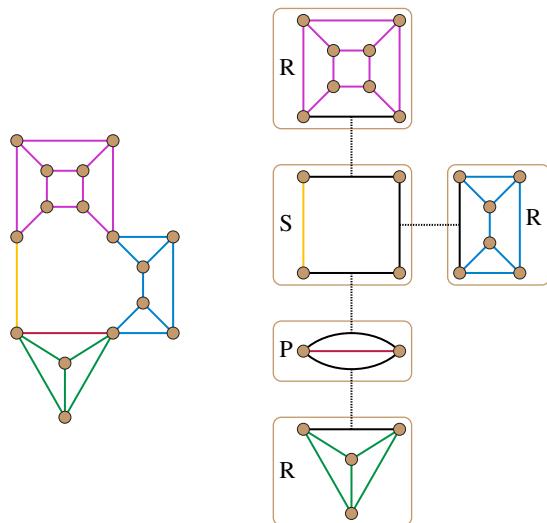
- Splay tree
- Potential method

### 9.4.6 Further reading

- Sleator, D. D.; Tarjan, R. E. (1983). “A Data Structure for Dynamic Trees”. *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81* (PDF). p. 114. doi:10.1145/800076.802464.

- Sleator, D. D.; Tarjan, R. E. (1985). “Self-Adjusting Binary Search Trees” (PDF). *Journal of the ACM* **32** (3): 652. doi:10.1145/3828.3835.
- Goldberg, A. V.; Tarjan, R. E. (1989). “Finding minimum-cost circulations by canceling negative cycles”. *Journal of the ACM* **36** (4): 873. doi:10.1145/76359.76368. — Application to min-cost circulation
- Link-Cut trees in: lecture notes in advanced data structures, Spring 2012, lecture 19. Prof. Erik Demaine, Scribes: Scribes: Justin Holmgren (2012), Jing Jian (2012), Maksim Stepanenko (2012), Mashhood Ishaque (2007).
- <http://compgeom.cs.uiuc.edu/~{}jeffe/teaching/datastructures/2006/notes/07-linkcut.pdf>

## 9.5 SPQR tree



A graph and its SPQR tree. The dashed black lines connect pairs of virtual edges, shown as black; the remaining edges are colored according to the triconnected component they belong to.

In graph theory, a branch of mathematics, the **triconnected components** of a biconnected graph are a system of smaller graphs that describe all of the 2-vertex cuts in the graph. An **SPQR tree** is a tree data structure used in computer science, and more specifically graph algorithms, to represent the triconnected components of a graph. The SPQR tree of a graph may be constructed in linear time<sup>[1]</sup> and has several applications in dynamic graph algorithms and graph drawing.

The basic structures underlying the SPQR tree, the triconnected components of a graph, and the connection between this decomposition and the planar embeddings of a planar graph, were first investigated by Saunders Mac

Lane (1937); these structures were used in efficient algorithms by several other researchers<sup>[2]</sup> prior to their formalization as the SPQR tree by Di Battista and Tamassia (1989, 1990, 1996).

### 9.5.1 Structure

An SPQR tree takes the form of an unrooted tree in which for each node  $x$  there is associated an undirected graph or multigraph  $G_x$ . The node, and the graph associated with it, may have one of four types, given the initials SPQR:

- In an S node, the associated graph is a **cycle graph** with three or more vertices and edges. This case is analogous to series composition in series-parallel graphs; the S stands for “series”.<sup>[3]</sup>
- In a P node, the associated graph is a **dipole graph**, a multigraph with two vertices and three or more edges, the **planar dual** to a cycle graph. This case is analogous to parallel composition in series-parallel graphs; the P stands for “parallel”.<sup>[3]</sup>
- In a Q node, the associated graph has a single real edge. This trivial case is necessary to handle the graph that has only one edge. In some works on SPQR trees, this type of node does not appear in the SPQR trees of graphs with more than one edge; in other works, all non-virtual edges are required to be represented by Q nodes with one real and one virtual edge, and the edges in the other node types must all be virtual.
- In an R node, the associated graph is a 3-connected graph that is not a cycle or dipole. The R stands for “rigid”: in the application of SPQR trees in planar graph embedding, the associated graph of an R node has a unique planar embedding.<sup>[3]</sup>

Each edge  $xy$  between two nodes of the SPQR tree is associated with two directed **virtual edges**, one of which is an edge in  $G_x$  and the other of which is an edge in  $G_y$ . Each edge in a graph  $G_x$  may be a virtual edge for at most one SPQR tree edge.

An SPQR tree  $T$  represents a 2-connected graph  $GT$ , formed as follows. Whenever SPQR tree edge  $xy$  associates the virtual edge  $ab$  of  $G_x$  with the virtual edge  $cd$  of  $G_y$ , form a single larger graph by merging  $a$  and  $c$  into a single supervertex, merging  $b$  and  $d$  into another single supervertex, and deleting the two virtual edges. That is, the larger graph is the **2-clique-sum** of  $G_x$  and  $G_y$ . Performing this gluing step on each edge of the SPQR tree produces the graph  $GT$ ; the order of performing the gluing steps does not affect the result. Each vertex in one of the graphs  $G_x$  may be associated in this way with a unique vertex in  $GT$ , the supervertex into which it was merged.

Typically, it is not allowed within an SPQR tree for two S nodes to be adjacent, nor for two P nodes to be adjacent, because if such an adjacency occurred the two nodes could be merged into a single larger node. With this assumption, the SPQR tree is uniquely determined from its graph. When a graph  $G$  is represented by an SPQR tree with no adjacent P nodes and no adjacent S nodes, then the graphs  $G_x$  associated with the nodes of the SPQR tree are known as the triconnected components of  $G$ .

### 9.5.2 Construction

The SPQR tree of a given 2-vertex-connected graph can be constructed in linear time.<sup>[1]</sup>

The problem of constructing the triconnected components of a graph was first solved in linear time by Hopcroft & Tarjan (1973). Based on this algorithm, Di Battista & Tamassia (1996) suggested that the full SPQR tree structure, and not just the list of components, should be constructible in linear time. After an implementation of a slower algorithm for SPQR trees was provided as part of the GDToolkit library, Gutwenger & Mutzel (2001) provided the first linear-time implementation. As part of this process of implementing this algorithm, they also corrected some errors in the earlier work of Hopcroft & Tarjan (1973).

The algorithm of Gutwenger & Mutzel (2001) includes the following overall steps.

1. Sort the edges of the graph by the pairs of numerical indices of their endpoints, using a variant of **radix sort** that makes two passes of **bucket sort**, one for each endpoint. After this sorting step, parallel edges between the same two vertices will be adjacent to each other in the sorted list and can be split off into a P-node of the eventual SPQR tree, leaving the remaining graph simple.
2. Partition the graph into split components; these are graphs that can be formed by finding a pair of separating vertices, splitting the graph at these two vertices into two smaller graphs (with a linked pair of virtual edges having the separating vertices as endpoints), and repeating this splitting process until no more separating pairs exist. The partition found in this way is not uniquely defined, because the parts of the graph that should become S-nodes of the SPQR tree will be subdivided into multiple triangles.
3. Label each split component with a P (a two-vertex split component with multiple edges), an S (a split component in the form of a triangle), or an R (any other split component). While there exist two split components that share a linked pair of virtual edges, and both components have type S or both have type P, merge them into a single larger component of the same time.

To find the split components, Gutwenger & Mutzel (2001) use **depth-first search** to find a structure that they call a palm tree; this is a **depth-first search tree** with its edges oriented away from the root of the tree, for the edges belonging to the tree, and towards the root for all other edges. They then find a special **preorder** numbering of the nodes in the tree, and use certain patterns in this numbering to identify pairs of vertices that can separate the graph into smaller components. When a component is found in this way, a **stack** data structure is used to identify the edges that should be part of the new component.

### 9.5.3 Usage

#### Finding 2-vertex cuts

With the SPQR tree of a graph  $G$  (without Q nodes) it is straightforward to find every pair of vertices  $u$  and  $v$  in  $G$  such that removing  $u$  and  $v$  from  $G$  leaves a disconnected graph, and the connected components of the remaining graphs:

- The two vertices  $u$  and  $v$  may be the two endpoints of a virtual edge in the graph associated with an R node, in which case the two components are represented by the two subtrees of the SPQR tree formed by removing the corresponding SPQR tree edge.
- The two vertices  $u$  and  $v$  may be the two vertices in the graph associated with a P node that has two or more virtual edges. In this case the components formed by the removal of  $u$  and  $v$  are represented by subtrees of the SPQR tree, one for each virtual edge in the node.
- The two vertices  $u$  and  $v$  may be two vertices in the graph associated with an S node such that either  $u$  and  $v$  are not adjacent, or the edge  $uv$  is virtual. If the edge is virtual, then the pair  $(u,v)$  also belongs to a node of type P and R and the components are as described above. If the two vertices are not adjacent then the two components are represented by two paths of the cycle graph associated with the S node and with the SPQR tree nodes attached to those two paths.

#### Representing all embeddings of planar graphs

If a planar graph is 3-connected, it has a unique planar embedding up to the choice of which face is the outer face and of orientation of the embedding: the faces of the embedding are exactly the nonseparating cycles of the graph. However, for a planar graph (with labeled vertices and edges) that is 2-connected but not 3-connected, there may be greater freedom in finding a planar embedding. Specifically, whenever two nodes in the SPQR tree of the graph are connected by a pair of virtual edges, it is possible to flip the orientation of one of the nodes relative

to the other one. Additionally, in a P node of the SPQR-tree, the different parts of the graph connected to virtual edges of the P node may be arbitrarily permuted. All planar representations may be described in this way.<sup>[4]</sup>

#### 9.5.4 See also

- Gomory–Hu tree, a different tree structure that characterizes the edge connectivity of a graph
- Tree decomposition, a generalization (no longer unique) to larger cuts

#### 9.5.5 Notes

- [1] Hopcroft & Tarjan (1973); Gutwenger & Mutzel (2001).
- [2] E.g., Hopcroft & Tarjan (1973) and Bienstock & Monma (1988), both of which are cited as precedents by Di Battista and Tamassia.
- [3] Di Battista & Tamassia (1989).
- [4] Mac Lane (1937).

#### 9.5.6 References

- Bienstock, Daniel; Monma, Clyde L. (1988), “On the complexity of covering vertices by faces in a planar graph”, *SIAM Journal on Computing* **17** (1): 53–76, doi:10.1137/0217004.
- Di Battista, Giuseppe; Tamassia, Roberto (1989), “Incremental planarity testing”, *Proc. 30th Annual Symposium on Foundations of Computer Science*, pp. 436–441, doi:10.1109/SFCS.1989.63515.
- Di Battista, Giuseppe; Tamassia, Roberto (1990), “On-line graph algorithms with SPQR-trees”, *Proc. 17th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science **443**, Springer-Verlag, pp. 598–611, doi:10.1007/BFb0032061.
- Di Battista, Giuseppe; Tamassia, Roberto (1996), “On-line planarity testing” (PDF), *SIAM Journal on Computing* **25** (5): 956–997, doi:10.1137/S0097539794280736.
- Gutwenger, Carsten; Mutzel, Petra (2001), “A linear time implementation of SPQR-trees”, *Proc. 8th International Symposium on Graph Drawing (GD 2000)*, Lecture Notes in Computer Science **1984**, Springer-Verlag, pp. 77–90, doi:10.1007/3-540-44541-2\_8.
- Hopcroft, John; Tarjan, Robert (1973), “Dividing a graph into triconnected components”, *SIAM Journal on Computing* **2** (3): 135–158, doi:10.1137/0202012.

- Mac Lane, Saunders (1937), “A structural characterization of planar combinatorial graphs”, *Duke Mathematical Journal* **3** (3): 460–472, doi:10.1215/S0012-7094-37-00336-3.

#### 9.5.7 External links

- SPQR tree implementation in the Open Graph Drawing Framework.
- The tree of the triconnected components Java implementation in the jBPT library (see TCTree class).

### 9.6 Spaghetti stack

In computer science, an **in-tree** or **parent pointer tree** is an N-ary tree data structure in which each node has a pointer to its **parent node**, but no pointers to **child nodes**. When used to implement a set of **stacks**, the structure is called a **spaghetti stack**, **cactus stack** or **saguaro stack** (after the **saguaro**, a kind of cactus).<sup>[1]</sup> Parent pointer trees are also used as **disjoint-set** data structures.

The structure can be regarded as a set of singly linked lists that **share** part of their structure, in particular, their tails. From any node, one can traverse to ancestors of the node, but not to any other node.

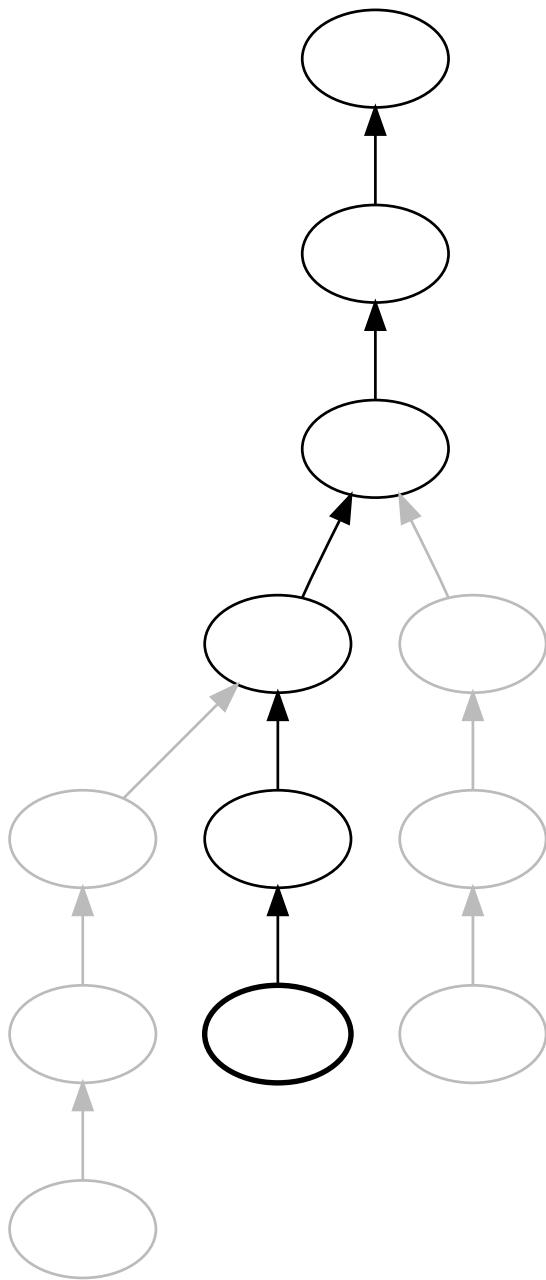
#### 9.6.1 Use as call stacks

Spaghetti stack structures arise in situations when records are dynamically pushed and popped onto a stack as execution progresses, but references to the popped records remain in use.

For example, a **compiler** for a language such as **C** creates a spaghetti stack as it opens and closes **symbol tables** representing block **scopes**. When a new block scope is opened, a symbol table is pushed onto a stack. When the closing curly brace is encountered, the scope is closed and the symbol table is popped. But that symbol table is remembered, rather than destroyed. And of course it remembers its higher level “parent” symbol table and so on. Thus when the compiler is later performing translations over the **abstract syntax tree**, for any given expression, it can fetch the symbol table representing that expression’s environment and can resolve references to identifiers. If the expression refers to a variable **X**, it is first sought after in the leaf symbol table representing the inner-most lexical scope, then in the parent and so on.

#### Use in programming language runtimes

The term **spaghetti stack** is closely associated with implementations of programming languages that support **continuations**. Spaghetti stacks are used to implement



Spaghetti stack with an "active" stack frame highlighted

the actual **run-time** stack containing variable bindings and other environmental features. When continuations must be supported, a function's local variables cannot be destroyed when that function returns: a saved continuation may later re-enter into that function, and will expect not only the variables there to be intact, but it will also expect the entire stack to be present so the function is able to return again. To resolve this problem, **stack frames** can be **dynamically allocated** in a spaghetti stack structure, and simply left behind to be **garbage collected** when no continuations refer to them any longer. This type of structure also solves both the upward and downward **funarg** problems, so first-class lexical **closures** are readily implemented in that substrate also.

Examples of languages that use spaghetti stacks are:

- Languages having first-class continuations such as **Scheme** and **Standard ML of New Jersey**
- Languages where the execution stack can be inspected and modified at runtime such as **Smalltalk**
- **Felix**
- **Cilk**

### 9.6.2 See also

- Burroughs large systems
- Persistent data structure

### 9.6.3 References

- [1] Clinger, W.; Harthimer, A.; Ost, E. (1988). "Implementation strategies for continuations". *Proceedings of the 1988 ACM conference on LISP and functional programming - LFP '88*. pp. 124–131. doi:10.1145/62678.62692. ISBN 089791273X.

## 9.7 Disjoint-set data structure



MakeSet creates 8 singletons.



After some operations of Union, some sets are grouped together.

In computer science, a **disjoint-set data structure**, also called a **union–find data structure** or **merge–find set**, is a data structure that keeps track of a **set** of elements partitioned into a number of **disjoint** (nonoverlapping) subsets. It supports two useful operations:

- **Find:** Determine which subset a particular element is in. **Find** typically returns an item from this set that serves as its "representative". Also, by comparing the result of two **Find** operations, one can determine whether two elements are in the same subset.
- **Union:** Join two subsets into a single subset.

The other important operation, **MakeSet**, which makes a set containing only a given element (a **singleton**), is generally trivial. With these three operations, many practical partitioning problems can be solved (see the *Applications* section).

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its *representative*, to represent the set as a whole. Then, *Find*( $x$ ) returns the representative of the set that  $x$  belongs to, and *Union* takes two set representatives as its arguments.

### 9.7.1 Disjoint-set linked lists

A simple approach to creating a disjoint-set data structure is to create a *linked list* for each set. The element at the head of each list is chosen as its representative.

*MakeSet* creates a list of one element. *Union* appends the two lists, a constant-time operation. The drawback of this implementation is that *Find* requires  $O(n)$  or linear time to traverse the list backwards from a given element to the head of the list.

This can be avoided by including in each linked list node a pointer to the head of the list; then *Find* takes constant time, since this pointer refers directly to the set representative. However, *Union* now has to update each element of the list being appended to make it point to the head of the new combined list, requiring  $\Omega(n)$  time.

When the length of each list is tracked, the required time can be improved by always appending the smaller list to the longer. Using this *weighted-union heuristic*, a sequence of  $m$  *MakeSet*, *Union*, and *Find* operations on  $n$  elements requires  $O(m + n \log n)$  time.<sup>[1]</sup> For asymptotically faster operations, a different data structure is needed.

### Analysis of the naive approach

We now explain the bound  $O(n \log(n))$  above.

Suppose you have a collection of lists and each node of each list contains an object, the name of the list to which it belongs, and the number of elements in that list. Also assume that the sum of the number of elements in all lists is  $n$  (i.e. there are  $n$  elements overall). We wish to be able to merge any two of these lists, and update all of their nodes so that they still contain the name of the list to which they belong. The rule for merging the lists  $A$  and  $B$  is that if  $A$  is larger than  $B$  then merge the elements of  $B$  into  $A$  and update the elements that used to belong to  $B$ , and vice versa.

Choose an arbitrary element of list  $L$ , say  $x$ . We wish to count how many times in the worst case will  $x$  need to have the name of the list to which it belongs updated. The element  $x$  will only have its name updated when the list it belongs to is merged with another list of the same size or of greater size. Each time that happens, the size of the list to which  $x$  belongs at least doubles. So finally, the question is “how many times can a number double before it is the size of  $n$ ?” (then the list containing  $x$  will contain all  $n$  elements). The answer is exactly  $\log_2(n)$ .

So for any given element of any given list in the structure described, it will need to be updated  $\log_2(n)$  times in the worst case. Therefore updating a list of  $n$  elements stored in this way takes  $O(n \log(n))$  time in the worst case. A *find* operation can be done in  $O(1)$  for this structure because each node contains the name of the list to which it belongs.

A similar argument holds for merging the trees in the data structures discussed below. Additionally, it helps explain the time analysis of some operations in the *binomial heap* and *Fibonacci heap* data structures.

### 9.7.2 Disjoint-set forests

Disjoint-set forests are data structures where each set is represented by a *tree data structure*, in which each node holds a reference to its parent node (see *Parent pointer tree*). They were first described by Bernard A. Galler and Michael J. Fischer in 1964,<sup>[2]</sup> although their precise analysis took years.

In a disjoint-set forest, the representative of each set is the root of that set’s tree. *Find* follows parent nodes until it reaches the root. *Union* combines two trees into one by attaching the root of one to the root of the other. One way of implementing these might be:

```
function MakeSet(x) x.parent := x
function Find(x) if x.parent == x return x else return Find(x.parent)
function Union(x, y) xRoot := Find(x) yRoot := Find(y)
xRoot.parent := yRoot
```

In this naive form, this approach is no better than the linked-list approach, because the tree it creates can be highly unbalanced; however, it can be enhanced in two ways.

The first way, called *union by rank*, is to always attach the smaller tree to the root of the larger tree. Since it is the depth of the tree that affects the running time, the tree with smaller depth gets added under the root of the deeper tree, which only increases the depth if the depths were equal. In the context of this algorithm, the term *rank* is used instead of *depth* since it stops being equal to the depth if path compression (described below) is also used. One-element trees are defined to have a rank of zero, and whenever two trees of the same rank  $r$  are united, the rank of the result is  $r+1$ . Just applying this technique alone yields a worst-case running-time of  $O(\log n)$  per *MakeSet*, *Union*, or *Find* operation. Pseudocode for the improved *MakeSet* and *Union*:

```
function MakeSet(x) x.parent := x x.rank := 0
function Union(x, y) xRoot := Find(x) yRoot := Find(y)
if xRoot == yRoot return // x and y are not already in same set.
Merge them. if xRoot.rank < yRoot.rank xRoot.parent := yRoot
else if xRoot.rank > yRoot.rank yRoot.parent := xRoot
else yRoot.parent := xRoot xRoot.rank := xRoot.rank + 1
```

The second improvement, called *path compression*, is a way of flattening the structure of the tree whenever *Find* is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative. To effect this, as *Find* recursively traverses up the tree, it changes each node's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly. Here is the improved *Find*:

```
function Find(x) if x.parent != x x.parent := Find(x.parent) return x.parent
```

These two techniques complement each other; applied together, the *amortized* time per operation is only  $O(\alpha(n))$ , where  $\alpha(n)$  is the *inverse of the function*  $n = f(x) = A(x, x)$ , and  $A$  is the extremely fast-growing *Ackermann function*. Since  $\alpha(n)$  is the inverse of this function,  $\alpha(n)$  is less than 5 for all remotely practical values of  $n$ . Thus, the amortized running time per operation is effectively a small constant.

In fact, this is asymptotically optimal: Fredman and Saks showed in 1989 that  $\Omega(\alpha(n))$  words must be accessed by *any* disjoint-set data structure per operation on average.<sup>[3]</sup>

### 9.7.3 Applications

Disjoint-set data structures model the *partitioning* of a set, for example to keep track of the *connected components* of an *undirected graph*. This model can then be used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle. The *Union–Find* algorithm is used in high-performance implementations of *unification*.<sup>[4]</sup>

This data structure is used by the *Boost Graph Library* to implement its *Incremental Connected Components* functionality. It is also used for implementing Kruskal's algorithm to find the *minimum spanning tree* of a graph.

Note that the implementation as disjoint-set forests doesn't allow deletion of edges—even without path compression or the rank heuristic.

### 9.7.4 History

While the ideas used in disjoint-set forests have long been familiar, Robert Tarjan was the first to prove the upper bound (and a restricted version of the lower bound) in terms of the *inverse Ackermann function*, in 1975.<sup>[5]</sup> Until this time the best bound on the time per operation, proven by Hopcroft and Ullman,<sup>[6]</sup> was  $O(\log^* n)$ , the *iterated logarithm* of  $n$ , another slowly growing function (but not quite as slow as the *inverse Ackermann function*).

Tarjan and Van Leeuwen also developed one-pass *Find* algorithms that are more efficient in practice while retaining the same worst-case complexity.<sup>[7]</sup>

In 2007, Sylvain Conchon and Jean-Christophe Filliâtre developed a *persistent* version of the disjoint-set forest data structure, allowing previous versions of the structure to be efficiently retained, and formalized its correctness using the proof assistant *Coq*.<sup>[8]</sup> However, the implementation is only asymptotic if used *ephemerally* or if the same version of the structure is repeatedly used with limited backtracking.

### 9.7.5 See also

- *Partition refinement*, a different data structure for maintaining disjoint sets, with updates that split sets apart rather than merging them together
- *Dynamic connectivity*

### 9.7.6 References

- [1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), “Chapter 21: Data structures for Disjoint Sets”, *Introduction to Algorithms* (Second ed.), MIT Press, pp. 498–524, ISBN 0-262-03293-7
- [2] Galler, Bernard A.; Fischer, Michael J. (May 1964), “An improved equivalence algorithm”, *Communications of the ACM* 7 (5): 301–303, doi:10.1145/364099.364331. The paper originating disjoint-set forests.
- [3] Fredman, M.; Saks, M. (May 1989), “The cell probe complexity of dynamic data structures”, *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*: 345–354, Theorem 5: Any *CPROBE*( $\log n$ ) implementation of the set union problem requires  $\Omega(m \alpha(m, n))$  time to execute  $m$  *Find*'s and  $n-1$  *Union*'s, beginning with  $n$  singleton sets.
- [4] Knight, Kevin (1989). “Unification: A multidisciplinary survey”. *ACM Computing Surveys* 21: 93–124. doi:10.1145/62029.62030.
- [5] Tarjan, Robert Endre (1975). “Efficiency of a Good But Not Linear Set Union Algorithm”. *Journal of the ACM* 22 (2): 215–225. doi:10.1145/321879.321884.
- [6] Hopcroft, J. E.; Ullman, J. D. (1973). “Set Merging Algorithms”. *SIAM Journal on Computing* 2 (4): 294–303. doi:10.1137/0202024.
- [7] Tarjan, Robert E.; van Leeuwen, Jan (1984), “Worst-case analysis of set union algorithms”, *Journal of the ACM* 31 (2): 245–281, doi:10.1145/62.2160
- [8] Conchon, Sylvain; Filliâtre, Jean-Christophe (October 2007), “A Persistent Union-Find Data Structure”, *ACM SIGPLAN Workshop on ML*, Freiburg, Germany

### 9.7.7 External links

- C++ implementation, part of the Boost C++ libraries
- A Java implementation with an application to color image segmentation, Statistical Region Merging (SRM), IEEE Trans. Pattern Anal. Mach. Intell. 26(11): 1452–1458 (2004)
- Java applet: A Graphical Union–Find Implementation, by Rory L. P. McGuire
- *Wait-free Parallel Algorithms for the Union–Find Problem*, a 1994 paper by Richard J. Anderson and Heather Woll describing a parallelized version of Union–Find that never needs to block
- Python implementation
- Visual explanation and C# code

# Chapter 10

## Space-partitioning trees

### 10.1 Space partitioning

In mathematics, **space partitioning** is the process of dividing a space (usually a Euclidean space) into two or more disjoint subsets (see also [partition of a set](#)). In other words, space partitioning divides a space into non-overlapping regions. Any point in the space can then be identified to lie in exactly one of the regions.

#### 10.1.1 Overview

Space-partitioning systems are often [hierarchical](#), meaning that a space (or a region of space) is divided into several regions, and then the same space-partitioning system is recursively applied to each of the regions thus created. The regions can be organized into a [tree](#), called a **space-partitioning tree**.

Most space-partitioning systems use [planes](#) (or, in higher dimensions, [hyperplanes](#)) to divide space: points on one side of the plane form one region, and points on the other side form another. Points exactly on the plane are usually arbitrarily assigned to one or the other side. Recursively partitioning space using planes in this way produces a [BSP tree](#), one of the most common forms of space partitioning.

#### 10.1.2 Use in computer graphics

Space partitioning is particularly important in [computer graphics](#), especially heavily used in [ray tracing](#), where it is frequently used to organize the objects in a virtual scene. A typical scene may contain millions of polygons. Performing a ray/polygon intersection test with each would be a very computationally expensive task.

Storing objects in a space-partitioning [data structure](#) ([kd-tree](#) or [BSP](#) for example) makes it easy and fast to perform certain kinds of geometry queries – for example in determining whether a ray intersects an object, space partitioning can reduce the number of intersection test to just a few per primary ray, yielding a logarithmic [time complexity](#) with respect to the number of polygons.<sup>[1][2][3]</sup>

Space partitioning is also often used in [scanline algo-](#)

rithms to eliminate the polygons out of the camera's viewing frustum, limiting the number of polygons processed by the pipeline. There is also a usage in [collision detection](#): determining whether two objects are close to each other can be much faster using space partitioning.

#### 10.1.3 Other uses

In [integrated circuit design](#), an important step is [design rule check](#). This step ensures that the completed design is manufacturable. The check involves rules that specify widths and spacings and other geometry patterns. A modern design can have billions of polygons that represent wires and transistors. Efficient checking relies heavily on geometry query. For example, a rule may specify that any polygon must be at least  $n$  nanometers from any other polygon. This is converted into a geometry query by enlarging a polygon by  $n$  at all sides and query to find all intersecting polygons.

#### 10.1.4 Types of space partitioning data structures

Common space partitioning systems include:

- [BSP trees](#)
- [Quadtrees](#)
- [Octrees](#)
- [kd-trees](#)
- [Bins](#)
- [R-trees](#)
- [Bounding volume hierarchies](#)
- [SEADSs](#).

#### 10.1.5 References

- [1] Tomas Nikodym (2010). "Ray Tracing Algorithm For Interactive Applications" (PDF). *Czech Technical University, FEE*.

- [2] Ingo Wald, William R. Mark et al. (2007). "State of the Art in Ray Tracing Animated Scenes". *EUROGRAPHICS*. CiteSeerX: 10.1.1.108.8495.
- [3] Ray Tracing - Auxiliary Data Structures

## 10.2 Binary space partitioning

For the .BSP file extension, see [BSP \(file format\)](#).

In computer science, **binary space partitioning (BSP)** is a method for recursively subdividing a space into convex sets by [hyperplanes](#). This subdivision gives rise to a representation of objects within the space by means of a [tree data structure](#) known as a **BSP tree**.

Binary space partitioning was developed in the context of 3D computer graphics,<sup>[1][2]</sup> where the structure of a BSP tree allows spatial information about the objects in a scene that is useful in [rendering](#), such as their ordering from front-to-back with respect to a viewer at a given location, to be accessed rapidly. Other applications include performing geometrical operations with shapes ([constructive solid geometry](#)) in [CAD](#),<sup>[3]</sup> [collision detection](#) in robotics and 3-D video games, ray tracing and other computer applications that involve handling of complex spatial scenes.

### 10.2.1 Overview

Binary space partitioning is a generic process of [recursively](#) dividing a scene into two until the partitioning satisfies one or more requirements. It can be seen as a generalisation of other spatial tree structures such as [k-d trees](#) and [quadtree](#)s, one where hyperplanes that partition the space may have any orientation, rather than being aligned with the coordinate axes as they are in [k-d trees](#) or [quadtree](#)s. When used in computer graphics to render scenes composed of planar [polygons](#), the partitioning planes are frequently (but not always) chosen to coincide with the planes defined by polygons in the scene.

The specific choice of partitioning plane and criterion for terminating the partitioning process varies depending on the purpose of the BSP tree. For example, in computer graphics rendering, the scene is divided until each node of the BSP tree contains only polygons that can render in arbitrary order. When [back-face culling](#) is used, each node therefore contains a convex set of polygons, whereas when rendering double-sided polygons, each node of the BSP tree contains only polygons in a single plane. In collision detection or ray tracing, a scene may be divided up into [primitives](#) on which collision or ray intersection tests are straightforward.

Binary space partitioning arose from the computer graphics need to rapidly draw three-dimensional scenes composed of polygons. A simple way to draw such scenes is

the [painter's algorithm](#), which produces polygons in order of distance from the viewer, back to front, painting over the background and previous polygons with each closer object. This approach has two disadvantages: time required to sort polygons in back to front order, and the possibility of errors in overlapping polygons. Fuchs and co-authors<sup>[2]</sup> showed that constructing a BSP tree solved both of these problems by providing a rapid method of sorting polygons with respect to a given viewpoint (linear in the number of polygons in the scene) and by subdividing overlapping polygons to avoid errors that can occur with the painter's algorithm. A disadvantage of binary space partitioning is that generating a BSP tree can be time-consuming. Typically, it is therefore performed once on static geometry, as a pre-calculation step, prior to rendering or other realtime operations on a scene. The expense of constructing a BSP tree makes it difficult and inefficient to directly implement moving objects into a tree.

BSP trees are often used by 3D [video games](#), particularly first-person shooters and those with indoor environments. Game engines utilising BSP trees include the [Doom engine](#) (probably the earliest game to use a BSP data structure was *Doom*), the [Quake engine](#) and its descendants. In video games, BSP trees containing the static geometry of a scene are often used together with a [Z-buffer](#), to correctly merge movable objects such as doors and characters onto the background scene. While binary space partitioning provides a convenient way to store and retrieve spatial information about polygons in a scene, it does not solve the problem of [visible surface determination](#).

### 10.2.2 Generation

The canonical use of a BSP tree is for rendering polygons (that are double-sided, that is, without [back-face culling](#)) with the painter's algorithm. Each polygon is designated with a front side and a back side which could be chosen arbitrarily and only affects the structure of the tree but not the required result.<sup>[2]</sup> Such a tree is constructed from an unsorted list of all the polygons in a scene. The recursive algorithm for construction of a BSP tree from that list of polygons is:<sup>[2]</sup>

1. Choose a polygon  $P$  from the list.
2. Make a node  $N$  in the BSP tree, and add  $P$  to the list of polygons at that node.
3. For each other polygon in the list:
  - (a) If that polygon is wholly in front of the plane containing  $P$ , move that polygon to the list of nodes in front of  $P$ .
  - (b) If that polygon is wholly behind the plane containing  $P$ , move that polygon to the list of nodes behind  $P$ .

- (c) If that polygon is intersected by the plane containing  $P$ , split it into two polygons and move them to the respective lists of polygons behind and in front of  $P$ .
- (d) If that polygon lies in the plane containing  $P$ , add it to the list of polygons at node  $N$ .
- 4. Apply this algorithm to the list of polygons in front of  $P$ .
- 5. Apply this algorithm to the list of polygons behind  $P$ .

The following diagram illustrates the use of this algorithm in converting a list of lines or polygons into a BSP tree. At each of the eight steps (i.-viii.), the algorithm above is applied to a list of lines, and one new node is added to the tree.

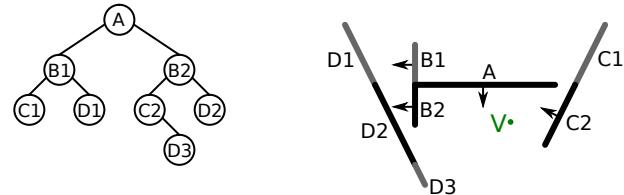
The final number of polygons or lines in a tree is often larger (sometimes much larger<sup>[2]</sup>) than the original list, since lines or polygons that cross the partitioning plane must be split into two. It is desirable to minimize this increase, but also to maintain reasonable **balance** in the final tree. The choice of which polygon or line is used as a partitioning plane (in step 1 of the algorithm) is therefore important in creating an efficient BSP tree.

### 10.2.3 Traversal

A BSP tree is **traversed** in a linear time, in an order determined by the particular function of the tree. Again using the example of rendering double-sided polygons using the painter's algorithm, to draw a polygon  $P$  correctly requires that all polygons behind the plane  $P$  lies in must be drawn first, then polygon  $P$ , then finally the polygons in front of  $P$ . If this drawing order is satisfied for all polygons in a scene, then the entire scene renders in the correct order. This procedure can be implemented by recursively traversing a BSP tree using the following algorithm.<sup>[2]</sup> From a given viewing location  $V$ , to render a BSP tree,

- 1. If the current node is a leaf node, render the polygons at the current node.
- 2. Otherwise, if the viewing location  $V$  is in front of the current node:
  - (a) Render the child BSP tree containing polygons behind the current node
  - (b) Render the polygons at the current node
  - (c) Render the child BSP tree containing polygons in front of the current node
- 3. Otherwise, if the viewing location  $V$  is behind the current node:
  - (a) Render the child BSP tree containing polygons in front of the current node

- (b) Render the polygons at the current node
- (c) Render the child BSP tree containing polygons behind the current node
- 4. Otherwise, the viewing location  $V$  must be exactly on the plane associated with the current node. Then:
  - (a) Render the child BSP tree containing polygons in front of the current node
  - (b) Render the child BSP tree containing polygons behind the current node



Applying this algorithm recursively to the BSP tree generated above results in the following steps:

- The algorithm is first applied to the root node of the tree, node  $A$ .  $V$  is in front of node  $A$ , so we apply the algorithm first to the child BSP tree containing polygons behind  $A$ 
  - This tree has root node  $B1$ .  $V$  is behind  $B1$  so first we apply the algorithm to the child BSP tree containing polygons in front of  $B1$ :
    - This tree is just the leaf node  $D1$ , so the polygon  $D1$  is rendered.
    - We then render the polygon  $B1$ .
    - We then apply the algorithm to the child BSP tree containing polygons behind  $B1$ :
      - This tree is just the leaf node  $C1$ , so the polygon  $C1$  is rendered.
  - We then draw the polygons of  $A$
  - We then apply the algorithm to the child BSP tree containing polygons in front of  $A$ 
    - This tree has root node  $B2$ .  $V$  is behind  $B2$  so first we apply the algorithm to the child BSP tree containing polygons in front of  $B2$ :
      - This tree is just the leaf node  $D2$ , so the polygon  $D2$  is rendered.
      - We then render the polygon  $B2$ .
      - We then apply the algorithm to the child BSP tree containing polygons behind  $B2$ :
        - This tree has root node  $C2$ .  $V$  is in front of  $C2$  so first we would apply the algorithm to the child BSP tree containing polygons behind  $C2$ . There is no such tree, however, so we continue.
        - We render the polygon  $C2$ .

- We apply the algorithm to the child BSP tree containing polygons in front of  $C2$
- This tree is just the leaf node  $D3$ , so the polygon  $D3$  is rendered.

The tree is traversed in linear time and renders the polygons in a far-to-near ordering ( $D1, B1, C1, A, D2, B2, C2, D3$ ) suitable for the painter's algorithm.

#### 10.2.4 Brushes

“Brushes” are templates, used in some 3D video games such as games based on the **Source game engine**, its predecessor the **Goldsrc engine**, **Unreal Engine**’s tool **Unreal Editor**, etc. to construct **levels**.<sup>[4]</sup> Brushes can be primitive shapes (such as cubes, spheres & cones), pre-defined shapes (such as staircases), or custom shapes (such as prisms and other **polyhedra**). Using **CSG** operations, complex rooms and objects can be created by adding, subtracting and intersecting brushes to and from one another.<sup>[5]</sup>

#### 10.2.5 Timeline

- 1969 Schumacker et al.<sup>[1]</sup> published a report that described how carefully positioned planes in a virtual environment could be used to accelerate polygon ordering. The technique made use of depth coherence, which states that a polygon on the far side of the plane cannot, in any way, obstruct a closer polygon. This was used in flight simulators made by GE as well as Evans and Sutherland. However, creation of the polygonal data organization was performed manually by scene designer.
- 1980 Fuchs et al.<sup>[2]</sup> extended Schumacker’s idea to the representation of 3D objects in a virtual environment by using planes that lie coincident with polygons to recursively partition the 3D space. This provided a fully automated and algorithmic generation of a hierarchical polygonal data structure known as a Binary Space Partitioning Tree (BSP Tree). The process took place as an off-line preprocessing step that was performed once per environment/object. At run-time, the view-dependent visibility ordering was generated by traversing the tree.
- 1981 Naylor’s Ph.D thesis containing a full development of both BSP trees and a graph-theoretic approach using strongly connected components for pre-computing visibility, as well as the connection between the two methods. BSP trees as a dimension independent spatial search structure was emphasized, with applications to visible surface determination. The thesis also included the first empirical data demonstrating that the size of the tree and the number of new polygons was reasonable (using a model of the Space Shuttle).
- 1983 Fuchs et al. describe a micro-code implementation of the BSP tree algorithm on an Ikonas frame buffer system. This was the first demonstration of real-time visible surface determination using BSP trees.
- 1987 Thibault and Naylor<sup>[3]</sup> described how arbitrary polyhedra may be represented using a BSP tree as opposed to the traditional b-rep (boundary representation). This provided a solid representation vs. a surface based-representation. Set operations on polyhedra were described using a tool, enabling **Constructive Solid Geometry** (CSG) in real-time. This was the fore runner of BSP level design using brushes, introduced in the Quake editor and picked up in the Unreal Editor.
- 1990 Naylor, Amanatides, and Thibault provide an algorithm for merging two BSP trees to form a new BSP tree from the two original trees. This provides many benefits including: combining moving objects represented by BSP trees with a static environment (also represented by a BSP tree), very efficient CSG operations on polyhedra, exact collisions detection in  $O(\log n * \log n)$ , and proper ordering of transparent surfaces contained in two interpenetrating objects (has been used for an x-ray vision effect).
- 1990 Teller and Séquin proposed the offline generation of potentially visible sets to accelerate visible surface determination in orthogonal 2D environments.
- 1991 Gordon and Chen [CHEN91] described an efficient method of performing front-to-back rendering from a BSP tree, rather than the traditional back-to-front approach. They utilised a special data structure to record, efficiently, parts of the screen that have been drawn, and those yet to be rendered. This algorithm, together with the description of BSP Trees in the standard computer graphics textbook of the day (*Computer Graphics: Principles and Practice*) was used by John Carmack in the making of *Doom*.
- 1992 Teller’s PhD thesis described the efficient generation of potentially visible sets as a pre-processing step to accelerate real-time visible surface determination in arbitrary 3D polygonal environments. This was used in *Quake* and contributed significantly to that game’s performance.
- 1993 Naylor answers the question of what characterizes a good BSP tree. He used expected case

models (rather than worst case analysis) to mathematically measure the expected cost of searching a tree and used this measure to build good BSP trees. Intuitively, the tree represents an object in a multi-resolution fashion (more exactly, as a tree of approximations). Parallels with Huffman codes and probabilistic binary search trees are drawn.

- 1993 Hayder Radha's PhD thesis described (natural) image representation methods using BSP trees. This includes the development of an optimal BSP-tree construction framework for any arbitrary input image. This framework is based on a new image transform, known as the Least-Square-Error (LSE) Partitioning Line (LPE) transform. H. Radha's thesis also developed an optimal rate-distortion (RD) image compression framework and image manipulation approaches using BSP trees.

### 10.2.6 References

- [1] Schumacker, Robert A. ;; Brand, Brigitte; Gilliland, Maurice G.; Sharp, Werner H (1969). Study for Applying Computer-Generated Images to Visual Simulation (Report). U.S. Air Force Human Resources Laboratory. p. 142. AFHRL-TR-69-14.
- [2] Fuchs, Henry; Kedem, Zvi. M; Naylor, Bruce F. (1980). "On Visible Surface Generation by A Priori Tree Structures". *SIGGRAPH '80 Proceedings of the 7th annual conference on Computer graphics and interactive techniques*. ACM, New York. pp. 124–133. doi:10.1145/965105.807481.
- [3] Thibault, William C.; Naylor, Bruce F. (1987). "Set operations on polyhedra using binary space partitioning trees". *SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. ACM, New York. pp. 153–162. doi:10.1145/37402.37421.
- [4] "Definition of Brush in the Valve Developer Community". Valve. Retrieved 2011-03-24.
- [5] "UDN – Two – BspBrushesTutorial". Epic Games, Inc. Retrieved 2012-04-21.

### 10.2.7 Additional references

- [NAYLOR90] B. Naylor, J. Amanatides, and W. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations", Computer Graphics (Siggraph '90), 24(3), 1990.
- [NAYLOR93] B. Naylor, "Constructing Good Partitioning Trees", Graphics Interface (annual Canadian CG conference) May, 1993.
- [CHEN91] S. Chen and D. Gordon. "Front-to-Back Display of BSP Trees." IEEE Computer Graphics & Algorithms, pp 79–85. September 1991.

- [RADHA91] H. Radha, R. Leonardi, M. Vetterli, and B. Naylor "Binary Space Partitioning Tree Representation of Images," *Journal of Visual Communications and Image Processing* 1991, vol. 2(3).
- [RADHA93] H. Radha, "Efficient Image Representation using Binary Space Partitioning Trees.", Ph.D. Thesis, Columbia University, 1993.
- [RADHA96] H. Radha, M. Vetterli, and R. Leonardi, "Image Compression Using Binary Space Partitioning Trees," *IEEE Transactions on Image Processing*, vol. 5, No.12, December 1996, pp. 1610–1624.
- [WINTER99] AN INVESTIGATION INTO REAL-TIME 3D POLYGON RENDERING USING BSP TREES. Andrew Steven Winter. April 1999. available online
- Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf (2000). *Computational Geometry* (2nd revised ed.). Springer-Verlag. ISBN 3-540-65620-0. Section 12: Binary Space Partitions: pp. 251–265. Describes a randomized Painter's Algorithm.
- Christer Ericson: *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Verlag Morgan Kaufmann, S. 349-382, Jahr 2005, ISBN 1-55860-732-3

### 10.2.8 External links

- BSP trees presentation
- Another BSP trees presentation
- A Java applet that demonstrates the process of tree generation
- A Master Thesis about BSP generating
- BSP Trees: Theory and Implementation
- BSP in 3D space

## 10.3 Segment tree

In computer science, a **segment tree** is a tree data structure for storing intervals, or segments. It allows querying which of the stored segments contain a given point. It can be implemented as a dynamic structure.<sup>[1]</sup> A similar data structure is the **interval tree**.

A segment tree for a set  $I$  of  $n$  intervals uses  $O(n \log n)$  storage and can be built in  $O(n \log n)$  time. Segment trees support searching for all the intervals that contain a query

point in  $O(\log n + k)$ ,  $k$  being the number of retrieved intervals or segments.<sup>[2]</sup> However, it can be modified to reduce the query time to  $O(\log n)$  for some types of queries, such as the finding the minimum value in the interval.<sup>[1]</sup>

Some applications of the segment tree are in the areas of computational geometry, and geographic information systems.

The segment tree can be generalized to higher dimension spaces as well.

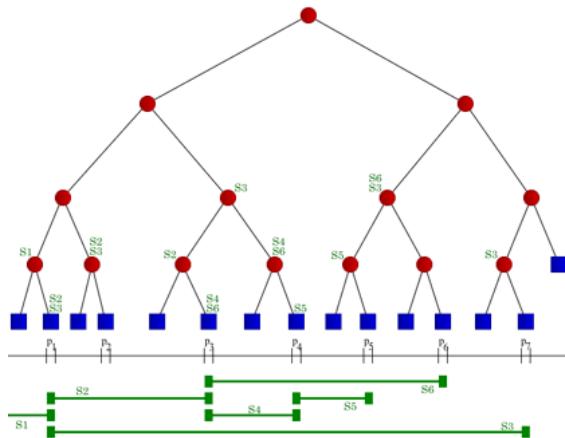
### 10.3.1 Structure description

*This section describes the structure of a segment tree in a one-dimensional space.*

Let  $S$  be a set of intervals, or segments. Let  $p_1, p_2, \dots, p_m$  be the list of distinct interval endpoints, sorted from left to right. Consider the partitioning of the real line induced by those points. The regions of this partitioning are called *elementary intervals*. Thus, the elementary intervals are, from left to right:

$$(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \dots, (p_{m-1}, p_m), [p_m, p_m], (p_m, +\infty)$$

That is, the list of elementary intervals consists of open intervals between two consecutive endpoints  $p_i$  and  $p_{i+1}$ , alternated with closed intervals consisting of a single endpoint. Single points are treated themselves as intervals because the answer to a query is not necessarily the same at the interior of an elementary interval and its endpoints.<sup>[3]</sup>



*Graphic example of the structure of the segment tree. This instance is built for the segments shown at the bottom.*

Given a set  $I$  of intervals, or segments, a segment tree  $T$  for  $I$  is structured as follows:

- $T$  is a binary tree.
- Its leaves correspond to the elementary intervals induced by the endpoints in  $I$ , in an ordered way: the leftmost leaf corresponds to the leftmost interval,

and so on. The elementary interval corresponding to a leaf  $v$  is denoted  $\text{Int}(v)$ .

- The internal nodes of  $T$  correspond to intervals that are the union of elementary intervals: the interval  $\text{Int}(N)$  corresponding to node  $N$  is the union of the intervals corresponding to the leaves of the tree rooted at  $N$ . That implies that  $\text{Int}(N)$  is the union of the intervals of its two children.
- Each node or leaf  $v$  in  $T$  stores the interval  $\text{Int}(v)$  and a set of intervals, in some data structure. This canonical subset of node  $v$  contains the intervals  $[x, x']$  from  $I$  such that  $[x, x']$  contains  $\text{Int}(v)$  and does not contain  $\text{Int}(\text{parent}(v))$ . That is, each segment in  $I$  stores the segments that span through its interval, but do not span through the interval of its parent.<sup>[4]</sup>

### 10.3.2 Storage requirements

*This section analyzes the storage cost of a segment tree in a one-dimensional space.*

A segment tree  $T$  on a set  $I$  of  $n$  intervals uses  $O(n \log n)$  storage.

*Proof:*

*Lemma:* Any interval  $[x, x']$  of  $I$  is stored in the canonical set for at most two nodes at the same depth.

*Proof:* Let  $v_1, v_2, v_3$  be the three nodes at the same depth, numbered from left to right; and let  $p(v)$  be the parent node of any given node  $v$ . Suppose  $[x, x']$  is stored at  $v_1$  and  $v_3$ . This means that  $[x, x']$  spans the whole interval from the left endpoint of  $\text{Int}(v_1)$  to the right endpoint of  $\text{Int}(v_3)$ . Note that all segments at a particular level are non-overlapping and ordered from left to right: this is true by construction for the level containing the leaves, and the property is not lost when moving from any level to the one above it by combining pairs of adjacent segments. Now either  $p(v_2) = p(v_1)$ , or the former is to the right of the latter (edges in the tree do not cross). In the first case,  $\text{Int}(p(v_2))$ 's leftmost point is the same as  $\text{Int}(v_1)$ 's leftmost point; in the second case,  $\text{Int}(p(v_2))$ 's leftmost point is to the right of  $\text{Int}(p(v_1))$ 's rightmost point, and therefore also to the right

of  $\text{Int}(v_1)$ 's rightmost point. In both cases,  $\text{Int}(p(v_2))$  begins at or to the right of  $\text{Int}(v_1)$ 's leftmost point. Similar reasoning shows that  $\text{Int}(p(v_2))$  ends at or to the left of  $\text{Int}(v_3)$ 's rightmost point.  $\text{Int}(p(v_2))$  must therefore be contained in  $[x, x']$ ; hence,  $[x, x']$  will not be stored at  $v_2$ .

The set  $I$  has at most  $4n + 1$  elementary intervals. Because  $T$  is a binary balanced tree with at most  $4n + 1$  leaves, its height is  $O(\log n)$ . Since any interval is stored at most twice at a given depth of the tree, that the total amount of storage is  $O(n \log n)$ .<sup>[5]</sup>

### 10.3.3 Construction

*This section describes the construction of a segment tree in a one-dimensional space.*

A segment tree from the set of segments  $I$ , can be built as follows. First, the endpoints of the intervals in  $I$  are sorted. The elementary intervals are obtained from that. Then, a balanced binary tree is built on the elementary intervals, and for each node  $v$  it is determined the interval  $\text{Int}(v)$  it represents. It remains to compute the canonical subsets for the nodes. To achieve this, the intervals in  $I$  are inserted one by one into the segment tree. An interval  $X = [x, x']$  can be inserted in a subtree rooted at  $T$ , using the following procedure:<sup>[6]</sup>

- If  $\text{Int}(T)$  is contained in  $X$  then store  $X$  at  $T$ , and finish.
- Else:
  - If  $X$  intersects the canonical subset of the left child of  $T$ , then insert  $X$  in that child, recursively.
  - If  $X$  intersects the canonical subset of the right child of  $T$ , then insert  $X$  in that child, recursively.

The complete construction operation takes  $O(n \log n)$  time,  $n$  being the number of segments in  $I$ .

*Proof*

Sorting the endpoints takes  $O(n \log n)$ . Building a balanced binary tree from the sorted endpoints, takes linear time on  $n$ .

The insertion of an interval  $X = [x, x']$  into the tree, costs  $O(\log n)$ .

*Proof:* Visiting every node takes constant time (assuming that canonical subsets are stored in

a simple data structure like a linked list). When we visit node  $v$ , we either store  $X$  at  $v$ , or  $\text{Int}(v)$  contains an endpoint of  $X$ . As proved above, an interval is stored at most twice at each level of the tree. There is also at most one node at every level whose corresponding interval contains  $x$ , and one node whose interval contains  $x'$ . So, at most four nodes per level are visited. Since there are  $O(\log n)$  levels, the total cost of the insertion is  $O(\log n)$ .<sup>[2]</sup>

### 10.3.4 Query

*This section describes the query operation of a segment tree in a one-dimensional space.*

A query for a segment tree, receives a point  $qx$ , and retrieves a list of all the segments stored which contain the point  $qx$ .

Formally stated; given a node (subtree)  $v$  and a query point  $qx$ , the query can be done using the following algorithm:

- Report all the intervals in  $\text{I}(v)$ .
- If  $v$  is not a leaf:
  - If  $qx$  is in  $\text{Int}(\text{left child of } v)$  then
    - Perform a query in the left child of  $v$ .
  - If  $qx$  is in  $\text{Int}(\text{right child of } v)$  then
    - Perform a query in the right child of  $v$ .

In a segment tree that contains  $n$  intervals, those containing a given query point can be reported in  $O(\log n + k)$  time, where  $k$  is the number of reported intervals.

*Proof:* The query algorithm visits one node per level of the tree, so  $O(\log n)$  nodes in total. In the other hand, at a node  $v$ , the segments in  $I$  are reported in  $O(1 + kv)$  time, where  $kv$  is the number of intervals at node  $v$ , reported. The sum of all the  $kv$  for all nodes  $v$  visited, is  $k$ , the number of reported segments.<sup>[5]</sup>

### 10.3.5 Generalization for higher dimensions

The segment tree can be generalized to higher dimension spaces, in the form of multi-level segment trees. In higher dimension versions, the segment tree stores a collection of axis-parallel (hyper-)rectangles, and can retrieve the

rectangles that contain a given query point. The structure uses  $O(n \log^d n)$  storage, and answers queries in  $O(\log^d n)$ .

The use of fractional cascading lowers the query time bound by a logarithmic factor. The use of the **interval tree** on the deepest level of associated structures lowers the storage bound with a logarithmic factor.<sup>[7]</sup>

### 10.3.6 Notes

The query that asks for all the intervals containing a given point, is often referred as *stabbing query*.<sup>[8]</sup>

The segment tree is less efficient than the interval tree for range queries in one dimension, due to its higher storage requirement:  $O(n \log n)$  against the  $O(n)$  of the interval tree. The importance of the segment tree is that the segments within each node's canonical subset can be stored in any arbitrary manner.<sup>[8]</sup>

For  $n$  intervals whose endpoints are in a small integer range (e.g., in the range  $[1, \dots, O(n)]$ ), optimal data structures exist with a linear preprocessing time and query time  $O(1+k)$  for reporting all  $k$  intervals containing a given query point.

Another advantage of the segment tree is that it can easily be adapted to counting queries; that is, to report the number of segments containing a given point, instead of reporting the segments themselves. Instead of storing the intervals in the canonical subsets, it can simply store the number of them. Such a segment tree uses linear storage, and requires an  $O(\log n)$  query time, so it is optimal.<sup>[9]</sup>

A version for higher dimensions of the interval tree and the **priority search tree** does not exist, that is, there is no clear extension of these structures that solves the analogous problem in higher dimensions. But the structures can be used as associated structure of segment trees.<sup>[7]</sup>

### 10.3.7 History

The segment tree was discovered by J. L. Bentley in 1977, in "Solutions to Klee's rectangle problems".<sup>[8]</sup>

### 10.3.8 References

- [1] "Range Minimum Query and Lowest Common Ancestor". *Topcoder*. Retrieved 30 Nov 2014.
- [2] (de Berg et al. 2000, p. 227)
- [3] (de Berg et al. 2000, p. 224)
- [4] (de Berg et al. 2000, pp. 225–226)
- [5] (de Berg et al. 2000, p. 226)
- [6] (de Berg et al. 2000, pp. 226–227)
- [7] (de Berg et al. 2000, p. 230)
- [8] (de Berg et al. 2000, p. 229)
- [9] (de Berg et al. 2000, pp. 229–230)

### 10.3.9 Sources cited

- de Berg, Mark; van Kreveld, Marc; Overmars, Mark; Schwarzkopf, Otfried (2000). "More Geometric Data Structures". *Computational Geometry: algorithms and applications* (2nd ed.). Springer-Verlag Berlin Heidelberg New York. doi:10.1007/978-3-540-77974-2. ISBN 3-540-65620-0.
- <http://www.cs.nthu.edu.tw/~{}wkhon/ds/ds10/tutorial/tutorial6.pdf>

## 10.4 Interval tree

In computer science, an **interval tree** is an ordered tree data structure to hold intervals. Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point. It is often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene. A similar data structure is the **segment tree**.

The trivial solution is to visit each interval and test whether it intersects the given point or interval, which requires  $\Theta(n)$  time, where  $n$  is the number of intervals in the collection. Since a query may return all intervals, for example if the query is a large interval intersecting all intervals in the collection, this is asymptotically optimal; however, we can do better by considering output-sensitive algorithms, where the runtime is expressed in terms of  $m$ , the number of intervals produced by the query. Interval trees are dynamic, i.e., they allow insertion and deletion of intervals. They obtain a query time of  $O(\log n)$  while the preprocessing time to construct the data structure is  $O(n \log n)$  (but the space consumption is  $O(n)$ ). If the endpoints of intervals are within a small integer range (e.g., in the range  $[1, \dots, O(n)]$ ), faster data structures exist<sup>[1]</sup> with preprocessing time  $O(n)$  and query time  $O(1+m)$  for reporting  $m$  intervals containing a given query point.

### 10.4.1 Naive approach

In a simple case, the intervals do not overlap and they can be inserted into a simple **binary search tree** and queried in  $O(\log n)$  time. However, with arbitrarily overlapping intervals, there is no way to compare two intervals for insertion into the tree since orderings sorted by the beginning points or the ending points may be different. A naive approach might be to build two parallel trees, one ordered by the beginning point, and one ordered by the ending point

of each interval. This allows discarding half of each tree in  $O(\log n)$  time, but the results must be merged, requiring  $O(n)$  time. This gives us queries in  $O(n + \log n) = O(n)$ , which is no better than brute-force.

Interval trees solve this problem. This article describes two alternative designs for an interval tree, dubbed the *centered interval tree* and the *augmented tree*.

### 10.4.2 Centered interval tree

Queries require  $O(\log n + m)$  time, with  $n$  being the total number of intervals and  $m$  being the number of reported results. Construction requires  $O(n \log n)$  time, and storage requires  $O(n)$  space.

#### Construction

Given a set of  $n$  intervals on the number line, we want to construct a data structure so that we can efficiently retrieve all intervals overlapping another interval or point.

We start by taking the entire range of all the intervals and dividing it in half at  $x_{center}$  (in practice,  $x_{center}$  should be picked to keep the tree relatively balanced). This gives three sets of intervals, those completely to the left of  $x_{center}$  which we'll call  $S_{left}$ , those completely to the right of  $x_{center}$  which we'll call  $S_{right}$ , and those overlapping  $x_{center}$  which we'll call  $S_{center}$ .

The intervals in  $S_{left}$  and  $S_{right}$  are recursively divided in the same manner until there are no intervals left.

The intervals in  $S_{center}$  that overlap the center point are stored in a separate data structure linked to the node in the interval tree. This data structure consists of two lists, one containing all the intervals sorted by their beginning points, and another containing all the intervals sorted by their ending points.

The result is a ternary tree with each node storing:

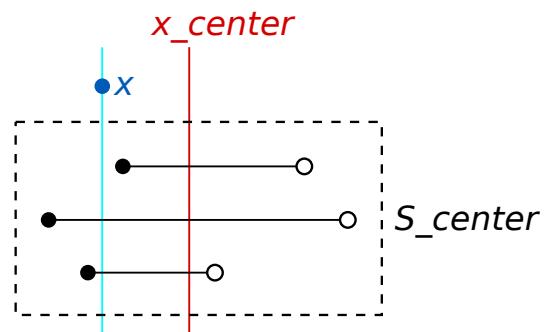
- A center point
- A pointer to another node containing all intervals completely to the left of the center point
- A pointer to another node containing all intervals completely to the right of the center point
- All intervals overlapping the center point sorted by their beginning point
- All intervals overlapping the center point sorted by their ending point

#### Intersecting

Given the data structure constructed above, we receive queries consisting of ranges or points, and return all the ranges in the original set overlapping this input.

**With a Point** The task is to find all intervals in the tree that overlap a given point  $x$ . The tree is walked with a similar recursive algorithm as would be used to traverse a traditional binary tree, but with extra affordance for the intervals overlapping the “center” point at each node.

For each tree node,  $x$  is compared to  $x_{center}$ , the midpoint used in node construction above. If  $x$  is less than  $x_{center}$ , the leftmost set of intervals,  $S_{left}$ , is considered. If  $x$  is greater than  $x_{center}$ , the rightmost set of intervals,  $S_{right}$ , is considered.



All intervals in  $S_{center}$  that begin before  $x$  must overlap  $x$  if  $x$  is less than  $x_{center}$ .

Similarly, the same technique also applies in checking a given interval. If a given interval ends at  $y$  and  $y$  is less than  $x_{center}$ , all intervals in  $S_{center}$  that begin before  $y$  must also overlap the given interval.

As each node is processed as we traverse the tree from the root to a leaf, the ranges in its  $S_{center}$  are processed. If  $x$  is less than  $x_{center}$ , we know that all intervals in  $S_{center}$  end after  $x$ , or they could not also overlap  $x_{center}$ . Therefore, we need only find those intervals in  $S_{center}$  that begin before  $x$ . We can consult the lists of  $S_{center}$  that have already been constructed. Since we only care about the interval beginnings in this scenario, we can consult the list sorted by beginnings. Suppose we find the closest number no greater than  $x$  in this list. All ranges from the beginning of the list to that found point overlap  $x$  because they begin before  $x$  and end after  $x$  (as we know because they overlap  $x_{center}$  which is larger than  $x$ ). Thus, we can simply start enumerating intervals in the list until the startpoint value exceeds  $x$ .

Likewise, if  $x$  is greater than  $x_{center}$ , we know that all intervals in  $S_{center}$  must begin before  $x$ , so we find those intervals that end after  $x$  using the list sorted by interval endings.

If  $x$  exactly matches  $x_{center}$ , all intervals in  $S_{center}$  can be added to the results without further processing and tree traversal can be stopped.

**With an Interval** For a result interval  $r$  to intersect our query interval  $q$  one of the following must hold: the start and/or end point of  $r$  is in  $q$ ; or  $r$  completely encloses  $q$ .

We first find all intervals with start and/or end points inside  $q$  using a separately constructed tree. In the one-dimensional case, we can use a search tree containing all the start and end points in the interval set, each with a pointer to its corresponding interval. A binary search in  $O(\log n)$  time for the start and end of  $q$  reveals the minimum and maximum points to consider. Each point within this range references an interval that overlaps  $q$  and is added to the result list. Care must be taken to avoid duplicates, since an interval might both begin and end within  $q$ . This can be done using a binary flag on each interval to mark whether or not it has been added to the result set.

Finally, we must find intervals that enclose  $q$ . To find these, we pick any point inside  $q$  and use the algorithm above to find all intervals intersecting that point (again, being careful to remove duplicates).

### Higher Dimensions

The interval tree data structure can be generalized to a higher dimension  $N$  with identical query and construction time and  $O(n \log n)$  space.

First, a [range tree](#) in  $N$  dimensions is constructed that allows efficient retrieval of all intervals with beginning and end points inside the query region  $R$ . Once the corresponding ranges are found, the only thing that is left are those ranges that enclose the region in some dimension. To find these overlaps,  $N$  interval trees are created, and one axis intersecting  $R$  is queried for each. For example, in two dimensions, the bottom of the square  $R$  (or any other horizontal line intersecting  $R$ ) would be queried against the interval tree constructed for the horizontal axis. Likewise, the left (or any other vertical line intersecting  $R$ ) would be queried against the interval tree constructed on the vertical axis.

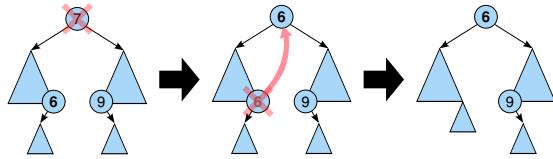
Each interval tree also needs an addition for higher dimensions. At each node we traverse in the tree,  $x$  is compared with  $S_{center}$  to find overlaps. Instead of two sorted lists of points as was used in the one-dimensional case, a range tree is constructed. This allows efficient retrieval of all points in  $S_{center}$  that overlap region  $R$ .

### Deletion

If after deleting an interval from the tree, the node containing that interval contains no more intervals, that node may be deleted from the tree. This is more complex than a normal binary tree deletion operation.

An interval may overlap the center point of several nodes in the tree. Since each node stores the intervals that overlap it, with all intervals completely to the left of its center point in the left subtree, similarly for the right subtree, it follows that each interval is stored in the node closest to the root from the set of nodes whose center point it overlaps.

Normal deletion operations in a binary tree (for the case where the node being deleted has two children) involve promoting a node further from the leaf to the position of the node being deleted (usually the leftmost child of the right subtree, or the rightmost child of the left subtree).



*Deleting a node with two children from a binary search tree using the in-order predecessor (rightmost node in the left subtree, labelled 6).*

As a result of this promotion, some nodes that were above the promoted node will become descendants of it; it is necessary to search these nodes for intervals that also overlap the promoted node, and move those intervals into the promoted node. As a consequence, this may result in new empty nodes, which must be deleted, following the same algorithm again.

### Balancing

The same issues that affect deletion also affect rotation operations; rotation must preserve the invariant that intervals are stored as close to the root as possible.

### 10.4.3 Augmented tree

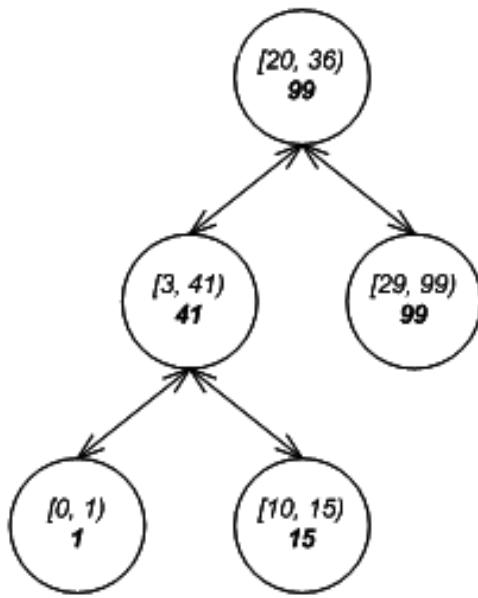
Another way to represent intervals is described in Cormen et al. (2009, Section 14.3: Interval trees, pp. 348–354).

Both insertion and deletion require  $O(\log n)$  time, with  $n$  being the total number of intervals in the tree prior to the insertion or deletion operation.

Use a simple ordered tree, for example a [binary search tree](#) or [self-balancing binary search tree](#), where the tree is ordered by the 'low' values of the intervals, and an extra annotation is added to every node recording the maximum high value among the tree: the node's high value and the high values of both its subtrees. It is simple to maintain this attribute in only  $O(h)$  steps during each addition or removal of a node, where  $h$  is the height of the node added or removed in the tree, by updating all ancestors of the node from the bottom up. Additionally, the [tree rotations](#) used during insertion and deletion may require updating the high value of the affected nodes.

Now, it is known that two intervals  $A$  and  $B$  overlap only when both  $A.\text{low} \leq B.\text{high}$  and  $A.\text{high} \geq B.\text{low}$ . When searching the trees for nodes overlapping with a given interval, you can immediately skip:

- all nodes to the right of nodes whose low value is



An augmented tree with low value as the key and maximum high as the extra annotation.

For example, when testing if the given interval  $I1=[-5, 2]$  overlaps the intervals in the tree shown above, we can immediately skip the right subtrees of nodes holding intervals  $I2=[20, 36]$  and  $I3=[3, 41]$  while traversing the tree. Since  $I2$ 's low value 20 and  $I3$ 's low value 3 are past the given interval  $I1$ 's end point 2, all the low values in the right subtrees in question must also be past  $I1$ 's end point and therefore can be skipped.

past the end of the given interval.

- all nodes that have their maximum 'high' value below the start of the given interval.

A total order can be defined on the intervals by ordering them first by their 'low' value and finally by their 'high' value. This ordering can be used to prevent duplicate intervals from being inserted into the tree in  $O(\log n)$  time, versus the  $O(k + \log n)$  time required to find duplicates if  $k$  intervals overlap a new interval.

### Java Example: Adding a new interval to the tree

The key of each node is the interval itself, hence nodes are ordered first by low value and finally by high value, and the value of each node is the end point of the interval:

```
public void add(Interval i) { put(i, i.getEnd()); }
```

### Java Example: Searching a point or an interval in the tree

To search for an interval, you walk the tree, omitting those branches which can't contain what you're looking

for. The simple case is looking for a point:

```
// Search for all intervals which contain "p", starting
// with the // node "n" and adding matching intervals
// to the list "result" public void search(IntervalNode
// n, Point p, List<Interval> result) { // Don't search
// nodes that don't exist if (n == null) return; // If
// p is to the right of the rightmost point of any
// interval // in this node and all children, there won't
// be any matches. if (p.compareTo(n.getValue()) >
// 0) return; // Search left children if (n.getLeft() !=
// null) search(IntervalNode (n.getLeft()), p, result);
// Check this node if (n.getKey().contains(p))
// result.add(n.getKey()); // If p is to the left of the start
// of this interval, // then it can't be in any child to the right.
if (p.compareTo(n.getKey().getStart()) < 0) return; // Otherwise, search right children if (n.getRight() !=
// null) search(IntervalNode (n.getRight()), p, result); }
```

where

```
a.compareTo(b) returns a negative value if a <
b
a.compareTo(b) returns zero if a = b
a.compareTo(b) returns a positive value if a >
b
```

The code to search for an interval is similar, except for the check in the middle:

```
// Check this node if (n.getKey().overlapsWith(i))
result.add (n.getKey());
```

`overlapsWith()` is defined as:

```
public boolean overlapsWith(Interval other) { return
start.compareTo(other.getEnd()) <= 0 &&
end.compareTo(other.getStart()) >= 0; }
```

### Higher dimension

This can be extended to higher dimensions by cycling through the dimensions at each level of the tree. For example, for two dimensions, the odd levels of the tree might contain ranges for the  $x$  coordinate, while the even levels contain ranges for the  $y$  coordinate. However, it is not quite obvious how the rotation logic will have to be extended for such cases to keep the tree balanced.

A much simpler solution is to use nested interval trees. First, create a tree using the ranges for the  $y$  coordinate. Now, for each node in the tree, add another interval tree on the  $x$  ranges, for all elements whose  $y$  range intersect that node's  $y$  range.

The advantage of this solution is that it can be extended to an arbitrary amount of dimensions using the same code base.

At first, the cost for the additional trees might seem prohibitive but that is usually not the case. As with the solution above, you need one node per  $x$  coordinate, so this cost is the same in both solutions. The only difference is that you need an additional tree structure per vertical interval. This structure is typically very small (a pointer to the root node plus maybe the number of nodes and the height of the tree).

#### 10.4.4 Medial- or length-oriented tree

A medial- or length-oriented tree is similar to an augmented tree, but symmetrical, with the binary search tree ordered by the medial points of the intervals. There is a maximum-oriented binary heap in every node, ordered by the length of the interval (or half of the length). Also we store the minimum and maximum possible value of the subtree in each node (thus the symmetry).

##### Overlap test

Using only start and end values of two intervals  $(a_i, b_i)$ , for  $i = 0, 1$ , the overlap test can be performed as follows:

$$a_0 \leq a_1 < b_0 \text{ OR } a_0 < b_1 \leq b_0 \text{ OR } a_1 \leq a_0 < b_1 \text{ OR } a_1 < b_0 \leq b_1$$

But with defining:

$$m_i = \frac{a_i + b_i}{2}$$

$$d_i = \frac{b_i - a_i}{2}$$

The overlap test is simpler:

$$|m_1 - m_0| < d_0 + d_1$$

##### Adding interval

Adding new intervals to the tree is the same as for a binary search tree using the medial value as the key. We push  $d_i$  onto the binary heap associated with the node, and update the minimum and maximum possible values associated with all higher nodes.

##### Searching for all overlapping intervals

Let's use  $a_q, b_q, m_q, d_q$  for the query interval, and  $M_n$  for the key of a node (compared to  $m_i$  of intervals)

Starting with root node, in each node, first we check if it is possible that our query interval overlaps with the node subtree using minimum and maximum values of node (if it is not, we don't continue for this node).

Then we calculate  $\min \{d_i\}$  for intervals inside this node (not its children) to overlap with query interval (knowing  $m_i = M_n$ ):

$$\min \{d_i\} = |m_q - M_n| - d_q$$

and perform a query on its binary heap for the  $d_i$ 's bigger than  $\min \{d_i\}$

Then we pass through both left and right children of the node, doing the same thing. In the worst-case, we have to scan all nodes of the binary search tree, but since binary heap query is optimum, this is acceptable (a 2-dimensional problem can not be optimum in both dimensions)

This algorithm is expected to be faster than a traditional interval tree (augmented tree) for search operations, adding elements is a little slower (the order of growth is the same).

#### 10.4.5 References

- [1] [http://en.wikipedia.org/wiki/Range\\_Queries#Semigroup\\_operators](http://en.wikipedia.org/wiki/Range_Queries#Semigroup_operators). Missing or empty |title= (help)
- Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry*, Second Revised Edition. Springer-Verlag 2000. Section 10.1: Interval Trees, pp. 212–217.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009), *Introduction to Algorithms* (3rd ed.), MIT Press and McGraw-Hill, ISBN 978-0-262-03384-8
- Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985
- Jens M. Schmidt. *Interval Stabbing Problems in Small Integer Ranges*. DOI. ISAAC'09, 2009

#### 10.4.6 External links

- CGAL : Computational Geometry Algorithms Library in C++ contains a robust implementation of Range Trees
- Interval Tree (an augmented self balancing avl tree implementation)
- Interval Tree (a ruby implementation)

## 10.5 Range tree

In computer science, a **range tree** is an ordered tree data structure to hold a list of points. It allows all points within a given range to be **reported** efficiently, and is typically used in two or higher dimensions. Range trees were introduced by Jon Louis Bentley in 1979.<sup>[1]</sup> Similar data structures were discovered independently by Lueker,<sup>[2]</sup> Lee and Wong,<sup>[3]</sup> and Willard.<sup>[4]</sup> The range tree is an alternative to the  $k$ -d tree. Compared to  $k$ -d trees, range trees offer faster query times of (in Big O notation)  $O(\log^d n + k)$

but worse storage of  $O(n \log^{d-1} n)$ , where  $n$  is the number of points stored in the tree,  $d$  is the dimension of each point and  $k$  is the number of points reported by a given query.

Bernard Chazelle improved this to query time  $O(\log^{d-1} n + k)$  and space complexity  $O\left(n \left(\frac{\log n}{\log \log n}\right)^{d-1}\right)$ .<sup>[5][6]</sup>

### 10.5.1 Description

A range tree on a set of 1-dimensional points is a balanced **binary search tree** on those points. The points stored in the tree are stored in the leaves of the tree; each internal node stores the largest value contained in its left subtree. A range tree on a set of points in  $d$ -dimensions is a **recursively defined multi-level binary search tree**. Each level of the data structure is a binary search tree on one of the  $d$ -dimensions. The first level is a binary search tree on the first of the  $d$ -coordinates. Each vertex  $v$  of this tree contains an associated structure that is a  $(d-1)$ -dimensional range tree on the last  $(d-1)$ -coordinates of the points stored in the subtree of  $v$ .

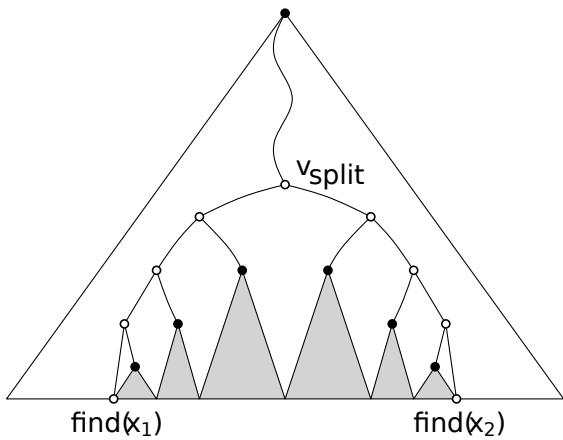
### 10.5.2 Operations

#### Construction

A 1-dimensional range tree on a set of  $n$  points is a binary search tree, which can be constructed in  $O(n \log n)$  time. Range trees in higher dimensions are constructed recursively by constructing a balanced binary search tree on the first coordinate of the points, and then, for each vertex  $v$  in this tree, constructing a  $(d-1)$ -dimensional range tree on the points contained in the subtree of  $v$ . Constructing a range tree this way would require  $O(n \log^d n)$  time.

This can be improved by noticing that a range tree on a set 2-dimensional points can be constructed in  $O(n \log n)$  time.<sup>[7]</sup> Let  $S$  be a set of  $n$  2-dimensional points. If  $S$  contains only one point, return a leaf containing that point. Otherwise, construct the associated structure of  $S$ , a 1-dimensional range tree on the  $y$ -coordinates of the points in  $S$ . Let  $x_m$  be the median  $x$ -coordinate of the points. Let  $SL$  be the set of points with  $x$ -coordinate less than or equal to  $x_m$  and let  $SR$  be the set of points with  $x$ -coordinate greater than  $x_m$ . Recursively construct  $vL$ , a 2-dimensional range tree on  $SL$ , and  $vR$ , a 2-dimensional range tree on  $SR$ . Create a vertex  $v$  with left-child  $vL$  and right-child  $vR$ . If we sort the points by their  $y$ -coordinates at the start of the algorithm, and maintain this ordering when splitting the points by their  $x$ -coordinate, we can construct the associated structures of each subtree in linear time. This reduces the time to construct a 2-dimensional range tree to  $O(n \log n)$ , which also reduces the time to construct a  $d$ -dimensional range tree to  $O(n \log^{d-1} n)$ .

#### Range queries



A 1-dimensional range query  $[x_1, x_2]$ . Points stored in the subtrees shaded in gray will be reported.  $find(x_1)$  and  $find(x_2)$  will be reported if they are inside the query interval.

Range trees can be used to find the set of points that lie inside a given interval. To report the points that lie in the interval  $[x_1, x_2]$ , we start by searching for  $x_1$  and  $x_2$ . At some vertex in the tree, the search paths to  $x_1$  and  $x_2$  will diverge. Let  $v_{\text{split}}$  be the last vertex that these two search paths have in common. Continue searching for  $x_1$  in the range tree. For every vertex  $v$  in the search path from  $v_{\text{split}}$  to  $x_1$ , if the value stored at  $v$  is greater than  $x_1$ , report every point in the right-subtree of  $v$ . If  $v$  is a leaf, report the value stored at  $v$  if it is inside the query interval. Similarly, reporting all of the points stored in the left-subtrees of the vertices with values less than  $x_2$  along the search path from  $v_{\text{split}}$  to  $x_2$ , and report the leaf of this path if it lies within the query interval.

Since the range tree is a balanced binary tree, the search paths to  $x_1$  and  $x_2$  have length  $O(\log n)$ . Reporting all of the points stored in the subtree of a vertex can be done in linear time using any **tree traversal** algorithm. It follows that the time to perform a range query is  $O(\log n + k)$ , where  $k$  is the number of points in the query interval.

Range queries in  $d$ -dimensions are similar. Instead of reporting all of the points stored in the subtrees of the search paths, perform a  $(d-1)$ -dimensional range query on the associated structure of each subtree. Eventually, a 1-dimensional range query will be performed and the correct points will be reported. Since a  $d$ -dimensional query consists of  $O(\log n)$   $(d-1)$ -dimensional range queries, it follows that the time required to perform a  $d$ -dimensional range query is  $O(\log^d n + k)$ , where  $k$  is the number of points in the query interval. This can be reduced to  $O(\log^{d-1} n + k)$  using the technique of **fractional cascading**.<sup>[2][4][7]</sup>

### 10.5.3 See also

- **$k$ -d tree**

- Segment tree

#### 10.5.4 References

- [1] Bentley, J. L. (1979). “Decomposable searching problems”. *Information Processing Letters* **8** (5): 244–251. doi:10.1016/0020-0190(79)90117-0.
- [2] Lueker, G. S. (1978). “A data structure for orthogonal range queries”. *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. pp. 28–21. doi:10.1109/SFCS.1978.1.
- [3] Lee, D. T.; Wong, C. K. (1980). “Quintary trees: A file structure for multidimensional database systems”. *ACM Transactions on Database Systems* **5** (3): 339. doi:10.1145/320613.320618.
- [4] Willard, Dan E. *The super-b-tree algorithm* (Technical report). Cambridge, MA: Aiken Computer Lab, Harvard University. TR-03-79.
- [5] Chazelle, Bernard (1990). “Lower Bounds for Orthogonal Range Searching: I. The Reporting Case” (PDF). *ACM* **37**: 200–212.
- [6] Chazelle, Bernard (1990). “Lower Bounds for Orthogonal Range Searching: II. The Arithmetic Model” (PDF). *ACM* **37**: 439–463.
- [7] de Berg, Mark; Cheong, Otfried; van Kreveld, Marc; Overmars, Mark (2008). *Computational Geometry*. doi:10.1007/978-3-540-77974-2. ISBN 978-3-540-77973-5.

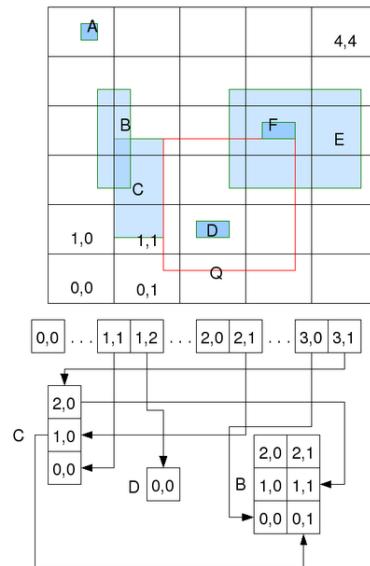
#### 10.5.5 External links

- Range and Segment Trees in CGAL, the Computational Geometry Algorithms Library.
- Lecture 8: Range Trees, Marc van Kreveld.

## 10.6 Bin

In computational geometry, the **bin data structure** allows efficient region queries, i.e., if there are some axis-aligned rectangles on a 2D plane, answer the question *Given a query rectangle, return all rectangles intersecting it.* **kd-tree** is another data structure that can answer this question efficiently. In the example in the figure, *A*, *B*, *C*, *D*, *E*, and *F* are existing rectangles, the query with the rectangle *Q* should return *C*, *D*, *E* and *F*, if we define all rectangles as closed intervals.

The data structure partitions a region of the 2D plane into uniform-sized *bins*. The bounding box of the bins encloses all *candidate* rectangles to be queried. All the bins are arranged in a 2D array. All the candidates are represented also as 2D arrays. The size of a candidate's array is the number of bins it intersects. For example, in the



The bin data structure

figure, candidate *B* has 6 elements arranged in a 3 row by 2 column array because it intersects 6 bins in such an arrangement. Each bin contains the head of a singly linked list. If a candidate intersects a bin, it is chained to the bin's linked list. Each element in a candidate's array is a link node in the corresponding bin's linked list.

#### 10.6.1 Operations

##### Query

From the query rectangle *Q*, we can find out which bin its lower-left corner intersects efficiently by simply subtracting the bin's bounding box's lower-left corner from the lower-left corner of *Q* and dividing the result by the width and height of a bin respectively. We then iterate the bins *Q* intersects and examine all the candidates in the linked-lists of these bins. For each candidate we check if it does indeed intersect *Q*. If so and it is not previously reported, then we report it. We can use the convention that we only report a candidate the first time we find it. This can be done easily by clipping the candidate against the query rectangle and comparing its lower-left corner against the current location. If it is a match then we report, otherwise we skip.

##### Insertion and deletion

Insertion is linear to the number of bins a candidate intersects because inserting a candidate into 1 bin is constant time. Deletion is more expensive because we need

to search the singly linked list of each bin the candidate intersects.

In a multithread environment, insert, delete and query are mutually exclusive. However, instead of locking the whole data structure, a sub-range of bins may be locked. Detailed performance analysis should be done to justify the overhead.

### 10.6.2 Efficiency and tuning

The analysis is similar to a [hash table](#). The worst-case scenario is that all candidates are concentrated in one bin. Then query is  $O(n)$ , delete is  $O(n)$ , and insert is  $O(1)$ , where  $n$  is the number of candidates. If the candidates are evenly spaced so that each bin has a constant number of candidates, The query is  $O(k)$  where  $k$  is the number of bins the query rectangle intersects. Insert and delete are  $O(m)$  where  $m$  is the number of bins the inserting candidate intersects. In practice delete is much slower than insert.

Like a hash table, bin's efficiency depends a lot on the distribution of both location and size of candidates and queries. In general, the smaller the query rectangle, the more efficient the query. The bin's size should be such that it contains as few candidates as possible but large enough so that candidates do not span too many bins. If a candidate span many bins, a query has to skip this candidate over and over again after it is reported at the first bin of intersection. For example, in the figure,  $E$  is visited 4 times in the query of  $Q$  and so has to be skipped 3 times.

To further speed up the query, divisions can be replaced by right shifts. This requires the number of bins along an axis direction to be an exponent of 2.

### 10.6.3 Compared to other range query data structures

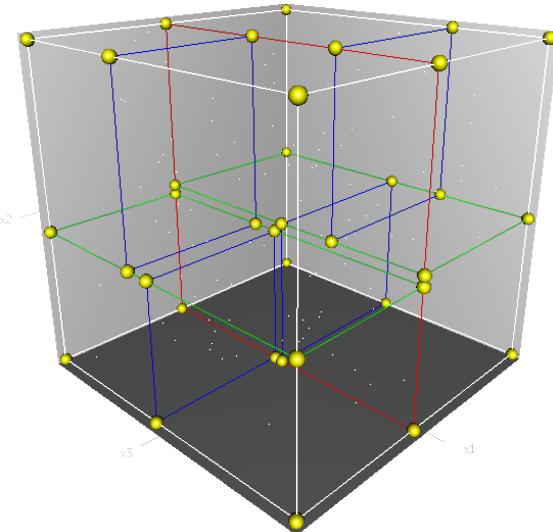
Against [kd-tree](#), the bin structure allows efficient insertion and deletion without the complexity of rebalancing. This can be very useful in algorithms that need to incrementally add shapes to the search data structure.

### 10.6.4 See also

- [kd-tree](#) is another efficient range query data structure.
- Space partitioning

## 10.7 k-d tree

In computer science, a **k-d tree** (short for *k-dimensional tree*) is a [space-partitioning data structure](#) for organizing



A 3-dimensional k-d tree. The first split (red) cuts the root cell (white) into two subcells, each of which is then split (green) into two subcells. Finally, each of those four is split (blue) into two subcells. Since there is no more splitting, the final eight are called leaf cells.

points in a  $k$ -dimensional space. k-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). k-d trees are a special case of binary space partitioning trees.

### 10.7.1 Informal description

The k-d tree is a [binary tree](#) in which every node is a  $k$ -dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting [hyperplane](#) that divides the space into two parts, known as [half-spaces](#). Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the  $k$ -dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x-value of the point, and its [normal](#) would be the unit x-axis.<sup>[1]</sup>

### 10.7.2 Operations on k-d trees

#### Construction

Since there are many possible ways to choose axis-aligned splitting planes, there are many different ways to construct k-d trees. The canonical method of k-d tree construction has the following constraints:<sup>[2]</sup>

- As one moves down the tree, one cycles through the axes used to select the splitting planes. (For example, in a 3-dimensional tree, the root would have an  $x$ -aligned plane, the root's children would both have  $y$ -aligned planes, the root's grandchildren would all have  $z$ -aligned planes, the root's great-grandchildren would all have  $x$ -aligned planes, the root's great-great-grandchildren would all have  $y$ -aligned planes, and so on.)
- Points are inserted by selecting the **median** of the points being put into the **subtree**, with respect to their coordinates in the axis being used to create the splitting plane. (Note the assumption that we feed the entire set of  $n$  points into the algorithm up-front.)

This method leads to a balanced  $k$ -d tree, in which each leaf node is about the same distance from the root. However, balanced trees are not necessarily optimal for all applications.

Note also that it is not *required* to select the median point. In that case, the result is simply that there is no guarantee that the tree will be balanced. A simple heuristic to avoid either coding a complex  $O(n)$  median-finding algorithm,<sup>[3][4]</sup> or using an  $O(n \log n)$  sort such as Heapsort or Mergesort to sort all  $n$  points, is to use the sort to find the median of a *fixed* number of *randomly* selected points to serve as the splitting plane. In practice, this technique often results in nicely balanced trees.

Given a list of  $n$  points, the following algorithm uses a median-finding sort to construct a balanced  $k$ -d tree containing those points.

```
function kdTree (list of points pointList, int depth) { // Select axis based on depth so that axis cycles through all valid values
 var int axis := depth mod k; // Sort point list and choose median as pivot element
 select median by axis
 from pointList; // Create node and construct subtrees
 var tree_node node; node.location := median; node.leftChild := kdTree(points in pointList before median, depth+1);
 node.rightChild := kdTree(points in pointList after median, depth+1); return node; }
```

It is common that points “after” the median include only the ones that are strictly greater than the median. For points that lie on the median, it is possible to define a “superkey” function that compares the points in all dimensions. In some cases, it is acceptable to let points equal to the median lie on one side of the median, for example, by splitting the points into a “lesser than” subset and a “greater than or equal to” subset.

This algorithm creates the invariant that for any node, all the nodes in the left **subtree** are on one side of a splitting **plane**, and all the nodes in the right subtree are on the other side. Points that lie on the splitting plane may appear on either side. The splitting plane of a node goes through the point associated with that node (referred to in the code as *node.location*).

Alternative algorithms for building a balanced  $k$ -d tree presort the data prior to building the tree. They then maintain the order of the presort during tree construction and hence eliminate the costly step of finding the median at each level of subdivision. Two such algorithms build a balanced  $k$ -d tree to sort triangles in order to improve the execution time of **ray tracing** for three-dimensional computer graphics. These algorithms presort  $n$  triangles prior to building the  $k$ -d tree, then build the tree in  $O(n \log n)$  time in the best case.<sup>[5][6]</sup> An algorithm that builds a balanced  $k$ -d tree to sort points has a worst-case complexity of  $O(kn \log n)$ .<sup>[7]</sup> This algorithm presorts  $n$  points in each of  $k$  dimensions using an  $O(n \log n)$  sort such as Heapsort or Mergesort prior to building the tree. It then maintains the order of these  $k$  presorts during tree construction and thereby avoids finding the median at each level of subdivision.

### Adding elements

One adds a new point to a  $k$ -d tree in the same way as one adds an element to any other search tree. First, traverse the tree, starting from the root and moving to either the left or the right child depending on whether the point to be inserted is on the “left” or “right” side of the splitting plane. Once you get to the node under which the child should be located, add the new point as either the left or right child of the leaf node, again depending on which side of the node's splitting plane contains the new node.

Adding points in this manner can cause the tree to become unbalanced, leading to decreased tree performance. The rate of tree performance degradation is dependent upon the spatial distribution of tree points being added, and the number of points added in relation to the tree size. If a tree becomes too unbalanced, it may need to be rebalanced to restore the performance of queries that rely on the tree balancing, such as nearest neighbour searching.

### Removing elements

To remove a point from an existing  $k$ -d tree, without breaking the invariant, the easiest way is to form the set of all nodes and leaves from the children of the target node, and recreate that part of the tree.

Another approach is to find a replacement for the point removed.<sup>[8]</sup> First, find the node  $R$  that contains the point to be removed. For the base case where  $R$  is a leaf node, no replacement is required. For the general case, find a replacement point, say  $p$ , from the subtree rooted at  $R$ . Replace the point stored at  $R$  with  $p$ . Then, recursively remove  $p$ .

For finding a replacement point, if  $R$  discriminates on  $x$  (say) and  $R$  has a right child, find the point with the minimum  $x$  value from the subtree rooted at the right child. Otherwise, find the point with the maximum  $x$  value from

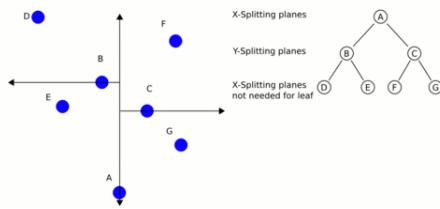
the subtree rooted at the left child.

## Balancing

Balancing a  $k$ -d tree requires care because  $k$ -d trees are sorted in multiple dimensions so the **tree rotation** technique cannot be used to balance them as this may break the invariant.

Several variants of balanced  $k$ -d trees exist. They include divided  $k$ -d tree, pseudo  $k$ -d tree,  $k$ -d B-tree, hB-tree and Bkd-tree. Many of these variants are adaptive  $k$ -d trees.

## Nearest neighbour search



Animation of NN searching with a  $k$ -d tree in two dimensions

The **nearest neighbour search** (NN) algorithm aims to find the point in the tree that is nearest to a given input point. This search can be done efficiently by using the tree properties to quickly eliminate large portions of the search space.

Searching for a nearest neighbour in a  $k$ -d tree proceeds as follows:

1. Starting with the root node, the algorithm moves down the tree recursively, in the same way that it would if the search point were being inserted (i.e. it goes left or right depending on whether the point is lesser than or greater than the current node in the split dimension).
2. Once the algorithm reaches a leaf node, it saves that node point as the “current best”
3. The algorithm unwinds the recursion of the tree, performing the following steps at each node:
  - (a) If the current node is closer than the current best, then it becomes the current best.
  - (b) The algorithm checks whether there could be any points on the other side of the splitting plane that are closer to the search point than the current best. In concept, this is done by intersecting the splitting **hyperplane** with a **hypersphere** around the search point that has a radius equal to the current nearest distance. Since the hyperplanes are all axis-aligned this is implemented as a simple comparison to

see whether the difference between the splitting coordinate of the search point and current node is lesser than the distance (overall coordinates) from the search point to the current best.

- i. If the hypersphere crosses the plane, there could be nearer points on the other side of the plane, so the algorithm must move down the other branch of the tree from the current node looking for closer points, following the same recursive process as the entire search.
- ii. If the hypersphere doesn't intersect the splitting plane, then the algorithm continues walking up the tree, and the entire branch on the other side of that node is eliminated.
4. When the algorithm finishes this process for the root node, then the search is complete.

Generally the algorithm uses squared distances for comparison to avoid computing square roots. Additionally, it can save computation by holding the squared current best distance in a variable for comparison.

Finding the nearest point is an  $O(\log n)$  operation in the case of randomly distributed points, although analysis in general is tricky. However an algorithm has been given that claims guaranteed  $O(\log n)$  complexity.<sup>[9]</sup>

In high-dimensional spaces, the **curse of dimensionality** causes the algorithm to need to visit many more branches than in lower-dimensional spaces. In particular, when the number of points is only slightly higher than the number of dimensions, the algorithm is only slightly better than a linear search of all of the points.

The algorithm can be extended in several ways by simple modifications. It can provide the  $k$  nearest neighbours to a point by maintaining  $k$  current bests instead of just one. A branch is only eliminated when  $k$  points have been found and the branch cannot have points closer than any of the  $k$  current bests.

It can also be converted to an approximation algorithm to run faster. For example, approximate nearest neighbour searching can be achieved by simply setting an upper bound on the number points to examine in the tree, or by interrupting the search process based upon a real time clock (which may be more appropriate in hardware implementations). Nearest neighbour for points that are in the tree already can be achieved by not updating the refinement for nodes that give zero distance as the result, this has the downside of discarding points that are not unique, but are co-located with the original search point.

Approximate nearest neighbour is useful in real-time applications such as robotics due to the significant speed increase gained by not searching for the best point exhaustively. One of its implementations is **best-bin-first**

search.

### Range search

A range search searches for ranges of parameters. For example, if a tree is storing values corresponding to income and age, then a range search might be something like looking for all members of the tree which have an age between 20 and 50 years and an income between 50,000 and 80,000. Since k-d trees divide the range of a domain in half at each level of the tree, they are useful for performing range searches.

Analyses of binary search trees has found that the worst case time for range search in a k-dimensional KD tree containing  $N$  nodes is given by the following equation.<sup>[10]</sup>

$$t_{\text{worst}} = O(k \cdot N^{1-\frac{1}{k}})$$

### 10.7.3 High-dimensional data

$k$ -d trees are not suitable for efficiently finding the nearest neighbour in high-dimensional spaces. As a general rule, if the dimensionality is  $k$ , the number of points in the data,  $N$ , should be  $N \gg 2^k$ . Otherwise, when  $k$ -d trees are used with high-dimensional data, most of the points in the tree will be evaluated and the efficiency is no better than exhaustive search,<sup>[11]</sup> and approximate nearest-neighbour methods should be used instead.

### 10.7.4 Complexity

- Building a static  $k$ -d tree from  $n$  points has the following worst-case complexity:
  - $O(n \log^2 n)$  if an  $O(n \log n)$  sort such as **Heapsort** or **Mergesort** is used to find the median at each level of the nascent tree;
  - $O(n \log n)$  if an  $O(n)$  median of medians algorithm<sup>[3][4]</sup> is used to select the median at each level of the nascent tree;
  - $O(kn \log n)$  if  $n$  points are presorted in each of  $k$  dimensions using an  $O(n \log n)$  sort such as **Heapsort** or **Mergesort** prior to building the  $k$ -d tree.<sup>[7]</sup>
- Inserting a new point into a balanced  $k$ -d tree takes  $O(\log n)$  time.
- Removing a point from a balanced  $k$ -d tree takes  $O(\log n)$  time.
- Querying an axis-parallel range in a balanced  $k$ -d tree takes  $O(n^{1-1/k} + m)$  time, where  $m$  is the number of the reported points, and  $k$  the dimension of the  $k$ -d tree.

- Finding 1 nearest neighbour in a balanced  $k$ -d tree with randomly distributed points takes  $O(\log n)$  time on average.

### 10.7.5 Variations

#### Volumetric objects

Instead of points, a  $k$ -d tree can also contain rectangles or hyperrectangles.<sup>[12][13]</sup> Thus range search becomes the problem of returning all rectangles intersecting the search rectangle. The tree is constructed the usual way with all the rectangles at the leaves. In an orthogonal range search, the *opposite* coordinate is used when comparing against the median. For example, if the current level is split along  $x_{\text{high}}$ , we check the  $x_{\text{low}}$  coordinate of the search rectangle. If the median is less than the  $x_{\text{low}}$  coordinate of the search rectangle, then no rectangle in the left branch can ever intersect with the search rectangle and so can be pruned. Otherwise both branches should be traversed. See also **interval tree**, which is a 1-dimensional special case.

#### Points only in leaves

It is also possible to define a  $k$ -d tree with points stored solely in leaves.<sup>[2]</sup> This form of  $k$ -d tree allows a variety of split mechanics other than the standard median split. The midpoint splitting rule<sup>[14]</sup> selects on the middle of the longest axis of the space being searched, regardless of the distribution of points. This guarantees that the aspect ratio will be at most 2:1, but the depth is dependent on the distribution of points. A variation, called sliding-midpoint, only splits on the middle if there are points on both sides of the split. Otherwise, it splits on point nearest to the middle. Maneewongvatana and Mount show that this offers “good enough” performance on common data sets. Using sliding-midpoint, an approximate nearest neighbour query can be answered in  $O(\frac{1}{\epsilon^d} \log n)$ . Approximate range counting can be answered in  $O(\log n + (\frac{1}{\epsilon})^d)$  with this method.

### 10.7.6 See also

- **implicit  $k$ -d tree**, a  $k$ -d tree defined by an implicit splitting function rather than an explicitly-stored set of splits
- **min/max  $k$ -d tree**, a  $k$ -d tree that associates a minimum and maximum value with each of its nodes
- **Ntrypy**, computer library for the rapid development of algorithms that uses a kd-tree for running on a parallel computer
- **Octree**, a higher-dimensional generalization of a quadtree

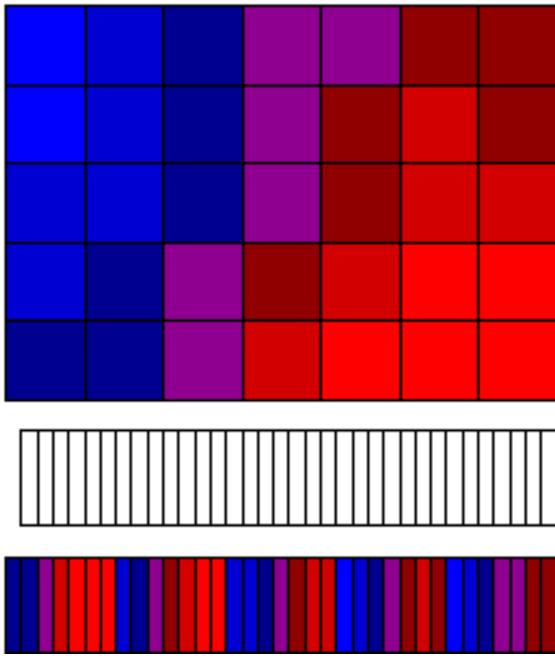
- Quadtree, a space-partitioning structure that splits at the geometric midpoint rather than the median coordinate
  - R-tree and bounding interval hierarchy, structure for partitioning objects rather than points, with overlapping regions
  - Recursive partitioning, a technique for constructing statistical decision trees that are similar to  $k$ -d trees
  - Klee's measure problem, a problem of computing the area of a union of rectangles, solvable using  $k$ -d trees
  - Guillotine problem, a problem of finding a  $k$ -d tree whose cells are large enough to contain a given set of rectangles
  - Ball tree, a multi-dimensional space partitioning useful for nearest neighbor search
- [10] Lee, D. T.; Wong, C. K. (1977). "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees". *Acta Informatica* **9**. doi:10.1007/BF00263763.
- [11] Jacob E. Goodman, Joseph O'Rourke and Piotr Indyk (Ed.) (2004). "Chapter 39 : Nearest neighbours in high-dimensional spaces". *Handbook of Discrete and Computational Geometry* (2nd ed.). CRC Press.
- [12] Rosenberg, J. B. (1985). "Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **4**: 53. doi:10.1109/TCAD.1985.1270098.
- [13] Houthuys, P. (1987). "Box Sort, a multidimensional binary sorting method for rectangular boxes, used for quick range searching". *The Visual Computer* **3** (4): 236. doi:10.1007/BF01952830.
- [14] S. Maneewongvatana and D. M. Mount. It's okay to be skinny, if your friends are fat. 4th Annual CGC Workshop on Computational Geometry, 1999.

### 10.7.7 References

- [1] Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching". *Communications of the ACM* **18** (9): 509. doi:10.1145/361002.361007.
- [2] "Orthogonal Range Searching". *Computational Geometry*. 2008. p. 95. doi:10.1007/978-3-540-77974-2\_5. ISBN 978-3-540-77973-5.
- [3] Blum, M.; Floyd, R. W.; Pratt, V. R.; Rivest, R. L.; Tarjan, R. E. (August 1973). "Time bounds for selection" (PDF). *Journal of Computer and System Sciences* **7** (4): 448–461. doi:10.1016/S0022-0000(73)80033-9.
- [4] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L.. *Introduction to Algorithms*. MIT Press and McGraw-Hill. Chapter 10.
- [5] Wald I, Havran V (September 2006). "On building fast kd-trees for ray tracing, and on doing that in  $O(N \log N)$ " (PDF). In: *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*: 61–69. doi:10.1109/RT.2006.280216.
- [6] Havran V, Bittner J (2002). "On improving k-d trees for ray shooting" (PDF). In: *Proceedings of the WSCG*: 209–216.
- [7] Brown RA (2015). "Building a balanced k-d tree in  $O(kn \log n)$  time". *Journal of Computer Graphics Techniques* **4** (1): 50–68.
- [8] Chandran, Sharat. *Introduction to kd-trees*. University of Maryland Department of Computer Science.
- [9] Friedman, J. H.; Bentley, J. L.; Finkel, R. A. (1977). "An Algorithm for Finding Best Matches in Logarithmic Expected Time". *ACM Transactions on Mathematical Software* **3** (3): 209. doi:10.1145/355744.355745.

### 10.7.8 External links

- libkd-tree++, an open-source STL-like implementation of  $k$ -d trees in C++.
- A tutorial on KD Trees
- FLANN and its fork nanoflann, efficient C++ implementations of  $k$ -d tree algorithms.
- Spatial C++ Library, a generic implementation of  $k$ -d tree as multi-dimensional containers, algorithms, in C++.
- kd-tree A simple C library for working with KD-Trees
- K-D Tree Demo, Java applet
- libANN Approximate Nearest Neighbour Library includes a  $k$ -d tree implementation
- Caltech Large Scale Image Search Toolbox: a Matlab toolbox implementing randomized  $k$ -d tree for fast approximate nearest neighbour search, in addition to LSH, Hierarchical K-Means, and Inverted File search algorithms.
- Heuristic Ray Shooting Algorithms, pp. 11 and after
- Into contains open source implementations of exact and approximate (k)NN search methods using  $k$ -d trees in C++.
- Math::Vector::Real::kdTree Perl implementation of  $k$ -d trees.



*Construction and storage of a 2D implicit max kd-tree using the grid median splitting-function. Each cell of the rectilinear grid has one scalar value from low (bright blue) to high (bright red) assigned to it. The grid's memory footprint is indicated in the lower line. The implicit max kd-tree's predefined memory footprint needs one scalar value less than that. The storing of the node's max values is indicated in the upper line.*

## 10.8 Implicit k-d tree

An **implicit k-d tree** is a **k-d tree** defined implicitly above a **rectilinear grid**. Its **split planes' positions** and **orientations** are not given explicitly but implicitly by some recursive splitting-function defined on the **hyperrectangles** belonging to the tree's **nodes**. Each inner node's split plane is positioned on a grid plane of the underlying grid, partitioning the node's grid into two subgrids.

### 10.8.1 Nomenclature and references

The terms "min/max **k-d tree**" and "implicit **k-d tree**" are sometimes mixed up. This is because the first publication using the term "implicit **k-d tree**" [1] did actually use explicit min/max **k-d trees** but referred to them as "implicit **k-d trees**" to indicate that they may be used to ray trace implicitly given iso surfaces. Nevertheless this publication used also slim **k-d trees** which are a subset of the implicit **k-d trees** with the restriction that they can only be built over integer hyperrectangles with sidelengths that are powers of two. Implicit **k-d trees** as defined here have recently been introduced, with applications in computer graphics.<sup>[2][3]</sup> As it is possible to assign attributes to implicit **k-d tree nodes**, one may refer to an implicit **k-d tree** which has min/max values assigned to its nodes as an "implicit min/max **k-d tree**".

### 10.8.2 Construction

Implicit **k-d trees** are in general not constructed explicitly. When accessing a node, its split plane orientation and position are evaluated using the specific splitting-function defining the tree. Different splitting-functions may result in different trees for the same underlying grid.

#### Splitting-functions

Splitting-functions may be adapted to special purposes. Underneath two specifications of special splitting-function classes.

- **Non-degenerated splitting-functions** do not allow the creation of degenerated nodes (nodes whose corresponding integer hyperrectangle's volume is equal zero). Their corresponding implicit **k-d trees** are **full binary trees**, which have for  $n$  leaf nodes  $n - 1$  inner nodes. Their corresponding implicit **k-d trees** are **non-degenerated implicit k-d trees**.
- **complete splitting-functions** are non-degenerated splitting-functions whose corresponding implicit **k-d tree**'s leaf nodes are single grid cells such that they have one inner node less than the amount of gridcells given in the grid. The corresponding implicit **k-d trees** are **complete implicit k-d trees**.

A complete splitting function is for example the **grid median splitting-function**. It creates fairly balanced implicit **k-d trees** by using  $k$ -dimensional integer hyperrectangles  $hyprec[2]/[k]$  belonging to each node of the implicit **k-d tree**. The hyperrectangles define which gridcells of the rectilinear grid belong to their corresponding node. If the volume of this hyperrectangle equals one, the corresponding node is a single grid cell and is therefore not further subdivided and marked as leaf node. Otherwise the hyperrectangle's longest extend is chosen as orientation  $o$ . The corresponding split plane  $p$  is positioned onto the grid plane that is closest to the hyperrectangle's grid median along that orientation.

Split plane orientation  $o$ :

$$o = \min\{\operatorname{argmax}(i = 1 \dots k: (hyprec[1][i] - hyprec[0][i]))\}$$

Split plane position  $p$ :

$$p = \operatorname{roundDown}((hyprec[0][o] + hyprec[1][o]) / 2)$$

#### Assigning attributes to implicit **k-d tree nodes**

An obvious advantage of implicit **k-d trees** is that their split plane's orientations and positions need not to be stored explicitly.

But some applications require besides the split plane's orientations and positions further attributes at the inner tree

nodes. These attributes may be for example single bits or single scalar values, defining if the subgrids belonging to the nodes are of interest or not. For complete implicit  $k$ -d trees it is possible to pre-allocate a correctly sized array of attributes and to assign each inner node of the tree to a unique element in that allocated array.

The amount of gridcells in the grid is equal the volume of the integer hyperrectangle belonging to the grid. As a complete implicit  $k$ -d tree has one inner node less than grid cells, it is known in advance how many attributes need to be stored. The relation "*Volume of integer hyperrectangle to inner nodes*" defines together with the complete splitting-function a recursive formula assigning to each split plane a unique element in the allocated array. The corresponding algorithm is given in C-pseudo code underneath.

```
// Assigning attributes to inner nodes of a complete
// implicit k-d tree // create an integer help hyperrectangle
hyprec (its volume vol(hyprec) is equal the
amount of leaves) int hyprec[2][k] = { { 0, ..., 0 }, { length_1, ..., length_k } }; // allocate once the array
of attributes for the entire implicit k-d tree attr *a =
new attr[volume(hyprec) - 1]; attr implicitKdTreeAttributes(int hyprec[2][k], attr *a) { if(vol(hyprec) >
1) // the current node is an inner node { // evaluate
the split plane's orientation o and its position p using
the underlying complete split-function int o, p;
completeSplittingFunction(hyprec, &o, &p); // evaluate the
children's integer hyperrectangles hyprec_l and hyprec_r
int hyprec_l[2][k], hyprec_r[2][k]; hyprec_l = hyprec;
hyprec_l[1][o] = p; hyprec_r = hyprec; hyprec_r[0][o]
= p; // evaluate the children's memory location a_l and
a_r attr* a_l = a + 1; attr* a_r = a + vol(hyprec_l); // evaluate
recursively the children's attributes c_l and c_r
attr c_l = implicitKdTreeAttributes(hyprec_l, a_l); attr
c_r = implicitKdTreeAttributes(hyprec_r, a_r); // merge
the children's attributes to the current attribute c attr c =
merge(c_l, c_r); // store the current attribute and return
it a[0] = c; return c; } // The current node is a leaf node.
Return the attribute belonging to the corresponding
gridcell return attribute(hyprec); }
```

It is worth mentioning that this algorithm works for all rectilinear grids. The corresponding integer hyperrectangle does not necessarily have to have sidelengths that are powers of two.

### 10.8.3 Applications

Implicit  $\text{max-}k$ -d trees are used for ray casting isosurfaces/MIP (maximum intensity projection). The attribute assigned to each inner node is the maximal scalar value given in the subgrid belonging to the node. Nodes are not traversed if their scalar values are smaller than the searched iso-value/current maximum intensity along the ray. The low storage requirements of the

implicit  $\text{max}$   $k$ -d tree and the favorable visualization complexity of ray casting allow to ray cast (and even change the isosurface for) very large scalar fields at interactive framerates on commodity PCs. Similarly an implicit  $\text{min/max}$   $k$ -d tree may be used to efficiently evaluate queries such as terrain line of sight.<sup>[4]</sup>

### 10.8.4 Complexity

Given an implicit  $k$ -d tree spanned over an  $k$ -dimensional grid with  $n$  gridcells.

- Assigning attributes to the nodes of the tree takes  $O(kn)$  time.
- Storing attributes to the nodes takes  $O(n)$  memory.
- Ray casting iso-surfaces/MIP an underlying scalar field using the corresponding implicit  $\text{max}$   $k$ -d tree takes roughly  $O(\log(n))$  time.

### 10.8.5 See also

- $k$ -d tree
- min/max  $k$ -d tree

### 10.8.6 References

- [1] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek and Hans-Peter Seidel "Faster Isosurface Ray Tracing using Implicit KD-Trees" IEEE Transactions on Visualization and Computer Graphics (2005)
- [2] Matthias Groß, Carsten Lojewski, Martin Bertram and Hans Hagen "Fast Implicit  $k$ -d Trees: Accelerated Isosurface Ray Tracing and Maximum Intensity Projection for Large Scalar Fields" CGIM07: Proceedings of Computer Graphics and Imaging (2007) 67-74
- [3] Matthias Groß (PhD, 2009) Towards Scientific Applications for Interactive Ray Casting
- [4] Bernardt Duvenhage "Using An Implicit Min/Max KD-Tree for Doing Efficient Terrain Line of Sight Calculations" in "Proceedings of the 6th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa", 2009.

## 10.9 min/max kd-tree

A **min/max kd-tree** is a  $k$ -d tree with two scalar values - a minimum and a maximum - assigned to its nodes. The minimum/maximum of an inner node is equal to the minimum/maximum of its children's minima/maxima.

### 10.9.1 Construction

Min/max *kd*-trees may be constructed recursively. Starting with the root node, the splitting plane orientation and position is evaluated. Then the children's splitting planes and min/max values are evaluated recursively. The min/max value of the current node is simply the minimum/maximum of its children's minima/maxima.

### 10.9.2 Properties

The min/max *kd*tree has - besides the properties of an *kd*-tree - the special property that an inner node's min/max values coincide each with a min/max value of either one child. This allows to discard the storage of min/max values at the leaf nodes by storing two bits at inner nodes, assigning min/max values to the children: Each inner node's min/max values will be known in advance, where the root node's min/max values are stored separately. Each inner node has besides two min/max values also two bits given, defining to which child those min/max values are assigned (0: to the left child 1: to the right child). The non-assigned min/max values of the children are the from the current node already known min/max values. The two bits may also be stored in the least significant bits of the min/max values which have therefore to be approximated by fractioning them down/up.

The resulting memory reduction is not minor, as the leaf nodes of full binary *kd*-trees are one half of the tree's nodes.

### 10.9.3 Applications

Min/max *kd*-trees are used for ray casting isosurfaces/MIP (maximum intensity projection). Isosurface ray casting only traverses nodes for which the chosen isovalue lies between the min/max values of the current node. Nodes that do not fulfill this requirement do not contain an isosurface to the given isovalue and are therefore skipped (empty space skipping). For MIP, nodes are not traversed if their maximum is smaller than the current maximum intensity along the ray. The favorable visualization complexity of ray casting allows to ray cast (and even change the isosurface for) very large scalar fields at interactive framerates on commodity PCs. Especially implicit max *kd*-trees are an optimal choice for visualizing scalar fields defined on rectilinear grids (see [1][2][3]). Similarly an implicit min/max *kd*-tree may be used to efficiently evaluate queries such as terrain line of sight.<sup>[4]</sup>

### 10.9.4 See also

- *k*-*d* tree
- implicit *kd*-tree

### 10.9.5 References

- [1] Matthias Groß, Carsten Lojewski, Martin Bertram and Hans Hagen "Fast Implicit KD-Trees: Accelerated Isosurface Ray Tracing and Maximum Intensity Projection for Large Scalar Fields" CGIM07: Proceedings of Computer Graphics and Imaging (2007) 67-74
- [2] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek and Hans-Peter Seidel "Faster Isosurface Ray Tracing using Implicit KD-Trees" IEEE Transactions on Visualization and Computer Graphics (2005)
- [3] Matthias Groß (PhD, 2009) Towards Scientific Applications for Interactive Ray Casting
- [4] Bernardt Duvenhage "Using An Implicit Min/Max KD-Tree for Doing Efficient Terrain Line of Sight Calculations" in "Proceedings of the 6th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa", 2009.

## 10.10 Adaptive k-d tree

An **adaptive k-d tree** is a **tree** for multidimensional points where successive levels may be split along different dimensions.

### 10.10.1 References

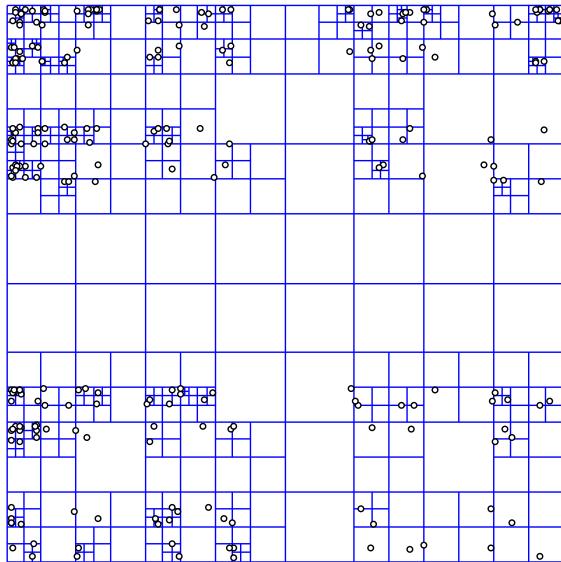
- Samet, Hanan (2006). *Foundations of multidimensional and metric data structures*. Morgan Kaufmann. ISBN 978-0-12-369446-1.

Black, Paul E. "Adaptive k-d tree". *Dictionary of Algorithms and Data Structures*. NIST.

## 10.11 Quadtree

A **quadtree** is a **tree data structure** in which each internal node has exactly four children. Quadtrees are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The regions may be square or rectangular, or may have arbitrary shapes. This data structure was named a quadtree by Raphael Finkel and J.L. Bentley in 1974. A similar partitioning is also known as a *Q-tree*. All forms of quadtrees share some common features:

- They decompose space into adaptable cells
- Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits
- The tree directory follows the spatial decomposition of the quadtree.



A point quadtree with point data. Bucket capacity 1.

### 10.11.1 Types

Quadtrees may be classified according to the type of data they represent, including areas, points, lines and curves. Quadtrees may also be classified by whether the shape of the tree is independent of the order data is processed. Some common types of quadtrees are:

#### The region quadtree

The region quadtree represents a partition of space in two dimensions by decomposing the region into four equal quadrants, subquadrants, and so on with each leaf node containing data corresponding to a specific subregion. Each node in the tree either has exactly four children, or has no children (a leaf node). The region quadtree is a type of [trie](#).

A region quadtree with a depth of  $n$  may be used to represent an image consisting of  $2^n \times 2^n$  pixels, where each pixel value is 0 or 1. The root node represents the entire image region. If the pixels in any region are not entirely 0s or 1s, it is subdivided. In this application, each leaf node represents a block of pixels that are all 0s or all 1s.

A region quadtree may also be used as a variable resolution representation of a data field. For example, the temperatures in an area may be stored as a quadtree, with each leaf node storing the average temperature over the subregion it represents.

If a region quadtree is used to represent a set of point data (such as the latitude and longitude of a set of cities), regions are subdivided until each leaf contains at most a single point.

#### Point quadtree

The point quadtree is an adaptation of a binary tree used to represent two-dimensional point data. It shares the features of all quadtrees but is a true tree as the center of a subdivision is always on a point. The tree shape depends on the order in which data is processed. It is often very efficient in comparing two-dimensional, ordered data points, usually operating in  $O(\log n)$  time.

**Node structure for a point quadtree** A node of a point quadtree is similar to a node of a binary tree, with the major difference being that it has four pointers (one for each quadrant) instead of two (“left” and “right”) as in an ordinary binary tree. Also a key is usually decomposed into two parts, referring to x and y coordinates. Therefore a node contains the following information:

- four pointers: quad[‘NW’], quad[‘NE’], quad[‘SW’], and quad[‘SE’]
- point; which in turn contains:
  - key; usually expressed as x, y coordinates
  - value; for example a name

#### Edge quadtree

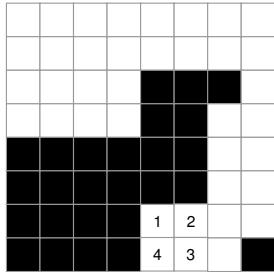
Edge quadtrees are specifically used to store lines rather than points. Curves are approximated by subdividing cells to a very fine resolution. This can result in extremely unbalanced trees which may defeat the purpose of indexing.

#### Polygonal map quadtree

The polygonal map quadtree (or PM Quadtree) is a variation of quadtree which is used to store collections of polygons that may be degenerate (meaning that they have isolated vertices or edges).<sup>[1]</sup> There are three main classes of PMQuadtrees, which vary depending on what information they store within each black node. PM3 quadtrees can store any amount of non-intersecting edges and at most one point. PM2 quadtrees are the same as PM3 quadtrees except that all edges must share the same end point. Finally PM1 quadtrees are similar to PM2, but black nodes can contain a point and its edges or just a set of edges that share a point, but you cannot have a point and a set of edges that do not contain the point.

### 10.11.2 Some common uses of quadtrees

- Image representation



- Spatial indexing
- Efficient collision detection in two dimensions
- View frustum culling of terrain data
- Storing sparse data, such as a formatting information for a spreadsheet or for some matrix calculations
- Solution of multidimensional fields (computational fluid dynamics, electromagnetism)
- Conway's Game of Life simulation program.<sup>[2]</sup>
- State estimation<sup>[3]</sup>
- Quadtrees are also used in the area of fractal image analysis

Quadtrees are the two-dimensional analog of octrees.

### 10.11.3 Pseudo code

The following pseudo code shows one means of implementing a quadtree which handles only points. There are other approaches available.

#### Prerequisites

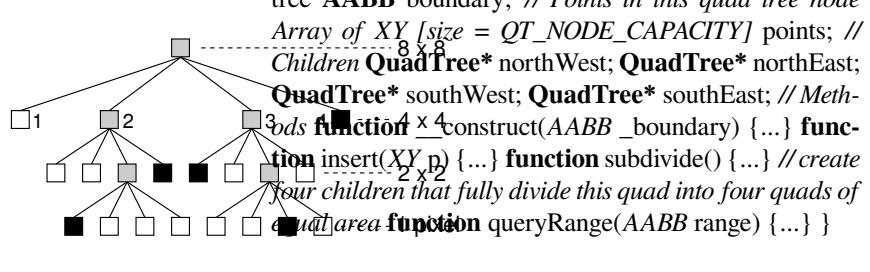
It is assumed these structures are used.

```
// Simple coordinate object to represent points and vectors
struct XY { float x; float y; function __construct(float _x, float _y) { ... } } // Axis-aligned bounding box with half dimension and center
struct AABB { XY center; float halfDimension; function __construct(XY center, float halfDimension) { ... } function containsPoint(XY p) { ... }
function intersectsAABB(AABB other) { ... } }
```

#### QuadTree class

This class represents both one quad tree and the node where it is rooted.

```
class QuadTree { // Arbitrary constant to indicate how many elements can be stored in this quad tree node
```



#### Insertion

The following method inserts a point into the appropriate quad of a quadtree, splitting if necessary.

```
class QuadTree {
 private XY boundary;
 private vector<XY> points;
 private QuadTree* northWest; QuadTree* northEast;
 QuadTree* southWest; QuadTree* southEast;
 public QuadTree(AABB boundary) {
 this->boundary = boundary;
 this->points = vector<XY>();
 this->subdivide();
 }
 function insert(XY p) {
 if (!boundary.containsPoint(p)) return false;
 if (points.size() < QT_NODE_CAPACITY) {
 points.append(p);
 return true;
 }
 if (northWest == NULL) {
 northWest = new QuadTree(boundary);
 northWest->subdivide();
 }
 if (northWest->insert(p)) return true;
 if (northEast == NULL) {
 northEast = new QuadTree(boundary);
 northEast->subdivide();
 }
 if (northEast->insert(p)) return true;
 if (southWest == NULL) {
 southWest = new QuadTree(boundary);
 southWest->subdivide();
 }
 if (southWest->insert(p)) return true;
 if (southEast == NULL) {
 southEast = new QuadTree(boundary);
 southEast->subdivide();
 }
 if (southEast->insert(p)) return true;
 return false;
 }
 function subdivide() {
 if (points.size() < QT_NODE_CAPACITY) return;
 vector<QuadTree*> children;
 for (int i = 0; i < 4; i++) {
 XY childBoundary;
 childBoundary.x = boundary.x + (i % 2) * halfDimension;
 childBoundary.y = boundary.y + (i / 2) * halfDimension;
 childBoundary.halfDimension = halfDimension / 2;
 children.push_back(new QuadTree(childBoundary));
 }
 for (int i = 0; i < points.size(); i++) {
 XY p = points[i];
 for (int j = 0; j < 4; j++) {
 XY childBoundary;
 childBoundary.x = boundary.x + (j % 2) * halfDimension;
 childBoundary.y = boundary.y + (j / 2) * halfDimension;
 childBoundary.halfDimension = halfDimension / 2;
 if (childBoundary.containsPoint(p)) {
 children[j]->insert(p);
 points.erase(points.begin() + i);
 break;
 }
 }
 }
 }
}
```

#### Query range

The following method finds all points contained within a range.

```
class QuadTree {
 private XY boundary;
 private vector<XY> points;
 private QuadTree* northWest; QuadTree* northEast;
 QuadTree* southWest; QuadTree* southEast;
 public QuadTree(AABB boundary) {
 this->boundary = boundary;
 this->points = vector<XY>();
 this->subdivide();
 }
 function queryRange(AABB range) {
 vector<XY> results;
 if (!boundary.intersectsAABB(range)) return results;
 if (points.size() < QT_NODE_CAPACITY) {
 if (range.containsPoint(points[0])) results.append(points[0]);
 return results;
 }
 if (northWest == NULL) {
 northWest = new QuadTree(boundary);
 northWest->subdivide();
 }
 northWest->queryRange(range);
 northEast->queryRange(range);
 southWest->queryRange(range);
 southEast->queryRange(range);
 for (int i = 0; i < points.size(); i++) {
 XY p = points[i];
 if (range.containsPoint(p)) results.append(p);
 }
 return results;
 }
}
```

### 10.11.4 See also

- Binary space partitioning

- Kd-tree
- Octree
- R-tree
- UB-tree
- Spatial database
- Subpaving
- Z-order curve
- C++ Implementation of a Quadtree used for spatial indexing of triangles
- Objective-C implementation of QuadTree used for GPS clustering
- SquareLanguage
- Working demonstration of Quadtree algorithm in Javascript
- MIT licensed Quadtree library in Javascript

### 10.11.5 References

#### Notes

- [1] Hanan Samet and Robert Webber. “Storing a Collection of Polygons Using Quadtrees”. *ACM Transactions on Graphics* July 1985: 182-222. *InfoLAB*. Web. 23 March 2012
- [2] Tomas G. Rokicki (2006-04-01). “An Algorithm for Compressing Space and Time”. Retrieved 2009-05-20.
- [3] Henning Eberhardt, Vesa Klumpp, Uwe D. Hanebeck, *Density Trees for Efficient Nonlinear State Estimation*, Proceedings of the 13th International Conference on Information Fusion, Edinburgh, United Kingdom, July, 2010.

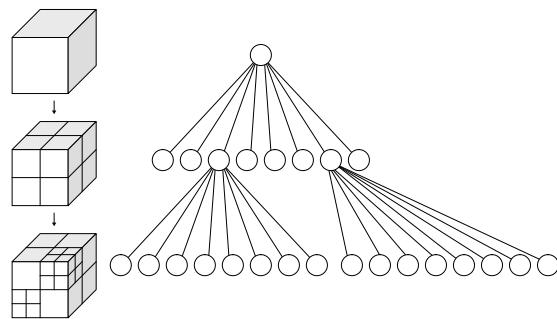
#### General references

1. Raphael Finkel and J.L. Bentley (1974). “Quad Trees: A Data Structure for Retrieval on Composite Keys”. *Acta Informatica* 4 (1): 1–9. doi:10.1007/BF00288933.
2. Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf (2000). *Computational Geometry* (2nd revised ed.). Springer-Verlag. ISBN 3-540-65620-0. Chapter 14: Quadtrees: pp. 291–306.
3. Samet, Hanan; Webber, Robert (July 1985). “Storing a Collection of Polygons Using Quadtrees” (PDF). Retrieved 23 March 2012.

### 10.11.6 External links

- A discussion of the Quadtree and an application
- Considerable discussion and demonstrations of Spatial Indexing
- Javascript Implementation of the QuadTree used for collision detection
- Java Implementation
- Java tutorial

## 10.12 Octree



Left: Recursive subdivision of a cube into octants. Right: The corresponding octree.

An **octree** is a tree data structure in which each **internal node** has exactly **eight children**. Octrees are most often used to partition a three dimensional space by recursively subdividing it into eight octants. Octrees are the three-dimensional analog of **quadtrees**. The name is formed from *oct* + *tree*, but note that it is normally written “*octree*” with only one “t”. Octrees are often used in **3D** graphics and **3D** game engines.

### 10.12.1 For spatial representation

Each node in an octree subdivides the space it represents into **eight octants**. In a **point region (PR) octree**, the node stores an explicit 3-dimensional point, which is the “center” of the subdivision for that node; the point defines one of the corners for each of the eight children. In a **matrix based (MX) octree**, the subdivision point is implicitly the center of the space the node represents. The root node of a PR octree can represent infinite space; the root node of an MX octree must represent a finite bounded space so that the implicit centers are well-defined. Note that Octrees are not the same as **k-d trees**: *k*-d trees split along a dimension and octrees split around a point. Also *k*-d trees are always binary, which is not the case for octrees. By using a depth-first search the nodes are to be traversed and only required surfaces are to be viewed.

## 10.12.2 History

The use of octrees for 3D computer graphics was pioneered by Donald Meagher at Rensselaer Polytechnic Institute, described in a 1980 report “Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer”,<sup>[1]</sup> for which he holds a 1995 patent (with a 1984 priority date) “High-speed image generation of complex solid objects using octree encoding”<sup>[2]</sup>

## 10.12.3 Common uses

- 3D computer graphics
- Spatial indexing
- Nearest neighbor search
- Efficient collision detection in three dimensions
- View frustum culling
- Fast Multipole Method
- Unstructured grid
- Finite element analysis
- Sparse voxel octree
- State estimation<sup>[3]</sup>
- Set estimation<sup>[4]</sup>

## 10.12.4 Application to color quantization

The octree color quantization algorithm, invented by Gervautz and Purgathofer in 1988, encodes image color data as an octree up to nine levels deep. Octrees are used because  $2^3 = 8$  and there are three color components in the RGB system. The node index to branch out from at the top level is determined by a formula that uses the most significant bits of the red, green, and blue color components, e.g.  $4r + 2g + b$ . The next lower level uses the next bit significance, and so on. Less significant bits are sometimes ignored to reduce the tree size.

The algorithm is highly memory efficient because the tree's size can be limited. The bottom level of the octree consists of leaf nodes that accrue color data not represented in the tree; these nodes initially contain single bits. If much more than the desired number of palette colors are entered into the octree, its size can be continually reduced by seeking out a bottom-level node and averaging its bit data up into a leaf node, pruning part of the tree. Once sampling is complete, exploring all routes in the tree down to the leaf nodes, taking note of the bits along the way, will yield approximately the required number of colors.

## 10.12.5 Implementation for point decomposition

The example recursive algorithm outline below (MATLAB syntax) decomposes an array of 3-dimensional points into octree style bins. The implementation begins with a single bin surrounding all given points, which then recursively subdivides into its 8 octree regions. Recursion is stopped when a given exit condition is met. Examples of such exit conditions (shown in code below) are:

- When a bin contains fewer than a given number of points
- When a bin reaches a minimum size or volume based on the length of its edges
- When recursion has reached a maximum number of subdivisions

```
function [binDepths,binParents,binCorners,pointBins] = OcTree(points)
binDepths = [0] % Initialize an array of bin depths with this single base-level bin
binParents = [0] % This base level bin is not a child of other bins
binCorners = [min(points) max(points)] % It surrounds all points in XYZ space
pointBins(:) = 1 % Initially, all points are assigned to this first bin
divide(1) % Begin dividing this first bin function
divide(binNo) % If this bin meets any exit conditions, do not divide it any further.
binPointCount = nnz(pointBins==binNo)
binEdgeLengths = binCorners(binNo,1:3) - binCorners(binNo,4:6)
binDepth = binDepths(binNo)
exitConditionsMet = binPointCount<value || min(binEdgeLengths)<value || binDepth>value
if exitConditionsMet
 return; % Exit recursive function
end % Otherwise, split this bin into 8 new sub-bins with a new division point
newDiv = (binCorners(binNo,1:3) + binCorners(binNo,4:6)) / 2
for i = 1:8
 newBinNo = length(binDepths) + 1
 binDepths(newBinNo) = binDepths(binNo) + 1
 binParents(newBinNo) = binNo
 binCorners(newBinNo) = [one of the 8 pairs of the newDiv with minCorner or maxCorner]
 oldBinMask = pointBins==binNo ...
 Calculate which points in pointBins==binNo now belong in newBinNo
 pointBins(newBinMask) = newBinNo % Recursively divide this newly created bin
end
divide(newBinNo)
```

## 10.12.6 Example color quantization

Taking the full list of colors of a 24-bit RGB image as point input to the Octree point decomposition implementation outlined above, the following example show the results of octree color quantization. The first image is the original (532818 distinct colors), while the second is the quantized image (184 distinct colors) using octree decomposition, with each pixel assigned the color at the center of the octree bin in which it falls. Alternatively, final

colors could be chosen at the centroid of all colors in each octree bin, however this added computation has very little effect on the visual result.<sup>[5]</sup>

```
% Read the original RGB image Img = imread('IMG_9980.CR2'); % Extract pixels as RGB
point triplets pts = reshape(Img,[],3); % Create OcTree
decomposition object using a target bin capacity OT =
OcTree(pts,'BinCapacity',ceil((size(pts,1) / 256) *7));
% Find which bins are "leaf nodes" on the octree object
leafs = find(~ismember(1:OT.BinCount, OT.BinParents) & ...
ismember(1:OT.BinCount,OT.PointBins)); % Find
the central RGB location of each leaf bin binCents =
mean(reshape(OT.BinBoundaries(leafs,:),[],3,2),3); % Make a new "indexed" image with a color map ImgIdx =
zeros(size(Img,1), size(Img,2)); for i = 1:length(leafs)
pxNos = find(OT.PointBins==leafs(i)); ImgIdx(pxNos) =
i; end ImgMap = binCents / 255; % Convert 8-bit
color to MATLAB rgb values % Display the original
532818-color image and resulting 184-color image figure
subplot(1,2,1), imshow(Img) title(sprintf('Original %d
color image', size(unique(pts,'rows'),1))) subplot(1,2,2),
imshow(ImgIdx, ImgMap) title(sprintf('Octree-
quantized %d color image', size(ImgMap,1)))
```

### 10.12.7 See also

- Binary space partitioning
- Bounding interval hierarchy
- *Cube 2: Sauerbraten*, a 3D game engine in which geometry is almost entirely based on octrees
- *id Tech 6* an in-development 3D game engine that utilizes voxels stored in octrees
- *Irrlicht Engine*, supports octree scene nodes
- Klee's measure problem
- Linear octree
- *OGRE*, has an Octree Scene Manager Implementation
- Subpaving
- Voxel

### 10.12.8 References

- [1] Meagher, Donald (October 1980). "Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer". *Rensselaer Polytechnic Institute* (Technical Report IPL-TR-80-111).
- [2] Meagher, Donald. "High-speed image generation of complex solid objects using octree encoding". USPO. Retrieved 20 September 2012.

[3] Henning Eberhardt, Vesa Klumpp, Uwe D. Hanebeck, *Density Trees for Efficient Nonlinear State Estimation*, Proceedings of the 13th International Conference on Information Fusion, Edinburgh, United Kingdom, July, 2010.

[4] V. Drevelle, L. Jaulin and B. Zerr, *Guaranteed Characterization of the Explored Space of a Mobile Robot by using Subpavings*, NOLCOS 2013.

[5] Bloomberg, Dan S. "Color quantization using octrees.", 4 September 2008. Retrieved on 12 December 2014.

### 10.12.9 External links

- Octree Quantization in Microsoft Systems Journal
- Color Quantization using Octrees in Dr. Dobb's
- Color Quantization using Octrees in Dr. Dobb's Source Code
- Octree Color Quantization Overview
- Parallel implementation of octtree generation algorithm, P. Sojan Lal, A Unnikrishnan, K Poulose Jacob, ICIP 1997, IEEE Digital Library
- Generation of Octrees from Raster Scan with Reduced Information Loss, P. Sojan Lal, A Unnikrishnan, K Poulose Jacob, IASTED International conference VIIP 2001
- C++ implementation (GPL license)
- Parallel Octrees for Finite Element Applications
- Dendro: parallel multigrid for octree meshes (MPI/C++ implementation)
- Video: Use of an octree in state estimation
- Source code of an OpenCL raytracer applet using an Octree
- MATLAB implementation of OcTree decomposition

## 10.13 Linear octrees

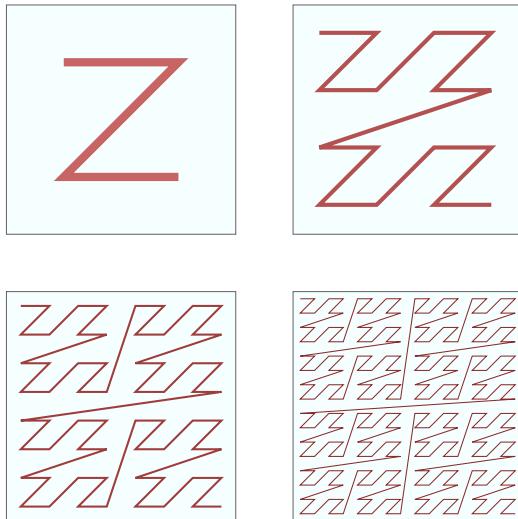
A **linear octree** is an octree that is represented by a linear array instead of a tree data structure.

To simplify implementation, a linear octree is usually complete (that is, every internal node has exactly 8 child nodes) and where the maximum permissible depth is fixed a priori (making it sufficient to store the complete list of leaf nodes). That is, all the nodes of the octree can be generated from the list of its leaf nodes. Space filling curves are often used to represent linear octrees.

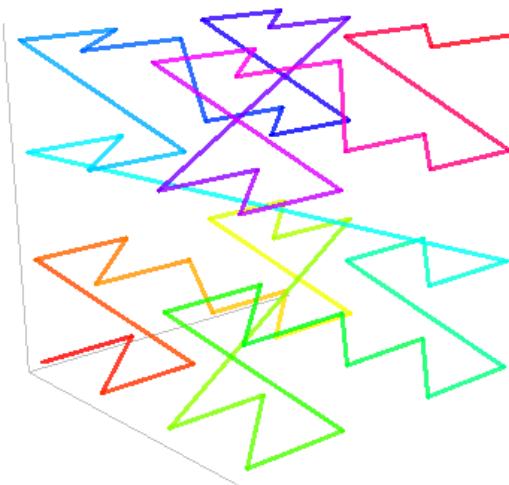
## 10.14 Z-order curve

Not to be confused with Z curve or Z-order.

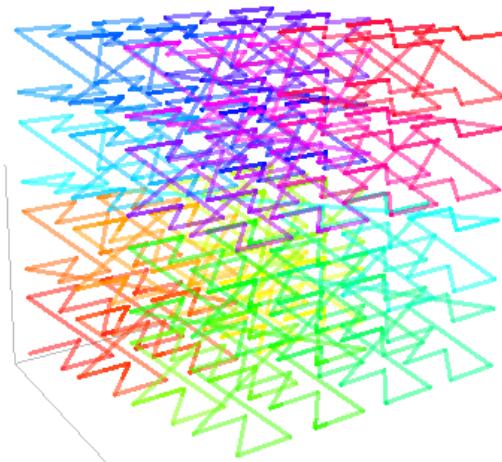
In mathematical analysis and computer science, Z-



Four iterations of the Z-order curve.



**order, Morton order, or Morton code** is a function which maps multidimensional data to one dimension while preserving locality of the data points. It was introduced in 1966 by G. M. Morton.<sup>[1]</sup> The z-value of a point in multidimensions is simply calculated by interleaving the **binary** representations of its coordinate values. Once the data are sorted into this ordering, any one-dimensional data structure can be used such as **binary search trees**, **B-trees**, **skip lists** or (with low significant bits truncated) **hash tables**. The resulting ordering can equivalently be described as the order one would get from a depth-first traversal of a **quadtree**.



Z-order curve iterations extended to three dimensions.

### 10.14.1 Coordinate values

The figure below shows the Z-values for the two dimensional case with integer coordinates  $0 \leq x \leq 7$ ,  $0 \leq y \leq 7$  (shown both in decimal and binary). Interleaving the binary coordinate values yields binary z-values as shown. Connecting the z-values in their numerical order produces the recursively Z-shaped curve. Two-dimensional Z-values are also called as quadkey ones.

| x:  | 0      | 1      | 2      | 3      | 4      | 5      | 6      | 7      |
|-----|--------|--------|--------|--------|--------|--------|--------|--------|
| 0   | 000    | 001    | 010    | 011    | 100    | 101    | 110    | 111    |
| 000 | 000000 | 000001 | 000100 | 000101 | 010000 | 010001 | 010100 | 010101 |
| 001 | 000010 | 000011 | 000110 | 000111 | 010010 | 010011 | 010110 | 010111 |
| 010 | 001000 | 001001 | 001100 | 001101 | 011000 | 011001 | 011100 | 011101 |
| 011 | 001010 | 001011 | 001110 | 001111 | 011010 | 011011 | 011110 | 011111 |
| 100 | 100000 | 100001 | 100100 | 100101 | 110000 | 110001 | 110100 | 110101 |
| 101 | 100010 | 100011 | 100110 | 100111 | 110010 | 110011 | 110110 | 110111 |
| 110 | 101000 | 101001 | 101100 | 101101 | 111000 | 111001 | 111100 | 111101 |
| 111 | 101010 | 101011 | 101110 | 101111 | 111010 | 111011 | 111110 | 111111 |

The Z-values of x's are described as binary numbers:

$x[] = \{0b000000, 0b000001, 0b000100, 0b000101, 0b010000, 0b010001, 0b010100, 0b010101\}$

The sum and subtraction of two x's are calculated by using bitwise operations:

$x[i+j] = ((x[i] \mid 0b101010) + x[j]) \& 0b01010101$   
 $x[i-j] = (x[i] - x[j]) \& 0b01010101$  if  $i \geq j$

### 10.14.2 Efficiently building quadtrees

The Z-ordering can be used to efficiently build a quadtree for a set of points.<sup>[2]</sup> The basic idea is to sort the input set according to Z-order. Once sorted, the points can either be stored in a binary search tree and used directly, which is called a linear quadtree,<sup>[3]</sup> or they can be used to build a pointer based quadtree.

The input points are usually scaled in each dimension to be positive integers, either as a fixed point representation over the unit range  $[0, 1]$  or corresponding to the machine word size. Both representations are equivalent and allow for the highest order non-zero bit to be found in constant time. Each square in the quadtree has a side length which is a power of two, and corner coordinates which are multiples of the side length. Given any two points, the *derived square* for the two points is the smallest square covering both points. The interleaving of bits from the x and y components of each point is called the *shuffle* of x and y, and can be extended to higher dimensions.<sup>[2]</sup>

Points can be sorted according to their shuffle without explicitly interleaving the bits. To do this, for each dimension, the most significant bit of the *exclusive or* of the coordinates of the two points for that dimension is examined. The dimension for which the most significant bit is largest is then used to compare the two points to determine their shuffle order.

The exclusive or operation masks off the higher order bits for which the two coordinates are identical. Since the shuffle interleaves bits from higher order to lower order, identifying the coordinate with the largest most significant bit, identifies the first bit in the shuffle order which differs, and that coordinate can be used to compare the two points.<sup>[4]</sup> This is shown in the following Python code:

```
def cmp_zorder(a, b): j = 0 k = 0 x = 0 for k in range(dim): y = a[k] ^ b[k] if less_msb(x, y): j = k x = y return a[j] - b[j]
```

One way to determine whether the most significant smaller is to compare the floor of the base-2 logarithm of each point. It turns out the following operation is equivalent, and only requires exclusive or operations:<sup>[4]</sup>

```
def less_msb(x, y): return x < y and x < (x ^ y)
```

It is also possible to compare floating point numbers using the same technique. The *less\_msb* function is modified to first compare the exponents. Only when they are equal is the standard *less\_msb* function used on the mantissas.<sup>[5]</sup>

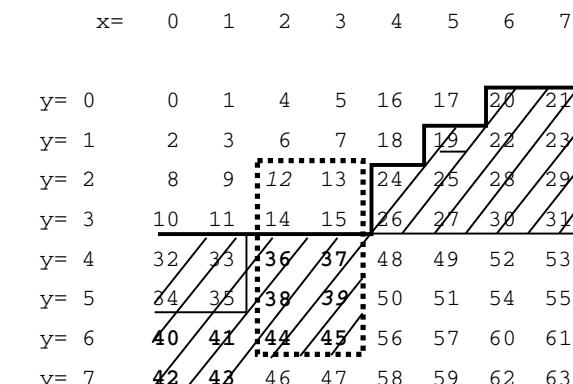
Once the points are in sorted order, two properties make it easy to build a quadtree: The first is that the points contained in a square of the quadtree form a contiguous interval in the sorted order. The second is that if more than one child of a square contains an input point, the square is the *derived square* for two adjacent points in the sorted order.

For each adjacent pair of points, the derived square is computed and its side length determined. For each derived square, the interval containing it is bounded by the first larger square to the right and to the left in sorted order.<sup>[2]</sup> Each such interval corresponds to a square in the quadtree. The result of this is a compressed quadtree, where only nodes containing input points or two or more children are present. A non-compressed quadtree can be built by restoring the missing nodes, if desired.

Rather than building a pointer based quadtree, the points can be maintained in sorted order in a data structure such as a binary search tree. This allows points to be added and deleted in  $O(\log n)$  time. Two quadtrees can be merged by merging the two sorted sets of points, and removing duplicates. Point location can be done by searching for the points preceding and following the query point in the sorted order. If the quadtree is compressed, the predecessor node found may be an arbitrary leaf inside the compressed node of interest. In this case, it is necessary to find the predecessor of the least common ancestor of the query point and the leaf found.<sup>[6]</sup>

### 10.14.3 Use with one-dimensional data structures for range searching

Although preserving locality well, for efficient range searches an algorithm is necessary for calculating, from a point encountered in the data structure, the next Z-value which is in the multidimensional search range:



In this example, the range being queried ( $x = 2, \dots, 3, y = 2, \dots, 6$ ) is indicated by the dotted rectangle. Its highest Z-value (MAX) is 45. In this example, the value  $F = 19$  is encountered when searching a data structure in increasing Z-value direction, so we would have to search in the interval between  $F$  and MAX (hatched area). To speed up the search, one would calculate the next Z-value which is in the search range, called BIGMIN (36 in the example) and only search in the interval between BIGMIN and MAX (bold values), thus skipping most of the hatched area. Searching in decreasing direction is analogous with LITMAX which is the highest Z-value in the query range lower than  $F$ . The BIGMIN problem has first been stated

and its solution shown in Tropf and Herzog.<sup>[7]</sup> This solution is also used in UB-trees (“GetNextZ-address”). As the approach does not depend on the one dimensional data structure chosen, there is still free choice of structuring the data, so well known methods such as balanced trees can be used to cope with dynamic data (in contrast for example to R-trees where special considerations are necessary). Similarly, this independence makes it easier to incorporate the method into existing databases.

Applying the method hierarchically (according to the data structure at hand), optionally in both increasing and decreasing direction, yields highly efficient multidimensional range search which is important in both commercial and technical applications, e.g. as a procedure underlying nearest neighbour searches. Z-order is one of the few multidimensional access methods that has found its way into commercial database systems (Oracle database 1995,<sup>[8]</sup> Transbase 2000<sup>[9]</sup>).

As long ago as 1966, G.M.Morton proposed Z-order for file sequencing of a static two dimensional geographical database. Areal data units are contained in one or a few quadratic frames represented by their sizes and lower right corner Z-values, the sizes complying with the Z-order hierarchy at the corner position. With high probability, changing to an adjacent frame is done with one or a few relatively small scanning steps.

#### 10.14.4 Related structures

As an alternative, the Hilbert curve has been suggested as it has a better order-preserving behaviour, but here the calculations are much more complicated, leading to significant processor overhead. BIGMIN source code for both Z-curve and Hilbert-curve were described in a patent by H. Tropf.<sup>[10]</sup>

For a recent overview on multidimensional data processing, including e.g. nearest neighbour searches, see Hanan Samet's textbook.<sup>[11]</sup>

#### 10.14.5 Applications in linear algebra

The Strassen algorithm for matrix multiplication is based on splitting the matrices in four blocks, and then recursively splitting each of these blocks in four smaller blocks, until the blocks are single elements (or more practically: until reaching matrices so small that the trivial algorithm is faster). Arranging the matrix elements in Z-order then improves locality, and has the additional advantage (compared to row- or column-major ordering) that the subroutine for multiplying two blocks does not need to know the total size of the matrix, but only the size of the blocks and their location in memory. Effective use of Strassen multiplication with Z-order has been demonstrated, see Valsalam and Skjellum's 2002 paper.<sup>[12]</sup>

#### 10.14.6 See also

- Space filling curve
- UB-tree
- Hilbert curve
- Hilbert R-tree
- Spatial index
- Geohash
- Locality preserving hashing
- Matrix representation
- Linear algebra

#### 10.14.7 References

- [1] Morton, G. M. (1966), *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*, Technical Report, Ottawa, Canada: IBM Ltd.
- [2] Bern, M.; Eppstein, D.; Teng, S.-H. (1999), “Parallel construction of quadtrees and quality triangulations”, *Int. J. Comp. Geom. & Appl.* **9** (6): 517–532, doi:10.1142/S0218195999000303.
- [3] Gargantini, I. (1982), “An effective way to represent quadtrees”, *Communications of the ACM* **25** (12): 905–910, doi:10.1145/358728.358741.
- [4] Chan, T. (2002), “Closest-point problems simplified on the RAM”, *ACM-SIAM Symposium on Discrete Algorithms*.
- [5] Connor, M.; Kumar, P (2009), “Fast construction of k-nearest neighbour graphs for point clouds”, *IEEE Transactions on Visualization and Computer Graphics* (PDF)
- [6] Har-Peled, S. (2010), *Data structures for geometric approximation* (PDF)
- [7] Tropf, H.; Herzog, H. (1981), “Multidimensional Range Search in Dynamically Balanced Trees” (PDF), *Angewandte Informatik* **2**: 71–77.
- [8] Gaede, Volker; Guenther, Oliver (1998), “Multidimensional access methods” (PDF), *ACM Computing Surveys* **30** (2): 170–231, doi:10.1145/280277.280279.
- [9] Ramsak, Frank; Markl, Volker; Fenk, Robert; Zirkel, Martin; Elhardt, Klaus; Bayer, Rudolf (2000), “Integrating the UB-tree into a Database System Kernel”, *Int. Conf. on Very Large Databases (VLDB)* (PDF), pp. 263–272.
- [10] US 7321890, Tropf, H., “Database system and method for organizing data elements according to a Hilbert curve”, issued January 22, 2008.
- [11] Samet, H. (2006), *Foundations of Multidimensional and Metric Data Structures*, San Francisco: Morgan-Kaufmann.

- [12] Vinod Valsalam, Anthony Skjellum: A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience* 14(10): 805-839 (2002)

### 10.14.8 External links

- STANN: A library for approximate nearest neighbor search, using Z-order curve
- Methods for programming bit interleaving, Sean Eron Anderson, Stanford University

## 10.15 UB-tree

The **UB-tree** as proposed by Rudolf Bayer and Volker Markl is a balanced tree for storing and efficiently retrieving multidimensional data. It is basically a **B+ tree** (information only in the leaves) with records stored according to **Z-order**, also called Morton order. Z-order is simply calculated by bitwise interleacing the keys.

Insertion, deletion, and point query are done as with ordinary B+ trees. To perform range searches in multidimensional point data, however, an algorithm must be provided for calculating, from a point encountered in the data base, the next Z-value which is in the multidimensional search range.

The original algorithm to solve this key problem was exponential with the dimensionality and thus not feasible<sup>[1]</sup> (“GetNextZ-address”). A solution to this “crucial part of the UB-tree range query” linear with the z-address bit length has been described later.<sup>[2]</sup> This method has already been described in an older paper<sup>[3]</sup> where using Z-order with search trees has first been proposed.

### 10.15.1 References

- [1] Markl, V. (1999). “MISTRAL: Processing Relational Queries using a Multidimensional Access Technique”.
- [2] Ramsak, Frank; Markl, Volker; Fenk, Robert; Zirkel, Martin; Elhardt, Klaus; Bayer, Rudolf (September 10–14, 2000). *Integrating the UB-tree into a Database System Kernel*. 26th International Conference on Very Large Data Bases. pp. 263–272.
- [3] Tropf, H.; Herzog, H. “Multidimensional Range Search in Dynamically Balanced Trees” (PDF). *Angewandte Informatik (Applied Informatics)* (2/1981): 71–77. ISSN 0013-5704.

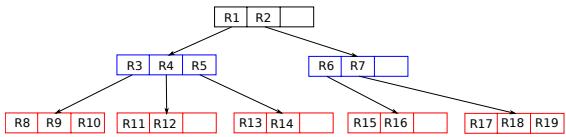
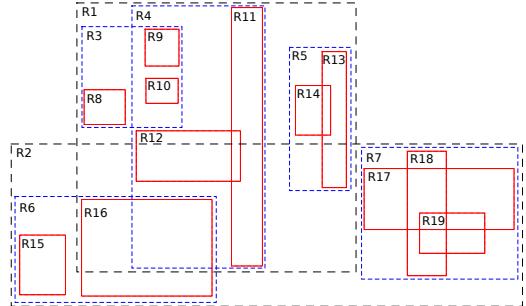
### 10.15.2 External links

- Mistral

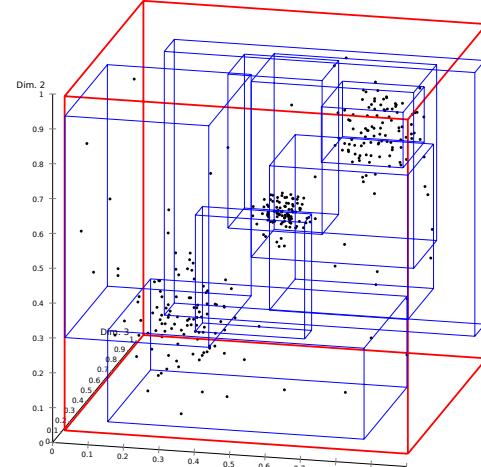
## 10.16 R-tree

This article is about the data structure. For the type of metric space, see **Real tree**.

**R-trees** are tree data structures used for spatial ac-



Simple example of an R-tree for 2D rectangles



Visualization of an R\*-tree for 3D points using ELKI (the cubes are directory pages)

cess methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. The R-tree was proposed by Antonin Guttman in 1984<sup>[1]</sup> and has found significant use in both theoretical and applied contexts.<sup>[2]</sup> A common real-world usage for an R-tree might be to store spatial objects such as restaurant locations or the polygons that typical maps are made of: streets, buildings, outlines of lakes, coastlines, etc. and then find answers quickly to queries such as “Find all museums within 2 km of my current location”, “retrieve all road segments within 2 km of my location” (to display them in a navigation system) or “find the near-

est gas station" (although not taking roads into account). The R-tree can also accelerate nearest neighbor search<sup>[3]</sup> for various distance metrics, including great-circle distance.<sup>[4]</sup>

### 10.16.1 R-tree idea

The key idea of the data structure is to group nearby objects and represent them with their **minimum bounding rectangle** in the next higher level of the tree; the "R" in R-tree is for rectangle. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. At the leaf level, each rectangle describes a single object; at higher levels the aggregation of an increasing number of objects. This can also be seen as an increasingly coarse approximation of the data set.

Similar to the **B-tree**, the R-tree is also a balanced search tree (so all leaf nodes are at the same height), organizes the data in pages, and is designed for storage on disk (as used in **databases**). Each page can contain a maximum number of entries, often denoted as  $M$ . It also guarantees a minimum fill (except for the root node), however best performance has been experienced with a minimum fill of 30%–40% of the maximum number of entries (B-trees guarantee 50% page fill, and **B\*-trees** even 66%). The reason for this is the more complex balancing required for spatial data as opposed to linear data stored in B-trees.

As with most trees, the searching algorithms (e.g., intersection, containment, nearest neighbor search) are rather simple. The key idea is to use the bounding boxes to decide whether or not to search inside a subtree. In this way, most of the nodes in the tree are never read during a search. Like B-trees, this makes R-trees suitable for large data sets and **databases**, where nodes can be paged to memory when needed, and the whole tree cannot be kept in main memory.

The key difficulty of R-trees is to build an efficient tree that on one hand is balanced (so the leaf nodes are at the same height) on the other hand the rectangles do not cover too much empty space and do not overlap too much (so that during search, fewer subtrees need to be processed). For example, the original idea for inserting elements to obtain an efficient tree is to always insert into the subtree that requires least enlargement of its bounding box. Once that page is full, the data is split into two sets that should cover the minimal area each. Most of the research and improvements for R-trees aims at improving the way the tree is built and can be grouped into two objectives: building an efficient tree from scratch (known as bulk-loading) and performing changes on an existing tree (insertion and deletion).

R-trees do not guarantee good **worst-case** performance, but generally perform well with real-world data.<sup>[5]</sup> While more of theoretical interest, the (bulk-loaded) **Priority R-tree** variant of the R-tree is worst-case optimal,<sup>[6]</sup> but due

to the increased complexity, has not received much attention in practical applications so far.

When data is organized in an R-tree, the  $k$  nearest neighbors (for any  $L^p$ -Norm) of all points can efficiently be computed using a spatial join.<sup>[7]</sup> This is beneficial for many algorithms based on the  $k$  nearest neighbors, for example the **Local Outlier Factor**. DeLi-Clu,<sup>[8]</sup> Density-Link-Clustering is a cluster analysis algorithm that uses the R-tree structure for a similar kind of spatial join to efficiently compute an **OPTICS** clustering.

### 10.16.2 Variants

- R\* tree
- R+ tree
- Hilbert R-tree
- X-tree

### 10.16.3 Algorithm

#### Data layout

Data in R-trees is organized in pages, that can have a variable number of entries (up to some pre-defined maximum, and usually above a minimum fill). Each entry within a non-leaf node stores two pieces of data: a way of identifying a **child node**, and the **bounding box** of all entries within this child node. Leaf nodes store the data required for each child, often a point or bounding box representing the child and an external identifier for the child. For point data, the leaf entries can be just the points themselves. For polygon data (that often requires the storage of large polygons) the common setup is to store only the **MBR** (minimum bounding rectangle) of the polygon along with a unique identifier in the tree.

#### Search

The input is a search rectangle (Query box). Searching is quite similar to searching in a **B+ tree**. The search starts from the root node of the tree. Every internal node contains a set of rectangles and pointers to the corresponding child node and every leaf node contains the rectangles of spatial objects (the pointer to some spatial object can be there). For every rectangle in a node, it has to be decided if it overlaps the search rectangle or not. If yes, the corresponding child node has to be searched also. Searching is done like this in a recursive manner until all overlapping nodes have been traversed. When a leaf node is reached, the contained bounding boxes (rectangles) are tested against the search rectangle and their objects (if there are any) are put into the result set if they lie within the search rectangle.

For priority search such as nearest neighbor search, the query consists of a point or rectangle. The root node is inserted into the priority queue. Until the queue is empty or the desired number of results have been returned the search continues by processing the nearest entry in the queue. Tree nodes are expanded and their children reinserted. Leaf entries are returned when encountered in the queue.<sup>[9]</sup> This approach can be used with various distance metrics, including great-circle distance for geographic data.<sup>[4]</sup>

## Insertion

To insert an object, the tree is traversed recursively from the root node. At each step, all rectangles in the current directory node are examined, and a candidate is chosen using a heuristic such as choosing the rectangle which requires least enlargement. The search then descends into this page, until reaching a leaf node. If the leaf node is full, it must be split before the insertion is made. Again, since an exhaustive search is too expensive, a heuristic is employed to split the node into two. Adding the newly created node to the previous level, this level can again overflow, and these overflows can propagate up to the root node; when this node also overflows, a new root node is created and the tree has increased in height.

**Choosing the insertion subtree** At each level, the algorithm needs to decide in which subtree to insert the new data object. When a data object is fully contained in a single rectangle, the choice is clear. When there are multiple options or rectangles in need of enlargement, the choice can have a significant impact on the performance of the tree.

In the classic R-tree, objects are inserted into the subtree that needs the least enlargement. In the more advanced R\*-tree, a mixed heuristic is employed. At leaf level, it tries to minimize the overlap (in case of ties, prefer least enlargement and then least area); at the higher levels, it behaves similar to the R-tree, but on ties again preferring the subtree with smaller area. The decreased overlap of rectangles in the R\*-tree is one of the key benefits over the traditional R-tree (this is also a consequence of the other heuristics used, not only the subtree choosing).

**Splitting an overflowing node** Since redistributing all objects of a node into two nodes has an exponential number of options, a heuristic needs to be employed to find the best split. In the classic R-tree, Guttman proposed two such heuristics, called QuadraticSplit and LinearSplit. In quadratic split, the algorithm searches for the pair of rectangles that is the worst combination to have in the same node, and puts them as initial objects into the two new groups. It then searches for the entry which has the strongest preference for one of the groups (in terms

of area increase) and assigns the object to this group until all objects are assigned (satisfying the minimum fill).

There are other splitting strategies such as Greene's Split,<sup>[10]</sup> the R\*-tree splitting heuristic<sup>[11]</sup> (which again tries to minimize overlap, but also prefers quadratic pages) or the linear split algorithm proposed by Ang and Tan<sup>[12]</sup> (which however can produce very unregular rectangles, which are less performant for many real world range and window queries). In addition to having a more advanced splitting heuristic, the R\*-tree also tries to avoid splitting a node by reinserting some of the node members, which is similar to the way a B-tree balances overflowing nodes. This was shown to also reduce overlap and thus increase tree performance.

Finally, the X-tree<sup>[13]</sup> can be seen as a R\*-tree variant that can also decide to not split a node, but construct a so-called super-node containing all the extra entries, when it doesn't find a good split (in particular for high-dimensional data).

## Deletion

Deleting an entry from a page may require updating the bounding rectangles of parent pages. However, when a page is underfull, it will not be balanced with its neighbors. Instead, the page will be dissolved and all its children (which may be subtrees, not only leaf objects) will be reinserted. If during this process the root node has a single element, the tree height can decrease.

## Bulk-loading

- Nearest-X - Objects are sorted by their first coordinate ("X") and then split into pages of the desired size.
- Packed Hilbert R-tree - variation of Nearest-X, but sorting using the Hilbert value of the center of a rectangle instead of using the X coordinate. There is no guarantee the pages will not overlap.
- Sort-Tile-Recursive (STR):<sup>[14]</sup> Another variation of Nearest-X, that estimates the total number of leaves required as  $l = \lceil \text{objects of number/capacity} \rceil$ , the required split factor in each dimension to achieve this as  $s = \lceil l^{1/d} \rceil$ , then repeatedly splits each dimensions successively into  $s$  equal sized partitions using 1-dimensional sorting. The resulting pages, if they occupy more than one page, are again bulk-loaded using the same algorithm. For point data, the leaf nodes will not overlap, and "tile" the data space into approximately equal sized pages.
- Priority R-tree

### 10.16.4 See also

- Segment tree
- Interval tree - A degenerate R-tree for one dimension (usually time).
- Bounding volume hierarchy
- Spatial index
- GiST

### 10.16.5 References

- [1] Guttman, A. (1984). “R-Trees: A Dynamic Index Structure for Spatial Searching”. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84* (PDF). p. 47. doi:10.1145/602259.602266. ISBN 0897911288.
- [2] Y. Manolopoulos; A. Nanopoulos; Y. Theodoridis (2006). *R-Trees: Theory and Applications*. Springer. ISBN 978-1-85233-977-7. Retrieved 8 October 2011.
- [3] Roussopoulos, N.; Kelley, S.; Vincent, F. D. R. (1995). “Nearest neighbor queries”. *Proceedings of the 1995 ACM SIGMOD international conference on Management of data - SIGMOD '95*. p. 71. doi:10.1145/223784.223794. ISBN 0897917316.
- [4] Schubert, E.; Zimek, A.; Kriegel, H. P. (2013). “Geometric Distance Queries on R-Trees for Indexing Geographic Data”. *Advances in Spatial and Temporal Databases*. Lecture Notes in Computer Science **8098**. p. 146. doi:10.1007/978-3-642-40235-7\_9. ISBN 978-3-642-40234-0.
- [5] Hwang, S.; Kwon, K.; Cha, S. K.; Lee, B. S. (2003). “Performance Evaluation of Main-Memory R-tree Variants”. *Advances in Spatial and Temporal Databases*. Lecture Notes in Computer Science **2750**. p. 10. doi:10.1007/978-3-540-45072-6\_2. ISBN 978-3-540-40535-1.
- [6] Arge, L.; De Berg, M.; Haverkort, H. J.; Yi, K. (2004). “The Priority R-tree”. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04* (PDF). p. 347. doi:10.1145/1007568.1007608. ISBN 1581138598.
- [7] Brinkhoff, T.; Kriegel, H. P.; Seeger, B. (1993). “Efficient processing of spatial joins using R-trees”. *ACM SIGMOD Record* **22** (2): 237. doi:10.1145/170036.170075.
- [8] Achtert, E.; Böhm, C.; Kröger, P. (2006). “DeLi-Clu: Boosting Robustness, Completeness, Usability, and Efficiency of Hierarchical Clustering by a Closest Pair Ranking”. *LNCS: Advances in Knowledge Discovery and Data Mining*. Lecture Notes in Computer Science **3918**: 119–128. doi:10.1007/11731139\_16. ISBN 978-3-540-33206-0.
- [9] Kuan, J.; Lewis, P. (1997). “Fast k nearest neighbour search for R-tree family”. *Proceedings of ICICS, 1997 International Conference on Information, Communications and Signal Processing. Theme: Trends in Information Systems Engineering and Wireless Multimedia Communications (Cat. No.97TH8237)*. p. 924. doi:10.1109/ICICS.1997.652114. ISBN 0-7803-3676-3.
- [10] Greene, D. (1989). “An implementation and performance analysis of spatial data access methods”. *[1989] Proceedings. Fifth International Conference on Data Engineering*. pp. 606–615. doi:10.1109/ICDE.1989.47268. ISBN 0-8186-1915-5.
- [11] Beckmann, N.; Kriegel, H. P.; Schneider, R.; Seeger, B. (1990). “The R\*-tree: an efficient and robust access method for points and rectangles”. *Proceedings of the 1990 ACM SIGMOD international conference on Management of data - SIGMOD '90* (PDF). p. 322. doi:10.1145/93597.98741. ISBN 0897913655.
- [12] Ang, C. H.; Tan, T. C. (1997). “New linear node splitting algorithm for R-trees”. In Scholl, Michel; Voisard, Agnès. *Proceedings of the 5th International Symposium on Advances in Spatial Databases (SSD '97), Berlin, Germany, July 15–18, 1997*. Lecture Notes in Computer Science **1262**. Springer. pp. 337–349. doi:10.1007/3-540-63238-7\_38.
- [13] Berchtold, Stefan; Keim, Daniel A.; Kriegel, Hans-Peter (1996). “The X-Tree: An Index Structure for High-Dimensional Data”. *Proceedings of the 22nd VLDB Conference* (Mumbai, India): 28–39.
- [14] Leutenegger, Scott T.; Edgington, Jeffrey M.; Lopez, Mario A. (February 1997). “STR: A Simple and Efficient Algorithm for R-Tree Packing”.

### 10.16.6 External links

- R-tree portal
- R-tree implementations: C & C++, Java, Java applet, Common Lisp, Python, Javascript, Javascript AMD module
- Boost.Geometry library containing R-tree implementation (various splitting algorithms)

## 10.17 R+ tree

An **R+ tree** is a method for looking up data using a location, often (x, y) coordinates, and often for locations on the surface of the earth. Searching on one number is a solved problem; searching on two or more, and asking for locations that are nearby in both x and y directions, requires craftier algorithms.

Fundamentally, an R+ tree is a tree data structure, a variant of the R tree, used for indexing spatial information.

### 10.17.1 Difference between R+ trees and R trees

R+ trees are a compromise between R-trees and kd-trees: they avoid overlapping of internal nodes by inserting an object into multiple leaves if necessary. **Coverage** is the entire area to cover all related rectangles. **Overlap** is the entire area which is contained in two or more nodes.<sup>[1]</sup> Minimal coverage reduces the amount of “dead space” (empty area) which is covered by the nodes of the R-tree. Minimal overlap reduces the set of search paths to the leaves (even more critical for the access time than minimal coverage). Efficient search requires minimal coverage and overlap.

R+ trees differ from R trees in that:

- Nodes are not guaranteed to be at least half filled
- The entries of any internal node do not overlap
- An object ID may be stored in more than one leaf node

### 10.17.2 Advantages

- Because nodes are not overlapped with each other, point query performance benefits since all spatial regions are covered by at most one node.
- A single path is followed and fewer nodes are visited than with the R-tree

### 10.17.3 Disadvantages

- Since rectangles are duplicated, an R+ tree can be larger than an R tree built on same data set.
- Construction and maintenance of R+ trees is more complex than the construction and maintenance of R trees and other variants of the R tree.

### 10.17.4 Notes

[1] Härdter, Rahm, Theo, Erhard (2007). *Datenbanksysteme*. (2., überarb. Aufl. ed.). Berlin [etc.]: Gardners Books. pp. 285, 286. ISBN 3-540-42133-5.

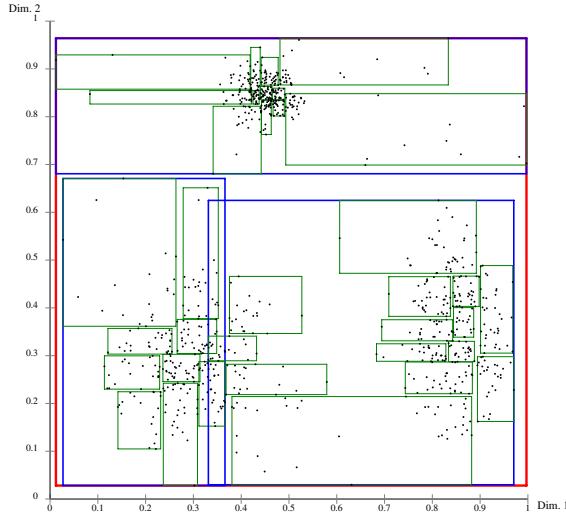
### 10.17.5 References

- T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In VLDB, 1987.

## 10.18 R\* tree

**R\*-trees** are a variant of **R-trees** used for indexing spatial information. R\*-trees have slightly higher construction cost than standard R-trees, as the data may need to be reinserted; but the resulting tree will usually have a better query performance. Like the standard R-tree, it can store both point and spatial data. It was proposed by Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger in 1990.<sup>[1]</sup>

### 10.18.1 Difference between R\*-trees and R-trees



R\*-Tree built by repeated insertion (in ELKI). There is little overlap in this tree, resulting in good query performance. Red and blue MBRs are index pages, green MBRs are leaf nodes.

Minimization of both coverage and overlap is crucial to the performance of R-trees. Overlap means that, on data query or insertion, more than one branch of the tree needs to be expanded (due to the way data is being split in regions which may overlap). A minimized coverage improves pruning performance, allowing to exclude whole pages from search more often, in particular for negative range queries.

The R\*-tree attempts to reduce both, using a combination of a revised node split algorithm and the concept of forced reinsertion at node overflow. This is based on the observation that R-tree structures are highly susceptible to the order in which their entries are inserted, so an insertion-built (rather than bulk-loaded) structure is likely to be sub-optimal. Deletion and reinsertion of entries allows them to “find” a place in the tree that may be more appropriate than their original location.

When a node overflows, a portion of its entries are removed from the node and reinserted into the tree. (In order to avoid an indefinite cascade of reinsertions caused by subsequent node overflow, the reinsertion routine may

be called only once in each level of the tree when inserting any one new entry.) This has the effect of producing more well-clustered groups of entries in nodes, reducing node coverage. Furthermore, actual node splits are often postponed, causing average node occupancy to rise. Re-insertion can be seen as a method of incremental tree optimization triggered on node overflow.

### 10.18.2 Performance

- Improved split heuristic produces pages that are more rectangular and thus better for many applications.
- Reinsertion method optimizes the existing tree, but increases complexity.
- Efficiently supports point and spatial data at the same time.

### 10.18.3 Algorithm and complexity

- The R\*-tree uses the same algorithm as the regular R-tree for query and delete operations.
- When inserting, the R\*-tree uses a combined strategy. For leaf nodes, overlap is minimized, while for inner nodes, enlargement and area are minimized.
- When splitting, the R\*-tree uses a topological split that chooses a split axis based on perimeter, then minimizes overlap.
- In addition to an improved split strategy, the R\*-tree also tries to avoid splits by reinserting objects and subtrees into the tree, inspired by the concept of balancing a B-tree.

Obviously, worst case query and delete complexity are thus identical to the R-Tree. The insertion strategy to the R\*-tree is with  $\mathcal{O}(M \log M)$  more complex than the linear split strategy ( $\mathcal{O}(M)$ ) of the R-tree, but less complex than the quadratic split strategy ( $\mathcal{O}(M^2)$ ) for a page size of  $M$  objects and has little impact on the total complexity. The total insert complexity is still comparable to the R-tree: reinsertions affect at most one branch of the tree and thus  $\mathcal{O}(\log n)$  reinsertions, comparable to performing a split on a regular R-tree. So on overall, the complexity of the R\*-tree is the same as that of a regular R-tree.

An implementation of the full algorithm must address many corner cases and tie situations not discussed here.

### 10.18.4 References

- [1] Beckmann, N.; Kriegel, H. P.; Schneider, R.; Seeger, B. (1990). “The R\*-tree: an efficient and robust access method for points and rectangles”. *Proceedings of the 1990 ACM SIGMOD international conference on Management of data - SIGMOD '90* (PDF). p. 322. doi:10.1145/93597.98741. ISBN 0897913655.

[2] Guttman, A. (1984). “R-Trees: A Dynamic Index Structure for Spatial Searching”. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84* (PDF). p. 47. doi:10.1145/602259.602266. ISBN 0897911288.

- [3] Ang, C. H.; Tan, T. C. (1997). “New linear node splitting algorithm for R-trees”. In Scholl, Michel; Voisard, Agnès. *Proceedings of the 5th International Symposium on Advances in Spatial Databases (SSD '97), Berlin, Germany, July 15–18, 1997*. Lecture Notes in Computer Science **1262**. Springer. pp. 337–349. doi:10.1007/3-540-63238-7\_38.

### 10.18.5 External links

Libraries containing R\*-trees:

- Boost.Geometry rtree documentation (C++, maybe R-tree only)
- ELKI R\*-tree package documentation (Java)
- Spatial Index Library (C++)
- SQLite R\*-tree module (C)
- TPIE Library (C++)
- XXL Library (Java, maybe R-tree only)

Demonstration and example code:

- A header-only C++ R\* Tree Implementation (probably buggy and it does not generate a R\*-tree, but a freely defined (by the code author) variation of the original definition)
- A 2D R\*-tree implementation (C/C++)
- rtree (Java, R-tree and R\*-tree. Incomplete: no paged disk backend, no reinsertions)
- R-tree Demo Applet (requires Java)

## 10.19 Hilbert R-tree

**Hilbert R-tree**, an R-tree variant, is an index for multi-dimensional objects like lines, regions, 3-D objects, or high-dimensional feature-based parametric objects. It can be thought of as an extension to B+-tree for multi-dimensional objects.

The performance of R-trees depends on the quality of the algorithm that clusters the data rectangles on a node. Hilbert R-trees use space-filling curves, and specifically

the Hilbert curve, to impose a linear ordering on the data rectangles.

There are two types of Hilbert R-trees, one for static databases, and one for dynamic databases. In both cases Hilbert space-filling curves are used to achieve better ordering of multidimensional objects in the node. This ordering has to be ‘good’, in the sense that it should group ‘similar’ data rectangles together, to minimize the area and perimeter of the resulting **minimum bounding rectangles** (MBRs). Packed Hilbert R-trees are suitable for static databases in which updates are very rare or in which there are no updates at all.

The dynamic Hilbert R-tree is suitable for dynamic databases where insertions, deletions, or updates may occur in real time. Moreover, dynamic Hilbert R-trees employ flexible deferred splitting mechanism to increase the space utilization. Every node has a well defined set of sibling nodes. By adjusting the split policy the Hilbert R-tree can achieve a degree of space utilization as high as is desired. This is done by proposing an ordering on the R-tree nodes. The Hilbert R-tree sorts rectangles according to the **Hilbert value** of the center of the rectangles (i.e., MBR). (The Hilbert value of a point is the length of the Hilbert curve from the origin to the point.) Given the ordering, every node has a well-defined set of sibling nodes; thus, deferred splitting can be used. By adjusting the split policy, the Hilbert R-tree can achieve as high utilization as desired. To the contrary, other R-tree variants have no control over the space utilization.

### 10.19.1 The basic idea

Although the following example is for a static environment, it explains the intuitive principles for good R-tree design. These principles are valid for both static and dynamic databases. Roussopoulos and Leifker proposed a method for building a packed R-tree that achieves almost 100% space utilization. The idea is to sort the data on the x or y coordinate of one of the corners of the rectangles. Sorting on any of the four coordinates gives similar results. In this discussion points or rectangles are sorted on the x coordinate of the lower left corner of the rectangle. In the discussion below the Roussopoulos and Leifker’s method is referred to as the *lowx packed R-tree*. The sorted list of rectangles is scanned; successive rectangles are assigned to the same R-tree leaf node until that node is full; a new leaf node is then created and the scanning of the sorted list continues. Thus, the nodes of the resulting R-tree will be fully packed, with the possible exception of the last node at each level. Thus, the utilization is  $\approx 100\%$ . Higher levels of the tree are created in a similar way.

Figure 1 highlights the problem of the *lowx packed R-tree*. Figure 1[Right] shows the leaf nodes of the R-tree that the *lowx* packing method will create for the points of Figure 1 [Left]. The fact that the resulting father nodes cover little area explains why the *lowx* packed R-tree

achieves excellent performance for point queries. However, the fact that the fathers have large perimeters, explains the degradation of performance for region queries. This is consistent with the analytical formulas for R-tree performance.<sup>[1]</sup> Intuitively, the packing algorithm should ideally assign nearby points to the same leaf node. Ignorance of the y coordinate by the *lowx* packed R-tree tends to violate this empirical rule.

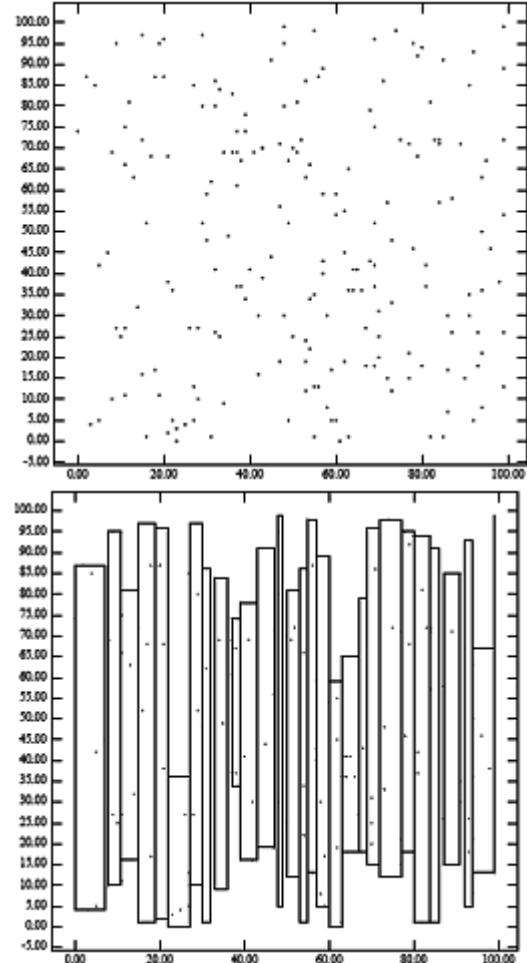


Figure 1: [Left] 200 points uniformly distributed; [Right] MBR of nodes generated by the ‘*lowx packed R-tree*’ algorithm

This section describes two variants of the Hilbert R-trees. The first index is suitable for the static database in which updates are very rare or in which there are no updates at all. The nodes of the resulting R-tree will be fully packed, with the possible exception of the last node at each level. Thus, the space utilization is  $\approx 100\%$ ; this structure is called a packed Hilbert R-tree. The second index, called a Dynamic Hilbert R-tree, supports insertions and deletions, and is suitable for a dynamic environment.

### 10.19.2 Packed Hilbert R-trees

The following provides a brief introduction to the Hilbert curve. The basic Hilbert curve on a  $2 \times 2$  grid, denoted

by  $H_1$  is shown in Figure 2. To derive a curve of order  $i$ , each vertex of the basic curve is replaced by the curve of order  $i - 1$ , which may be appropriately rotated and/or reflected. Figure 2 also shows the Hilbert curves of order two and three. When the order of the curve tends to infinity, like other space filling curves, the resulting curve is a fractal, with a fractal dimension of two.<sup>[1][2]</sup> The Hilbert curve can be generalized for higher dimensionalities. Algorithms for drawing the two-dimensional curve of a given order can be found in<sup>[3]</sup> and.<sup>[2]</sup> An algorithm for higher dimensionalities is given in.<sup>[4]</sup>

The path of a space filling curve imposes a linear ordering on the grid points; this path may be calculated by starting at one end of the curve and following the path to the other end. The actual coordinate values of each point can be calculated. However, for the Hilbert curve this is much harder than for example the **Z-order curve**. Figure 2 shows one such ordering for a 4x4 grid (see curve  $H_2$ ). For example, the point (0,0) on the  $H_2$  curve has a Hilbert value of 0, while the point (1,1) has a Hilbert value of 2.

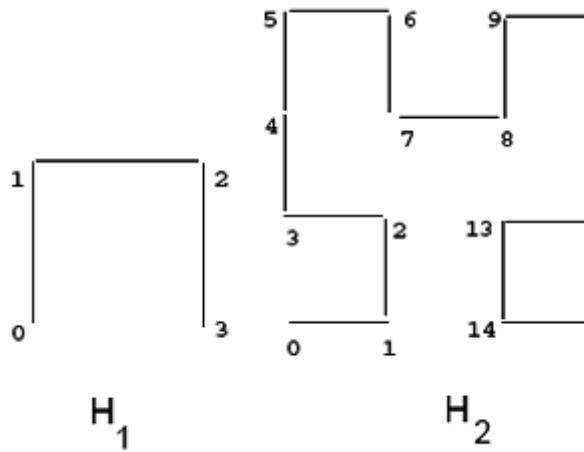


Figure 2: Hilbert curves of order 1, 2, and 3

The Hilbert curve imposes a linear ordering on the data rectangles and then traverses the sorted list, assigning each set of  $C$  rectangles to a node in the R-tree. The final result is that the set of data rectangles on the same node will be close to each other in the linear ordering, and most likely in the native space; thus, the resulting R-tree nodes will have smaller areas. Figure 2 illustrates the intuitive reasons why our Hilbert-based methods will result in good performance. The data is composed of points (the same points as given in Figure 1). By grouping the points according to their Hilbert values, the MBRs of the resulting R-tree nodes tend to be small square-like rectangles. This indicates that the nodes will likely have small area and small perimeters. Small area values result in good performance for point queries; small area and small perimeter values lead to good performance for larger queries.

### Algorithm Hilbert-Pack

(packs rectangles into an R-tree)

Step 1. Calculate the Hilbert value for each data rectangle  
 Step 2. Sort data rectangles on ascending Hilbert values  
 Step 3. /\* Create leaf nodes (level l=0) \*/

- While (there are more rectangles)
  - generate a new R-tree node
  - assign the next  $C$  rectangles to this node

Step 4. /\* Create nodes at higher level (l + 1) \*/

- While (there are > 1 nodes at level l)
  - sort nodes at level  $l \geq 0$  on ascending creation time
  - repeat Step 3

The assumption here is that the data are static or the frequency of modification is low. This is a simple heuristic for constructing an R-tree with 100% space utilization which at the same time will have as good response time as possible.

### 10.19.3 Dynamic Hilbert R-trees

The performance of R-trees depends on the quality of the algorithm that clusters the data rectangles on a node. Hilbert R-trees use space-filling curves, and specifically the Hilbert curve, to impose a linear ordering on the data rectangles. The Hilbert value of a rectangle is defined as the Hilbert value of its center.<sup>[3]</sup>

### Tree structure

The Hilbert R-tree has the following structure. A leaf node contains at most  $C_l$  entries each of the form  $(R, \text{obj\_id})$  where  $C_l$  is the capacity of the leaf,  $R$  is the MBR of the real object  $(x_{\text{low}}, x_{\text{high}}, y_{\text{low}}, y_{\text{high}})$  and  $\text{obj\_id}$  is a pointer to the object description record. The main difference between the Hilbert R-tree and the R\*-tree<sup>[5]</sup> is that non-leaf nodes also contain information about the LHV (Largest Hilbert Value). Thus, a non-leaf node in the Hilbert R-tree contains at most  $C_n$  entries of the form  $(R, \text{ptr}, \text{LHV})$  where  $C_n$  is the capacity of a non-leaf node,  $R$  is the MBR that encloses all the children of that node,  $\text{ptr}$  is a pointer to the child node, and  $\text{LHV}$  is the largest Hilbert value among the data rectangles enclosed by  $R$ . Notice that since the non-leaf node picks one of the Hilbert values of the children to be the value of its own LHV, there is not extra cost for calculating the Hilbert values of the MBR of non-leaf nodes. Figure 3 illustrates some rectangles organized in a Hilbert R-tree. The Hilbert values of the centers are the numbers near

the 'x' symbols (shown only for the parent node 'II'). The LHV's are in [brackets]. Figure 4 shows how the tree of Figure 3 is stored on the disk; the contents of the parent node 'I' are shown in more detail. Every data rectangle in node 'T' has a Hilbert value  $v \leq 33$ ; similarly every rectangle in node 'II' has a Hilbert value greater than 33 and  $\leq 107$ , etc.

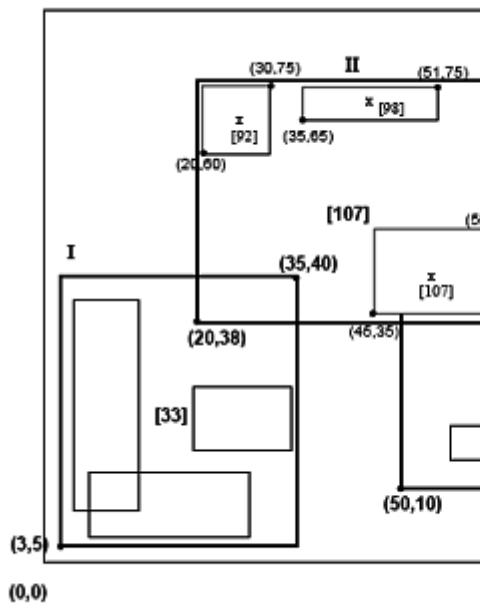


Figure 3: Data rectangles organized in a Hilbert R-tree (Hilbert values and LHV's are in Brackets)

A plain R-tree splits a node on overflow, creating two nodes from the original one. This policy is called a 1-to-2 splitting policy. It is possible also to defer the split, waiting until two nodes split into three. Note that this is similar to the B\*-tree split policy. This method is referred to as the 2-to-3 splitting policy. In general, this can be extended to s-to-(s+1) splitting policy; where s is the order of the splitting policy. To implement the order-s splitting policy, the overflowing node tries to push some of its entries to one of its  $s - 1$  siblings; if all of them are full, then s-to-(s+1) split need to be done. The  $s - 1$  siblings are called the cooperating siblings. Next, the algorithms for searching, insertion, and overflow handling are described in details.

## Searching

The searching algorithm is similar to the one used in other R-tree variants. Starting from the root, it descends the tree and examines all nodes that intersect the query rectangle. At the leaf level, it reports all entries that intersect the query window  $w$  as qualified data items.

**Algorithm Search(node Root, rect w):**

### S1. Search nonleaf nodes:

Invoke Search for every entry whose MBR intersects the query window  $w$ .

## S2. Search leaf nodes:

Report all entries that intersect the query window  $w$  as candidates. **(100, 100)**

| LHV                     | XL | YL | XH | YH | LHV                        | XL | YL | XH | YH | LHV |
|-------------------------|----|----|----|----|----------------------------|----|----|----|----|-----|
| 33                      | 3  | 6  | 36 | 40 | 107                        | 20 | 38 | 56 | 75 | 206 |
| le Hilbert values <= 33 |    |    |    |    | data Hilbert values <= 107 |    |    |    |    |     |

Figure 4: The file structure for the Hilbert R-tree

## Insertion

To insert a new rectangle  $r$  in the Hilbert R-tree, the Hilbert value  $h$  of the center of the new rectangle is used as a key. At each level the node with the minimum LHV value greater than  $h$  of all its siblings is chosen. When a leaf node is reached, the rectangle  $r$  is inserted in its correct order according to  $h$ . After a new rectangle is inserted in a leaf node  $N$ , **AdjustTree** is called to fix the MBR and LHV values in the upper-level nodes.

**Algorithm Insert(node Root, rect r):** /\* Inserts a new rectangle r in the Hilbert R-tree. h is the Hilbert value of the rectangle\*/

11. Find the appropriate leaf node:

Invoke ChooseLeaf( $r, h$ ) to select a leaf node  $L$  in which to place  $r$ .

I2. Insert r in a leaf node L:

If  $L$  has an empty slot, insert  $r$  in  $L$  in the appropriate place according to the Hilbert order and return.

If  $L$  is full, invoke  $\text{HandleOverflow}(L, r)$ , which will return new leaf if split was inevitable,

### I3. Propagate changes upward:

- Form a set S that contains L, its cooperating siblings and the new leaf (if any)  
Invoke **AdjustTree(S)**.
- I4. Grow tree taller:
- If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.
- Algorithm ChooseLeaf(rect r, int h):**  
/\* Returns the leaf node in which to place a new rectangle r. \*/
- C1. Initialize:
- Set N to be the root node.
- C2. Leaf check:
- If N is a leaf\_ return N.
- C3. Choose subtree:
- If N is a non-leaf node, choose the entry (R, ptr, LHV)  
with the minimum LHV value greater than h.
- C4. Descend until a leaf is reached:
- Set N to the node pointed by ptr and repeat from C2.
- Algorithm AdjustTree(set S):**  
/\* S is a set of nodes that contains the node being updated, its cooperating siblings (if overflow has occurred) and the newly created node NN (if split has occurred). The routine ascends from the leaf level towards the root, adjusting MBR and LHV of nodes that cover the nodes in S. It propagates splits (if any) \*/
- A1. If root level is reached, stop.
- A2. Propagate node split upward:
- Let Np be the parent node of N.  
If N has been split, let NN be the new node.  
Insert NN in Np in the correct order according to its Hilbert value if there is room. Otherwise, invoke **HandleOverflow(Np, NN)**.  
If Np is split, let PP be the new node.
- A3. Adjust the MBR's and LHV's in the parent level:  
Let P be the set of parent nodes for the nodes in S.  
Adjust the corresponding MBR's and LHV's of the nodes in P appropriately.
- A4. Move up to next level:  
Let S become the set of parent nodes P, with NN = PP, if Np was split.  
repeat from A1.
- Deletion**
- In the Hilbert R-tree, there is no need to re-insert orphaned nodes whenever a father node underflows. Instead, keys can be borrowed from the siblings or the underflowing node is merged with its siblings. This is possible because the nodes have a clear ordering (according to Largest Hilbert Value, LHV); in contrast, in R-trees there is no such concept concerning sibling nodes. Notice that deletion operations require s cooperating siblings, while insertion operations require s - 1 siblings.
- Algorithm Delete(r):**
- D1. Find the host leaf:
- Perform an exact match search to find the leaf node L  
that contains r.
- D2. Delete r :
- Remove r from node L.
- D3. If L underflows
- borrow some entries from s cooperating siblings.  
if all the siblings are ready to underflow.  
merge s + 1 to s nodes,  
adjust the resulting nodes.
- D4. Adjust MBR and LHV in parent levels.
- form a set S that contains L and its cooperating siblings (if underflow has occurred).  
invoke **AdjustTree(S)**.

### Overflow handling

The overflow handling algorithm in the Hilbert R-tree treats the overflowing nodes either by moving some of the entries to one of the  $s - 1$  cooperating siblings or by splitting  $s$  nodes into  $s + 1$  nodes.

#### Algorithm HandleOverflow(node N, rect r):

/\* return the new node if a split occurred. \*/

H1. Let  $\epsilon$  be a set that contains all the entries from N

and its  $s - 1$  cooperating siblings.

H2. Add  $r$  to  $\epsilon$ .

H3. If at least one of the  $s - 1$  cooperating siblings is not full,

distribute  $\epsilon$  evenly among the  $s$  nodes according to Hilbert values.

H4. If all the  $s$  cooperating siblings are full,

create a new node NN and

distribute  $\epsilon$  evenly among the  $s + 1$  nodes according

to Hilbert values

return NN.

### 10.19.4 Notes

- [1] I. Kamel and C. Faloutsos, On Packing R-trees, Second International ACM Conference on Information and Knowledge Management (CIKM), pages 490–499, Washington D.C., 1993.
- [2] H. Jagadish. Linear clustering of objects with multiple attributes. In Proc. of ACM SIGMOD Conf., pages 332–342, Atlantic City, NJ, May 1990.
- [3] J. Griffiths. An algorithm for displaying a class of space-filling curves, Software-Practice and Experience 16(5), 403–411, May 1986.
- [4] T. Bially. Space-filling curves. Their generation and their application to bandwidth reduction. IEEE Trans. on Information Theory. IT15(6), 658–664, November 1969.
- [5] Beckmann, N.; Kriegel, H. P.; Schneider, R.; Seeger, B. (1990). “The R\*-tree: an efficient and robust access method for points and rectangles”. *Proceedings of the 1990 ACM SIGMOD international conference on Management of data - SIGMOD '90* (PDF). p. 322. doi:10.1145/93597.98741. ISBN 0897913655.

### 10.19.5 References

- I. Kamel and C. Faloutsos. Parallel R-Trees. In Proc. of ACM SIGMOD Conf., pages 195–204 San Diego, CA, June 1992. Also available as Tech. Report UMIACS TR 92-1, CS-TR-2820.

- I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In Proc. of VLDB Conf., pages 500–509, Santiago, Chile, September 1994. Also available as Tech\_ Report UMIACS TR 93-12.1 CS-TR-3032.1.
- N. Koudas, C. Faloutsos and I. Kamel. Declustering Spatial Databases on a Multi-Computer Architecture, International Conference on Extending Database Technology (EDBT), pages 592–614, 1996.
- N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using Packed R-trees. In Proc. of ACM SIGMOD, pages 17–31, Austin, TX, May 1985.
- M. Schroeder. Fractals, Chaos, Power Laws: Minutes From an Infinite Paradise. W.H. Freeman and Company, NY, 1991.
- T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: a dynamic index for multi-dimensional objects. In Proc. 13th International Conference on VLDB, pages 507–518, England, September 1987.

## 10.20 X-tree

This article is about a tree data structure for storing data in multiple dimensions. For the XTree file manager, see [XTree](#).

In computer science, an **X-tree** is an index tree structure based on the **R-tree** used for storing data in many dimensions. It differs from R-trees, R+-trees and R\*-trees because it emphasizes prevention of overlap in the bounding boxes, which increasingly becomes a problem in high dimensions. In cases where nodes cannot be split without preventing overlap, the node split will be deferred, resulting in **super-nodes**. In extreme cases, the tree will linearize, which defends against worst-case behaviors observed in some other data structures.

### 10.20.1 References

- Berchtold, Stefan; Keim, Daniel A.; Kriegel, Hans-Peter (1996). “The X-tree: An Index Structure for High-Dimensional Data”. *Proceedings of the 22nd VLDB Conference* (Mumbai, India): 28–39.

## 10.21 Metric tree

This article is about the data structure. For the type of metric space, see [Real tree](#).

A **metric tree** is any **tree data structure** specialized to index data in **metric spaces**. Metric trees exploit properties of metric spaces such as the **triangle inequality** to make accesses to the data more efficient. Examples include the **M-tree**, **vp-trees**, **cover trees**, **MVP Trees**, and **bk trees**.<sup>[1]</sup>

### 10.21.1 Multidimensional search

Most algorithms and data structures for searching a dataset are based on the classical **binary search algorithm**, and generalizations such as the **k-d tree** or **range tree** work by interleaving the **binary search algorithm** over the separate coordinates and treating each spatial coordinate as an independent search constraint. These data structures are well-suited for **range query** problems asking for every point  $(x, y)$  that satisfies  $\min_x \leq x \leq \max_x$  and  $\min_y \leq y \leq \max_y$ .

A limitation of these multidimensional search structures is that they are only defined for searching over objects that can be treated as vectors. They aren't applicable for the more general case in which the algorithm is given only a collection of objects and a function for measuring the distance or similarity between two objects. If, for example, someone were to create a function that returns a value indicating how similar one image is to another, a natural algorithmic problem would be to take a dataset of images and find the ones that are similar according to the function to a given query image.

### 10.21.2 Metric data structures

If there is no structure to the similarity measure then a **brute force search** requiring the comparison of the query image to every image in the dataset is the best that can be done. If, however, the similarity function satisfies the **triangle inequality** then it is possible to use the result of each comparison to prune the set of candidates to be examined.

The first article on metric trees, as well as the first use of the term “metric tree”, published in the open literature was by **Jeffrey Uhlmann** in 1991.<sup>[2]</sup> Other researchers were working independently on similar data structures. In particular, Peter Yianilos claimed to have independently discovered the same method, which he called a **vantage point tree** (VP-tree).<sup>[3]</sup> The research on metric tree data structures blossomed in the late 1990s and included an examination by Google co-founder **Sergey Brin** of their use for very large databases.<sup>[4]</sup> The first textbook on metric data structures was published in 2006.<sup>[1]</sup>

### 10.21.3 References

- [1] Samet, Hanan (2006). *Foundations of multidimensional and metric data structures*. Morgan Kaufmann. ISBN 978-0-12-369446-1.

- [2] Uhlmann, Jeffrey (1991). “Satisfying General Proximity/Similarity Queries with Metric Trees”. *Information Processing Letters* **40** (4). doi:10.1016/0020-0190(91)90074-r.
- [3] Yianilos, Peter N. (1993). “Data structures and algorithms for nearest neighbor search in general metric spaces”. *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics Philadelphia, PA, USA. pp. 311–321. pny93. Retrieved 2008-08-22.
- [4] Brin, Sergey (1995). “Near Neighbor Search in Large Metric Spaces”. *21st International Conference on Very Large Data Bases (VLDB)*.

## 10.22 Vp-tree

A **vantage-point tree**, or **VP tree** is a **BSP tree** that segregates data in a **metric space** by choosing a position in the space (the “vantage point”) and dividing the data points into two partitions: those that are nearer to the vantage point than a threshold, and those that are not. By repeatedly applying this procedure to partition the data into smaller and smaller sets, a **tree data structure** is created where neighbors in the tree are likely to be neighbors in the space.<sup>[1]</sup> Peter Yianilos claimed that the VP-tree was discovered independently by him (Peter Yianilos) and by Jeffrey Uhlmann.<sup>[1]</sup> Yet, Uhlmann published this method before Yianilos in 1991.<sup>[2]</sup> Uhlmann called the data structure a **metric tree**, the name VP-tree was proposed by Yianilos. Vantage point trees have been generalized to non-metric spaces using Bregman divergences by Nielsen et al.<sup>[3]</sup>

This iterative partitioning process is similar to that of a **k-d tree**, but uses circular (or spherical, hyperspherical, etc.) rather than rectilinear partitions. In 2D Euclidean space, this can be visualized as a series of circles segregating the data.

The VP tree is particularly useful in dividing data in a non-standard metric space into a **BSP tree**.

### 10.22.1 Understanding a VP tree

The way a VP tree stores data can be represented by a circle.<sup>[4]</sup> First, understand that each **node** of this **tree** contains an input point and a radius. All the left children of a given **node** are the points inside the circle and all the right children of a given **node** are outside of the circle. The **tree** itself does not need to know any other information about what is being stored. All it needs is the distance function that satisfies the properties of the **metric space**.<sup>[4]</sup> Just imagine a circle with a radius. The left children are all located inside the circle and the right children are located outside the circle.

### 10.22.2 Searching through a VP tree

Suppose there is a need to find the two nearest targets from a given point (The point will be placed relatively close to distance). Since there are no points yet, it is assumed that the middle point (center) is the closest target. Now a variable is needed to keep track of the distance  $X$  (This will change if another distance is greater). To determine whether we go to left or right child will depend on the given point.<sup>[4]</sup> If the point is closer to the radius than the outer shell, search the left child. Otherwise, search the right child. Once the point (the neighbor) is found, the variable will be updated because the distance has increased.

From here, all the points within the radius have been considered. To complete the search, we will now find the closest point outside the radius (right child) and determine the second neighbor. The search method will be the same, but it will be for the right child.<sup>[4]</sup>

### 10.22.3 Advantages of a VP tree

1. Instead of inferring multidimensional points for domain before the index being built, we build the index directly based on the distance.<sup>[4]</sup> Doing this, avoids pre-processing steps.
2. Updating a VP tree is relatively easy compared to the fast-map approach. For fast maps, after inserting or deleting data, there will come a time when fast-map will have to rescan itself. That takes up too much time and it is unclear to know when the rescanning will start.
3. Distance based methods are flexible. It is “able to index objects that are represented as feature vectors of a fixed number of dimensions.”<sup>[4]</sup>

### 10.22.4 Implementation examples

1. In Python
2. In C
3. In Java
4. In Java (alternative implementation)
5. In JavaScript

### 10.22.5 References

- [1] Yianilos (1993). “Data structures and algorithms for nearest neighbor search in general metric spaces”. *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics Philadelphia, PA, USA. pp. 311–321. pny93. Retrieved 2008-08-22.

[2] Uhlmann, Jeffrey (1991). “Satisfying General Proximity/Similarity Queries with Metric Trees”. *Information Processing Letters* **40** (4). doi:10.1016/0020-0190(91)90074-r.

[3] Nielsen, Frank (2009). “Bregman vantage point trees for efficient nearest Neighbor Queries”. *Proceedings of Multimedia and Exp (ICME)*. IEEE. pp. 878–881.

[4] Fu, Ada Wai-chee; Polly Mei-shuen Chan, Yin-Ling Cheung, Yiu Sang Moon (2000). “Dynamic vp-tree indexing for  $n$ -nearest neighbor search given pair-wise distances”. *The VLDB Journal — The International Journal on Very Large Data Bases*. Springer-Verlag New York, Inc. Secaucus, NJ, USA. pp. 154–173. vp. Retrieved 2012-10-02.

### 10.22.6 External links

- Understanding VP Trees

### 10.22.7 Further reading

- Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces

## 10.23 BK-tree

A **BK-tree** is a metric tree suggested by Walter Austin Burkhard and Robert M. Keller specifically adapted to discrete metric spaces. For simplicity, let us consider **integer** discrete metric  $d(x, y)$ . Then, BK-tree is defined in the following way. An arbitrary element  $a$  is selected as root node. The root node may have zero or more subtrees. The  $k$ -th subtree is recursively built of all elements  $b$  such that  $d(a, b) = k$ . BK-trees can be used for approximate string matching in a dictionary .

### 10.23.1 See also

- Levenshtein distance – the distance metric commonly used when building a BK-tree
- Damerau–Levenshtein distance – a modified form of Levenshtein distance that allows transpositions

### 10.23.2 References

- <sup>4</sup> W. Burkhard and R. Keller. Some approaches to best-match file searching, CACM, 1973
- <sup>4</sup> R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed queries trees. In M. Crochemore and D. Gusfield, editors, 5th Combinatorial Pattern Matching, LNCS 807, pages 198–212, Asilomar, CA, June 1994.

- ^ Ricardo Baeza-Yates and Gonzalo Navarro. Fast Approximate String Matching in a Dictionary. Proc. SPIRE'98

### 10.23.3 External links

- A BK-tree implementation in Common Lisp with test results and performance graphs.
- An explanation of BK-Trees and their relationship to metric spaces
- An explanation of BK-Trees with an implementation in C#

# Chapter 11

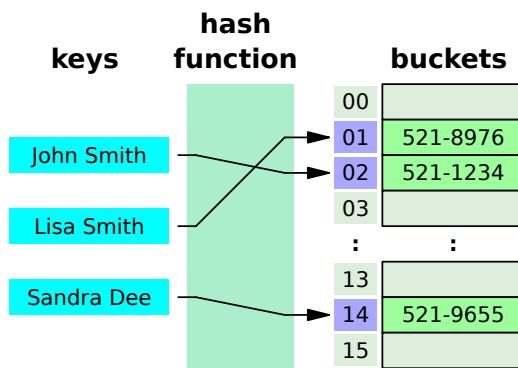
## Hashes

### 11.1 Hash table

Not to be confused with **Hash list** or **Hash tree**.

“Rehash” redirects here. For the *South Park* episode, see [Rehash \(South Park\)](#). For the IRC command, see [List of Internet Relay Chat commands § REHASH](#).

In computing, a **hash table** (**hash map**) is a data struc-



*A small phone book as a hash table*

ture used to implement an **associative array**, a structure that can map **keys** to **values**. A hash table uses a **hash function** to compute an *index* into an array of **buckets** or **slots**, from which the correct value can be found.

Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that **hash collisions**—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of **instructions**) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at (amortized<sup>[2]</sup>) constant average cost per operation.<sup>[3][4]</sup>

In many situations, hash tables turn out to be more efficient than **search trees** or any other **table** lookup structure. For this reason, they are widely used in many kinds

of computer software, particularly for associative arrays, database indexing, caches, and sets.

#### 11.1.1 Hashing

Main article: [Hash function](#)

The idea of hashing is to distribute the entries (key/value pairs) across an array of *buckets*. Given a key, the algorithm computes an *index* that suggests where the entry can be found:

$$\text{index} = f(\text{key}, \text{array\_size})$$

Often this is done in two steps:

$$\text{hash} = \text{hashfunc}(\text{key}) \quad \text{index} = \text{hash \% array\_size}$$

In this method, the *hash* is independent of the array size, and it is then *reduced* to an index (a number between 0 and *array\_size* – 1) using the **modulo operator** (%).

In the case that the array size is a **power of two**, the remainder operation is reduced to **masking**, which improves speed, but can increase problems with a poor hash function.

#### Choosing a good hash function

A good hash function and implementation algorithm are essential for good hash table performance, but may be difficult to achieve.

A basic requirement is that the function should provide a **uniform distribution** of hash values. A non-uniform distribution increases the number of collisions and the cost of resolving them. Uniformity is sometimes difficult to ensure by design, but may be evaluated empirically using statistical tests, e.g., a Pearson’s chi-squared test for discrete uniform distributions.<sup>[5][6]</sup>

The distribution needs to be uniform only for table sizes that occur in the application. In particular, if one uses dynamic resizing with exact doubling and halving of the table size *s*, then the hash function needs to be uniform only when *s* is a **power of two**. On the other hand, some hashing algorithms provide uniform hashes only when *s* is a **prime number**.<sup>[7]</sup>

For open addressing schemes, the hash function should also avoid *clustering*, the mapping of two or more keys to consecutive slots. Such clustering may cause the lookup cost to skyrocket, even if the load factor is low and collisions are infrequent. The popular multiplicative hash<sup>[3]</sup> is claimed to have particularly poor clustering behavior.<sup>[7]</sup>

Cryptographic hash functions are believed to provide good hash functions for any table size  $s$ , either by modulo reduction or by *bit masking*. They may also be appropriate if there is a risk of malicious users trying to *sabotage* a network service by submitting requests designed to generate a large number of collisions in the server's hash tables. However, the risk of sabotage can also be avoided by cheaper methods (such as applying a secret *salt* to the data, or using a *universal hash function*).

### Perfect hash function

If all keys are known ahead of time, a *perfect hash function* can be used to create a perfect hash table that has no collisions. If *minimal perfect hashing* is used, every location in the hash table can be used as well.

Perfect hashing allows for *constant time* lookups in the worst case. This is in contrast to most chaining and open addressing methods, where the time for lookup is low on average, but may be very large,  $O(n)$ , for some sets of keys.

### 11.1.2 Key statistics

A critical statistic for a hash table is called the *load factor*. This is simply the number of entries divided by the number of buckets, that is,  $n/k$  where  $n$  is the number of entries and  $k$  is the number of buckets.

If the load factor is kept reasonable, the hash table should perform well, provided the hashing is good. If the load factor grows too large, the hash table will become slow, or it may fail to work (depending on the method used). The expected *constant time* property of a hash table assumes that the load factor is kept below some bound. For a *fixed* number of buckets, the time for a lookup grows with the number of entries and therefore the desired constant time is not achieved.

Second to that, one can examine the variance of number of entries per bucket. For example, two tables both have 1000 entries and 1000 buckets; one has exactly one entry in each bucket, the other has all entries in the same bucket. Clearly the hashing is not working in the second one.

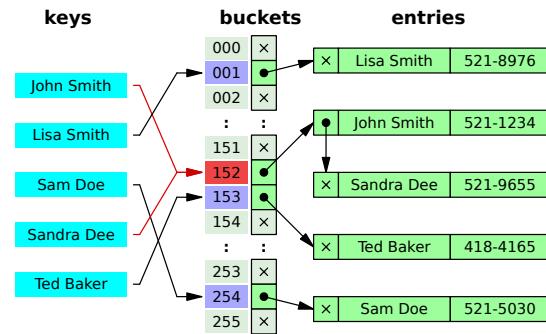
A low load factor is not especially beneficial. As the load factor approaches 0, the proportion of unused areas in the hash table increases, but there is not necessarily any reduction in search cost. This results in wasted memory.

### 11.1.3 Collision resolution

Hash *collisions* are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the *birthday problem* there is approximately a 95% chance of at least two of the keys being hashed to the same slot.

Therefore, most hash table implementations have some collision resolution strategy to handle such events. Some common strategies are described below. All these methods require that the keys (or pointers to them) be stored in the table, together with the associated values.

#### Separate chaining



Hash collision resolved by separate chaining.

In the method known as *separate chaining*, each bucket is independent, and has some sort of *list* of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.

In a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that. Therefore, structures that are efficient in time and space for these cases are preferred. Structures that are efficient for a fairly large number of entries per bucket are not needed or desirable. If these cases happen often, the hashing is not working well, and this needs to be fixed.

**Separate chaining with linked lists** Chained hash tables with *linked lists* are popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods.

The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is *sufficiently uniform*, the *average* cost of a lookup depends only on the average number of keys per bucket—that is, it is roughly proportional to the

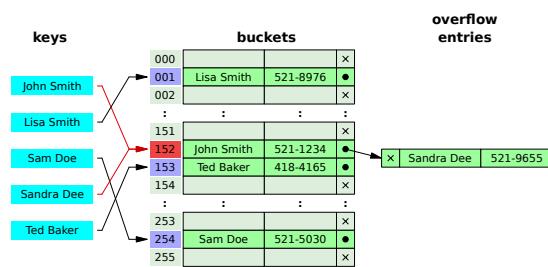
load factor.

For this reason, chained hash tables remain effective even when the number of table entries  $n$  is much higher than the number of slots. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list.

For separate-chaining, the worst-case scenario is when all entries are inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure. If the latter is a linear list, the lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number  $n$  of entries in the table.

The bucket chains are often implemented as [ordered lists](#), sorted by the key field; this choice approximately halves the average cost of unsuccessful lookups, compared to an unordered list. However, if some keys are much more likely to come up than others, an unordered list with [move-to-front heuristic](#) may be more effective. More sophisticated data structures, such as balanced search trees, are worth considering only if the load factor is large (about 10 or more), or if the hash distribution is likely to be very non-uniform, or if one must guarantee good performance even in a worst-case scenario. However, using a larger table and/or a better hash function may be even more effective in those cases.

Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space overhead of the next pointer in each entry record can be significant. An additional disadvantage is that traversing a linked list has poor [cache performance](#), making the processor cache ineffective.



Hash collision by separate chaining with head records in the bucket array.

**Separate chaining with list head cells** Some chaining implementations store the first record of each chain in the slot array itself.<sup>[4]</sup> The number of pointer traversals is decreased by one for most cases. The purpose is to increase cache efficiency of hash table access.

The disadvantage is that an empty bucket takes the same space as a bucket with one entry. To save space, such hash

tables often have about as many slots as stored entries, meaning that many slots have two or more entries.

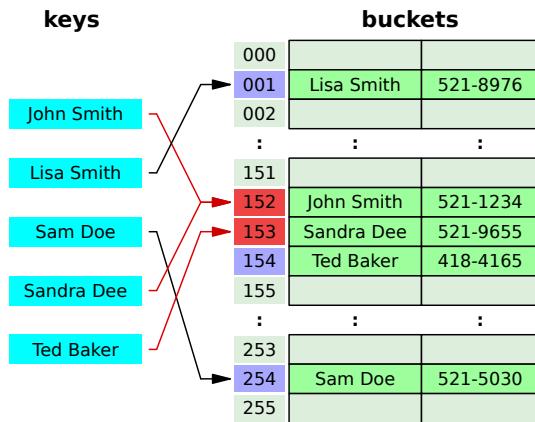
**Separate chaining with other structures** Instead of a list, one can use any other data structure that supports the required operations. For example, by using a [self-balancing tree](#), the theoretical worst-case time of common hash table operations (insertion, deletion, lookup) can be brought down to  $O(\log n)$  rather than  $O(n)$ . However, this approach is only worth the trouble and extra memory cost if long delays must be avoided at all costs (e.g., in a real-time application), or if one must guard against many entries hashed to the same slot (e.g., if one expects extremely non-uniform distributions, or in the case of web sites or other publicly accessible services, which are vulnerable to malicious key distributions in requests).

The variant called [array hash table](#) uses a [dynamic array](#) to store all the entries that hash to the same slot.<sup>[8][9][10]</sup> Each newly inserted entry gets appended to the end of the dynamic array that is assigned to the slot. The dynamic array is resized in an *exact-fit* manner, meaning it is grown only by as many bytes as needed. Alternative techniques such as growing the array by block sizes or *pages* were found to improve insertion performance, but at a cost in space. This variation makes more efficient use of [CPU caching](#) and the [translation lookaside buffer](#) (TLB), because slot entries are stored in sequential memory positions. It also dispenses with the next pointers that are required by linked lists, which saves space. Despite frequent array resizing, space overheads incurred by the operating system such as memory fragmentation were found to be small.

An elaboration on this approach is the so-called [dynamic perfect hashing](#),<sup>[11]</sup> where a bucket that contains  $k$  entries is organized as a perfect hash table with  $k^2$  slots. While it uses more memory ( $n^2$  slots for  $n$  entries, in the worst case and  $n \cdot k$  slots in the average case), this variant has guaranteed constant worst-case lookup time, and low amortized time for insertion.

## Open addressing

In another strategy, called [open addressing](#), all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.<sup>[12]</sup> The name “open addressing” refers to the fact that the location (“address”) of the item is not determined by its hash value. (This method is also called [closed hashing](#); it should not be confused with “open hashing” or “closed addressing” that usually



Hash collision resolved by open addressing with linear probing (interval=1). Note that “Ted Baker” has a unique hash, but nevertheless collided with “Sandra Dee”, that had previously collided with “John Smith”.

mean separate chaining.)

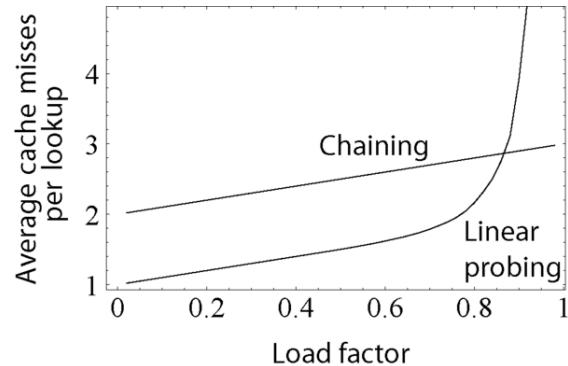
Well-known probe sequences include:

- **Linear probing**, in which the interval between probes is fixed (usually 1)
- **Quadratic probing**, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- **Double hashing**, in which the interval between probes is computed by another hash function

A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs.

Open addressing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are consecutive in the probe order. Using separate chaining, the only concern is that too many objects map to the *same* hash value; whether they are adjacent or nearby is completely irrelevant.

Open addressing only saves memory if the entries are small (less than four times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two words.



This graph compares the average number of cache misses required to look up elements in tables with chaining and linear probing. As the table passes the 80%-full mark, linear probing’s performance drastically degrades.

Open addressing avoids the time overhead of allocating each new entry record, and can be implemented even in the absence of a memory allocator. It also avoids the extra indirection required to access the first entry of each bucket (that is, usually the only one). It also has better **locality of reference**, particularly with linear probing. With small record sizes, these factors can yield better performance than chaining, particularly for lookups. Hash tables with open addressing are also easier to **serialize**, because they do not use pointers.

On the other hand, normal open addressing is a poor choice for large elements, because these elements fill entire **CPU cache lines** (negating the cache advantage), and a large amount of space is wasted on large empty table slots. If the open addressing table only stores references to elements (external storage), it uses space comparable to chaining even for large records but loses its speed advantage.

Generally speaking, open addressing is better used for hash tables with small records that can be stored within the table (internal storage) and fit in a cache line. They are particularly suitable for elements of one **word** or less. If the table is expected to have a high load factor, the records are large, or the data is variable-sized, chained hash tables often perform as well or better.

Ultimately, used sensibly, any kind of hash table algorithm is usually fast *enough*; and the percentage of a calculation spent in hash table code is low. Memory usage is rarely considered excessive. Therefore, in most cases the differences between these algorithms are marginal, and other considerations typically come into play.

**Coalesced hashing** A hybrid of chaining and open addressing, **coalesced hashing** links together chains of nodes within the table itself.<sup>[12]</sup> Like open addressing, it achieves space usage and (somewhat diminished) cache advantages over chaining. Like chaining, it does not exhibit clustering effects; in fact, the table can be efficiently filled to a high density. Unlike chaining, it cannot have

more elements than table slots.

**Cuckoo hashing** Another alternative open-addressing solution is **cuckoo hashing**, which ensures constant lookup time in the worst case, and constant amortized time for insertions and deletions. It uses two or more hash functions, which means any key/value pair could be in two or more locations. For lookup, the first hash function is used; if the key/value is not found, then the second hash function is used, and so on. If a collision happens during insertion, then the key is re-hashed with the second hash function to map it to another bucket. If all hash functions are used and there is still a collision, then the key it collided with is removed to make space for the new key, and the old key is re-hashed with one of the other hash functions, which maps it to another bucket. If that location also results in a collision, then the process repeats until there is no collision or the process traverses all the buckets, at which point the table is resized. By combining multiple hash functions with multiple cells per bucket, very high space utilization can be achieved.

**Hopscotch hashing** Another alternative open-addressing solution is **hopscotch hashing**,<sup>[13]</sup> which combines the approaches of **cuckoo hashing** and **linear probing**, yet seems in general to avoid their limitations. In particular it works well even when the load factor grows beyond 0.9. The algorithm is well suited for implementing a resizable concurrent hash table.

The hopscotch hashing algorithm works by defining a neighborhood of buckets near the original hashed bucket, where a given entry is always found. Thus, search is limited to the number of entries in this neighborhood, which is logarithmic in the worst case, constant on average, and with proper alignment of the neighborhood typically requires one cache miss. When inserting an entry, one first attempts to add it to a bucket in the neighborhood. However, if all buckets in this neighborhood are occupied, the algorithm traverses buckets in sequence until an open slot (an unoccupied bucket) is found (as in linear probing). At that point, since the empty bucket is outside the neighborhood, items are repeatedly displaced in a sequence of hops. (This is similar to cuckoo hashing, but with the difference that in this case the empty slot is being moved into the neighborhood, instead of items being moved out with the hope of eventually finding an empty slot.) Each hop brings the open slot closer to the original neighborhood, without invalidating the neighborhood property of any of the buckets along the way. In the end, the open slot has been moved into the neighborhood, and the entry being inserted can be added to it.

### Robin Hood hashing

One interesting variation on double-hashing collision resolution is **Robin Hood hashing**.<sup>[14][15]</sup> The idea is that a

new key may displace a key already inserted, if its probe count is larger than that of the key at the current position. The net effect of this is that it reduces worst case search times in the table. This is similar to ordered hash tables<sup>[16]</sup> except that the criterion for bumping a key does not depend on a direct relationship between the keys. Since both the worst case and the variation in the number of probes is reduced dramatically, an interesting variation is to probe the table starting at the expected successful probe value and then expand from that position in both directions.<sup>[17]</sup> External Robin Hashing is an extension of this algorithm where the table is stored in an external file and each table position corresponds to a fixed-sized page or bucket with  $B$  records.<sup>[18]</sup>

### 2-choice hashing

2-choice hashing employs 2 different hash functions,  $h_1(x)$  and  $h_2(x)$ , for the hash table. Both hash functions are used to compute two table locations. When an object is inserted in the table, then it is placed in the table location that contains fewer objects (with the default being the  $h_1(x)$  table location if there is equality in bucket size). 2-choice hashing employs the principle of the power of two choices.<sup>[19]</sup>

#### 11.1.4 Dynamic resizing

The good functioning of a hash table depends on the fact that the table size is proportional to the number of entries. With a fixed size, and the common structures, it is similar to linear search, except with a better constant factor. In some cases, the number of entries may be definitely known in advance, for example keywords in a language. More commonly, this is not known for sure, if only due to later changes in code and data. It is one serious, although common, mistake to not provide *any* way for the table to resize. A general-purpose hash table “class” will almost always have some way to resize, and it is good practice even for simple “custom” tables. An implementation should check the load factor, and do something if it becomes too large (this needs to be done only on inserts, since that is the only thing that would increase it).

To keep the load factor under a certain limit, e.g., under 3/4, many table implementations expand the table when items are inserted. For example, in **Java**’s `HashMap` class the default load factor threshold for table expansion is 0.75 and in **Python**’s `dict`, table size is resized when load factor is greater than 2/3.

Since buckets are usually implemented on top of a **dynamic array** and any constant proportion for resizing greater than 1 will keep the load factor under the desired limit, the exact choice of the constant is determined by the same space-time tradeoff as for **dynamic arrays**.

Resizing is accompanied by a full or incremental table **rehash** whereby existing items are mapped to new bucket

locations.

To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table—followed by a rehash—when items are deleted. From the point of **space-time tradeoffs**, this operation is similar to the deallocation in dynamic arrays.

### Resizing by copying all entries

A common approach is to automatically trigger a complete resizing when the load factor exceeds some threshold  $r_{\max}$ . Then a new larger table is allocated, all the entries of the old table are removed and inserted into this new table, and the old table is returned to the free storage pool. Symmetrically, when the load factor falls below a second threshold  $r_{\min}$ , all entries are moved to a new smaller table.

For hash tables that shrink and grow frequently, the resizing downward can be skipped entirely. In this case, the table size is proportional to the number of entries that ever were in the hash table, rather than the current number. The disadvantage is that memory usage will be higher, and thus cache behavior may be worse. For best control, a “shrink-to-fit” operation can be provided that does this only on request.

If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizings, amortized over all insert and delete operations, is still a constant, independent of the number of entries  $n$  and of the number  $m$  of operations performed.

For example, consider a table that was created with the minimum possible size and is doubled each time the load ratio exceeds some threshold. If  $m$  elements are inserted into that table, the total number of extra re- insertions that occur in all dynamic resizings of the table is at most  $m - 1$ . In other words, dynamic resizing roughly doubles the cost of each insert or delete operation.

### Incremental resizing

Some hash table implementations, notably in **real-time systems**, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to perform the resizing gradually:

- During the resize, allocate the new hash table, but keep the old table unchanged.
- In each lookup or delete operation, check both tables.
- Perform insertion operations only in the new table.
- At each insertion also move  $r$  elements from the old table to the new table.

- When all elements are removed from the old table, deallocate it.

To ensure that the old table is completely copied over before the new table itself needs to be enlarged, it is necessary to increase the size of the table by a factor of at least  $(r + 1)/r$  during resizing.

Disk-based hash tables almost always use some scheme of incremental resizing, since the cost of rebuilding the entire table on disk would be too high.

### Monotonic keys

If it is known that key values will always increase (or decrease) **monotonically**, then a variation of **consistent hashing** can be achieved by keeping a list of the single most recent key value at each hash table resize operation. Upon lookup, keys that fall in the ranges defined by these list entries are directed to the appropriate hash function—and indeed hash table—both of which can be different for each range. Since it is common to grow the overall number of entries by doubling, there will only be  $O(\lg(N))$  ranges to check, and binary search time for the redirection would be  $O(\lg(\lg(N)))$ . As with consistent hashing, this approach guarantees that any key’s hash, once issued, will never change, even when the hash table is later grown.

### Other solutions

**Linear hashing**<sup>[20]</sup> is a hash table algorithm that permits incremental hash table expansion. It is implemented using a single hash table, but with two possible look-up functions.

Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called **consistent hashing**, is prevalent in disk-based and distributed hashes, where rehashing is prohibitively costly.

### 11.1.5 Performance analysis

In the simplest model, the hash function is completely unspecified and the table does not resize. For the best possible choice of hash function, a table of size  $k$  with open addressing has no collisions and holds up to  $k$  elements, with a single comparison for successful lookup, and a table of size  $k$  with chaining and  $n$  keys has the minimum  $\max(0, n-k)$  collisions and  $O(1 + n/k)$  comparisons for lookup. For the worst choice of hash function, every insertion causes a collision, and hash tables degenerate to linear search, with  $\Omega(n)$  amortized comparisons per insertion and up to  $n$  comparisons for a successful lookup.

Adding rehashing to this model is straightforward. As in a **dynamic array**, geometric resizing by a factor of  $b$  im-

plies that only  $n/b^i$  keys are inserted  $i$  or more times, so that the total number of insertions is bounded above by  $bn/(b-1)$ , which is  $O(n)$ . By using rehashing to maintain  $n < k$ , tables using both chaining and open addressing can have unlimited elements and perform successful lookup in a single comparison for the best choice of hash function.

In more realistic models, the hash function is a **random variable** over a probability distribution of hash functions, and performance is computed on average over the choice of hash function. When this distribution is **uniform**, the assumption is called “simple uniform hashing” and it can be shown that hashing with chaining requires  $\Theta(1 + n/k)$  comparisons on average for an unsuccessful lookup, and hashing with open addressing requires  $\Theta(1/(1 - n/k))$ .<sup>[21]</sup> Both these bounds are constant, if we maintain  $n/k < c$  using table resizing, where  $c$  is a fixed constant less than 1.

### 11.1.6 Features

#### Advantages

The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large. Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

If the set of key-value pairs is fixed and known ahead of time (so insertions and deletions are not allowed), one may reduce the average lookup cost by a careful choice of the hash function, bucket table size, and internal data structures. In particular, one may be able to devise a hash function that is collision-free, or even perfect (see below). In this case the keys need not be stored in the table.

#### Drawbacks

Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree. Thus hash tables are not effective when the number of entries is very small. (However, in some cases the high cost of computing the hash function can be mitigated by saving the hash value together with the key.)

For certain string processing applications, such as **spell-checking**, hash tables may be less efficient than tries, finite automata, or **Judy arrays**. Also, if each key is represented by a small enough number of bits, then, instead of a hash table, one may use the key directly as the index into an array of values. Note that there are no collisions in this case.

The entries stored in a hash table can be enumerated efficiently (at constant cost per entry), but only in some pseudo-random order. Therefore, there is no efficient way to locate an entry whose key is *nearest* to a given key. Listing all  $n$  entries in some specific order generally requires a separate sorting step, whose cost is proportional to  $\log(n)$  per entry. In comparison, ordered search trees have lookup and insertion cost proportional to  $\log(n)$ , but allow finding the nearest key at about the same cost, and *ordered* enumeration of all entries at constant cost per entry.

If the keys are not stored (because the hash function is collision-free), there may be no easy way to enumerate the keys that are present in the table at any given moment.

Although the *average* cost per operation is constant and fairly small, the cost of a single operation may be quite high. In particular, if the hash table uses **dynamic resizing**, an insertion or deletion operation may occasionally take time proportional to the number of entries. This may be a serious drawback in real-time or interactive applications.

Hash tables in general exhibit poor **locality of reference**—that is, the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger **microprocessor cache** misses that cause long delays. Compact data structures such as arrays searched with **linear search** may be faster, if the table is relatively small and keys are compact. The optimal performance point varies from system to system.

Hash tables become quite inefficient when there are many collisions. While extremely uneven hash distributions are extremely unlikely to arise by chance, a **malicious adversary** with knowledge of the hash function may be able to supply information to a hash that creates worst-case behavior by causing excessive collisions, resulting in very poor performance, e.g., a **denial of service attack**.<sup>[22]</sup> In critical applications, **universal hashing** can be used; a data structure with better worst-case guarantees may be preferable.<sup>[23]</sup>

### 11.1.7 Uses

#### Associative arrays

Hash tables are commonly used to implement many types of in-memory tables. They are used to implement **associative arrays** (arrays whose indices are arbitrary **strings** or other complicated objects), especially in interpreted programming languages like **Perl**, **Ruby**, **Python**, and **PHP**.

When storing a new item into a **multimap** and a hash collision occurs, the multimap unconditionally stores both items.

When storing a new item into a typical associative array

and a hash collision occurs, but the actual keys themselves are different, the associative array likewise stores both items. However, if the key of the new item exactly matches the key of an old item, the associative array typically erases the old item and overwrites it with the new item, so every item in the table has a unique key.

### Database indexing

Hash tables may also be used as **disk**-based data structures and **database indices** (such as in **dbm**) although **B-trees** are more popular in these applications.

### Caches

Hash tables can be used to implement **caches**, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media. In this application, hash collisions can be handled by discarding one of the two colliding entries—usually erasing the old item that is currently stored in the table and overwriting it with the new item, so every item in the table has a unique hash value.

### Sets

Besides recovering the entry that has a given key, many hash table implementations can also tell whether such an entry exists or not.

Those structures can therefore be used to implement a **set data structure**, which merely records whether a given key belongs to a specified set of keys. In this case, the structure can be simplified by eliminating all parts that have to do with the entry values. Hashing can be used to implement both static and dynamic sets.

### Object representation

Several dynamic languages, such as **Perl**, **Python**, **JavaScript**, and **Ruby**, use hash tables to implement objects. In this representation, the keys are the names of the members and methods of the object, and the values are pointers to the corresponding member or method.

### Unique data representation

Hash tables can be used by some programs to avoid creating multiple character strings with the same contents. For that purpose, all strings in use by the program are stored in a single *string pool* implemented as a hash table, which is checked whenever a new string has to be created. This technique was introduced in **Lisp** interpreters under the name **hash consing**, and can be used with many other

kinds of data (expression trees in a symbolic algebra system, records in a database, files in a file system, binary decision diagrams, etc.)

### String interning

Main article: [String interning](#)

## 11.1.8 Implementations

### In programming languages

Many programming languages provide hash table functionality, either as built-in associative arrays or as standard **library** modules. In **C++11**, for example, the **unordered\_map** class provides hash tables for keys and values of arbitrary type.

In **PHP 5**, the Zend 2 engine uses one of the hash functions from **Daniel J. Bernstein** to generate the hash values used in managing the mappings of data pointers stored in a hash table. In the PHP source code, it is labelled as **DJBX33A** (Daniel J. Bernstein, Times 33 with Addition).

**Python**'s built-in hash table implementation, in the form of the **dict** type, as well as **Perl**'s hash type (%) are used internally to implement namespaces and therefore need to pay more attention to security, i.e., collision attacks. Python sets also use hashes internally, for fast lookup (though they store only keys, not values).<sup>[24]</sup>

In the **.NET Framework**, support for hash tables is provided via the non-generic **Hashtable** and generic **Dictionary** classes, which store key-value pairs, and the generic **HashSet** class, which stores only values.

### Independent packages

- **SparseHash** (formerly Google SparseHash) An extremely memory-efficient **hash\_map** implementation, with only 2 bits/entry of overhead. The **SparseHash** library has several C++ hash map implementations with different performance characteristics, including one that optimizes for memory use and another that optimizes for speed.
- **SunriseDD** An open source C library for hash table storage of arbitrary data objects with lock-free lookups, built-in reference counting and guaranteed order iteration. The library can participate in external reference counting systems or use its own built-in reference counting. It comes with a variety of hash functions and allows the use of runtime supplied hash functions via callback mechanism. Source code is well documented.
- **uthash** This is an easy-to-use hash table for C structures.

### 11.1.9 History

The idea of hashing arose independently in different places. In January 1953, H. P. Luhn wrote an internal IBM memorandum that used hashing with chaining.<sup>[25]</sup> G. N. Amdahl, E. M. Boehme, N. Rochester, and Arthur Samuel implemented a program using hashing at about the same time. Open addressing with linear probing (relatively prime stepping) is credited to Amdahl, but Ershov (in Russia) had the same idea.<sup>[25]</sup>

### 11.1.10 See also

- Rabin–Karp string search algorithm
- Stable hashing
- Consistent hashing
- Extendible hashing
- Lazy deletion
- Pearson hashing
- PhotoDNA

### Related data structures

There are several data structures that use hash functions but cannot be considered special cases of hash tables:

- **Bloom filter**, memory efficient data-structure designed for constant-time approximate lookups; uses hash function(s) and can be seen as an approximate hash table.
- **Distributed hash table (DHT)**, a resilient dynamic table spread over several nodes of a network.
- **Hash array mapped trie**, a trie structure, similar to the array mapped trie, but where each key is hashed first.

### 11.1.11 References

- [1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). *Introduction to Algorithms* (3rd ed.). Massachusetts Institute of Technology. pp. 253–280. ISBN 978-0-262-03384-8.
- [2] Charles E. Leiserson, *Amortized Algorithms, Table Doubling, Potential Method* Lecture 13, course MIT 6.046J/18.410J Introduction to Algorithms—Fall 2005
- [3] Knuth, Donald (1998). 'The Art of Computer Programming'. 3: *Sorting and Searching* (2nd ed.). Addison-Wesley. pp. 513–558. ISBN 0-201-89685-0.
- [4] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. 221–252. ISBN 978-0-262-53196-2.
- [5] Pearson, Karl (1900). "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling". *Philosophical Magazine, Series 5* **50** (302). pp. 157–175. doi:10.1080/14786440009463897.
- [6] Plackett, Robin (1983). "Karl Pearson and the Chi-Squared Test". *International Statistical Review (International Statistical Institute (ISI))* **51** (1). pp. 59–72. doi:10.2307/1402731.
- [7] Thomas Wang (1997), Prime Double Hash Table. Retrieved April 27, 2012
- [8] Askitis, Nikolas; Zobel, Justin (October 2005). *Cache-conscious Collision Resolution in String Hash Tables. Proceedings of the 12th International Conference, String Processing and Information Retrieval (SPIRE 2005)*. 3772/2005. pp. 91–102. doi:10.1007/11575832\_11. ISBN 978-3-540-29740-6.
- [9] Askitis, Nikolas; Sinha, Ranjan (2010). "Engineering scalable, cache and space efficient tries for strings". *The VLDB Journal* **17** (5): 633–660. doi:10.1007/s00778-010-0183-9. ISSN 1066-8888.
- [10] Askitis, Nikolas (2009). *Fast and Compact Hash Tables for Integer Keys* (PDF). *Proceedings of the 32nd Australasian Computer Science Conference (ACSC 2009)* **91**. pp. 113–122. ISBN 978-1-920682-72-9.
- [11] Erik Demaine, Jeff Lind. 6.897: Advanced Data Structures. MIT Computer Science and Artificial Intelligence Laboratory. Spring 2003. [http://courses.csail.mit.edu/6.897/spring03/scribe\\_notes/L2/lecture2.pdf](http://courses.csail.mit.edu/6.897/spring03/scribe_notes/L2/lecture2.pdf)
- [12] Tenenbaum, Aaron M.; Langsam, Yedidyah; Augenstein, Moshe J. (1990). *Data Structures Using C*. Prentice Hall. pp. 456–461, p. 472. ISBN 0-13-199746-7.
- [13] Herlihy, Maurice; Shavit, Nir; Tzafrir, Moran (2008). "Hopscotch Hashing". *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*. Berlin, Heidelberg: Springer-Verlag. pp. 350–364.
- [14] Celis, Pedro (1986). *Robin Hood hashing* (PDF) (Technical report). Computer Science Department, University of Waterloo. CS-86-14.
- [15] Goossaert, Emmanuel (2013). "Robin Hood hashing".
- [16] Amble, Ole; Knuth, Don (1974). "Ordered hash tables". *Computer Journal* **17** (2): 135. doi:10.1093/comjnl/17.2.135.
- [17] Viola, Alfredo (October 2005). "Exact distribution of individual displacements in linear probing hashing". *Transactions on Algorithms (TALG) (ACM)* **1** (2): 214–242. doi:10.1145/1103963.1103965.

- [18] Celis, Pedro (March 1988). *External Robin Hood Hashing* (Technical report). Computer Science Department, Indiana University. TR246.
- [19] <http://www.eecs.harvard.edu/~{}michaelm/postscripts/handbook2001.pdf>
- [20] Litwin, Witold (1980). "Linear hashing: A new tool for file and table addressing". *Proc. 6th Conference on Very Large Databases*. pp. 212–223.
- [21] Doug Dunham. *CS 4521 Lecture Notes*. University of Minnesota Duluth. Theorems 11.2, 11.6. Last modified April 21, 2009.
- [22] Alexander Klink and Julian Wälde's *Efficient Denial of Service Attacks on Web Application Platforms*, December 28, 2011, 28th Chaos Communication Congress. Berlin, Germany.
- [23] Crosby and Wallach's *Denial of Service via Algorithmic Complexity Attacks*.
- [24] <http://stackoverflow.com/questions/513882/python-list-vs-dict-for-look-up-table>
- [25] Mehta, Dinesh P.; Sahni, Sartaj. *Handbook of Datastructures and Applications*. pp. 9–15. ISBN 1-58488-435-5.

### 11.1.12 Further reading

- Tamassia, Roberto; Goodrich, Michael T. (2006). "Chapter Nine: Maps and Dictionaries". *Data structures and algorithms in Java : [updated for Java 5.0]* (4th ed.). Hoboken, NJ: Wiley. pp. 369–418. ISBN 0-471-73884-0.
- McKenzie, B. J.; Harries, R.; Bell, T. (Feb 1990). "Selecting a hashing algorithm". *Software Practice & Experience* **20** (2): 209–224. doi:10.1002/spe.4380200207.

### 11.1.13 External links

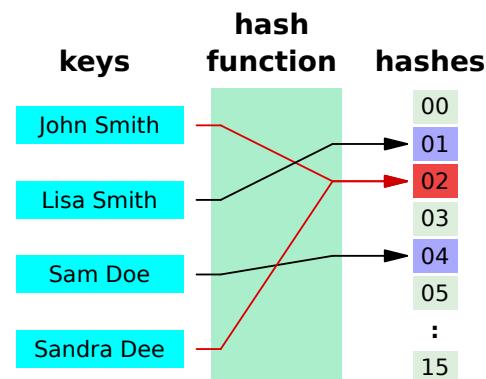
- A Hash Function for Hash Table Lookup by Bob Jenkins.
- Hash Tables by SparkNotes—explanation using C
- Hash functions by Paul Hsieh
- Design of Compact and Efficient Hash Tables for Java
- Libhashish hash library
- NIST entry on hash tables
- Open addressing hash table removal algorithm from ICI programming language, *ici\_set\_unassign* in *set.c* (and other occurrences, with permission).
- A basic explanation of how the hash table works by Reliable Software

- Lecture on Hash Tables
- Hash-tables in C—two simple and clear examples of hash tables implementation in C with linear probing and chaining
- Open Data Structures – Chapter 5 – Hash Tables
- MIT's Introduction to Algorithms: Hashing 1 MIT OCW lecture Video
- MIT's Introduction to Algorithms: Hashing 2 MIT OCW lecture Video
- How to sort a HashMap (Java) and keep the duplicate entries
- How python dictionary works

## 11.2 Hash function

This article is about a programming concept. For other meanings of "hash" and "hashing", see Hash (disambiguation).

A **hash function** is any function that can be used to map



A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

digital **data** of arbitrary size to digital data of fixed size. The values returned by a hash function are called **hash values**, **hash codes**, **hash sums**, or simply **hashes**. One use is a data structure called a **hash table**, widely used in computer software for rapid data lookup. Hash functions accelerate table or database lookup by detecting duplicated records in a large file. An example is finding similar stretches in DNA sequences. They are also useful in **cryptography**. A **cryptographic hash function** allows one to easily verify that some input data matches a stored hash value, but makes it hard to construct any data that would hash to the same value or find any two unique data pieces that hash to the same value. This principle is used by the **PGP** algorithm for data validation and by many password checking systems.

Hash functions are related to (and often confused with) checksums, check digits, fingerprints, randomization functions, error-correcting codes, and ciphers. Although these concepts overlap to some extent, each has its own uses and requirements and is designed and optimized differently. The Hash Keeper database maintained by the American National Drug Intelligence Center, for instance, is more aptly described as a catalogue of file fingerprints than of hash values.

### 11.2.1 Uses

#### Hash tables

Hash functions are primarily used in [hash tables](#), to quickly locate a data record (e.g., a dictionary definition) given its [search key](#) (the headword). Specifically, the hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored. Hash tables, in turn, are used to implement [associative arrays](#) and [dynamic sets](#).

Typically, the domain of a hash function (the set of possible keys) is larger than its range (the number of different table indexes), and so it will map several different keys to the same index. Therefore, each slot of a hash table is associated with (implicitly or explicitly) a [set](#) of records, rather than a single record. For this reason, each slot of a hash table is often called a *bucket*, and hash values are also called *bucket indices*.

Thus, the hash function only hints at the record's location — it tells where one should start looking for it. Still, in a half-full table, a good hash function will typically narrow the search down to only one or two entries.

#### Caches

Hash functions are also used to build [caches](#) for large data sets stored in slow media. A cache is generally simpler than a hashed search table, since any collision can be resolved by discarding or writing back the older of the two colliding items. This is also used in file comparison.

#### Bloom filters

Main article: [Bloom filter](#)

Hash functions are an essential ingredient of the [Bloom filter](#), a space-efficient probabilistic [data structure](#) that is used to test whether an [element](#) is a member of a [set](#).

#### Finding duplicate records

Main article: [Hash table](#)

When storing records in a large unsorted file, one may use a hash function to map each record to an index into a table  $T$ , and to collect in each bucket  $T[i]$  a [list](#) of the numbers of all records with the same hash value  $i$ . Once the table is complete, any two duplicate records will end up in the same bucket. The duplicates can then be found by scanning every bucket  $T[i]$  which contains two or more members, fetching those records, and comparing them. With a table of appropriate size, this method is likely to be much faster than any alternative approach (such as sorting the file and comparing all consecutive pairs).

#### Protecting data

Main article: [Security of cryptographic hash functions](#)

A hash value can be used to uniquely identify secret information. This requires that the hash function is [collision resistant](#), which means that it is very hard to find data that generate the same hash value. These functions are categorized into [cryptographic hash functions](#) and [provably secure hash functions](#). Functions in the second category are the most secure but also too slow for most practical purposes. Collision resistance is accomplished in part by generating very large hash values. For example [SHA-1](#), one of the most widely used cryptographic hash functions, generates 160 bit values.

#### Finding similar records

Main article: [Locality sensitive hashing](#)

Hash functions can also be used to locate table records whose key is similar, but not identical, to a given key; or pairs of records in a large file which have similar keys. For that purpose, one needs a hash function that maps similar keys to hash values that differ by at most  $m$ , where  $m$  is a small integer (say, 1 or 2). If one builds a table  $T$  of all record numbers, using such a hash function, then similar records will end up in the same bucket, or in nearby buckets. Then one need only check the records in each bucket  $T[i]$  against those in buckets  $T[i+k]$  where  $k$  ranges between  $-m$  and  $m$ .

This class includes the so-called [acoustic fingerprint](#) algorithms, that are used to locate similar-sounding entries in large collection of [audio files](#). For this application, the hash function must be as insensitive as possible to data capture or transmission errors, and to trivial changes such as timing and volume changes, compression, etc.<sup>[1]</sup>

### Finding similar substrings

The same techniques can be used to find equal or similar stretches in a large collection of strings, such as a document repository or a [genomic database](#). In this case, the input strings are broken into many small pieces, and a hash function is used to detect potentially equal pieces, as above.

The [Rabin–Karp algorithm](#) is a relatively fast string searching algorithm that works in  $O(n)$  time on average. It is based on the use of hashing to compare strings.

### Geometric hashing

This principle is widely used in [computer graphics](#), [computational geometry](#) and many other disciplines, to solve many [proximity problems](#) in the plane or in [three-dimensional space](#), such as finding [closest pairs](#) in a set of points, similar shapes in a list of shapes, similar images in an [image database](#), and so on. In these applications, the set of all inputs is some sort of [metric space](#), and the hashing function can be interpreted as a [partition](#) of that space into a grid of [cells](#). The table is often an array with two or more indices (called a *grid file*, *grid index*, *bucket grid*, and similar names), and the hash function returns an [index tuple](#). This special case of hashing is known as [geometric hashing](#) or *the grid method*. Geometric hashing is also used in [telecommunications](#) (usually under the name [vector quantization](#)) to encode and [compress](#) multi-dimensional signals.

### Standard uses of hashing in cryptography

Main article: [Cryptographic hash function](#)

Some standard applications that employ hash functions include authentication, message integrity (using an [HMAC](#) (Hashed MAC)), message fingerprinting, data corruption detection, and digital signature efficiency.

## 11.2.2 Properties

Good hash functions, in the original sense of the term, are usually required to satisfy certain properties listed below. The exact requirements are dependent on the application, for example a hash function well suited to indexing data will probably be a poor choice for a [cryptographic hash function](#).

### Determinism

A hash procedure must be [deterministic](#)—meaning that for a given input value it must always generate the same hash value. In other words, it must be a [function](#) of the data to be hashed, in the mathematical sense of the term.

This requirement excludes hash functions that depend on external variable parameters, such as [pseudo-random number generators](#) or the time of day. It also excludes functions that depend on the memory address of the object being hashed, because that address may change during execution (as may happen on systems that use certain methods of [garbage collection](#)), although sometimes rehashing of the item is possible.

### Uniformity

A good hash function should map the expected inputs as evenly as possible over its output range. That is, every hash value in the output range should be generated with roughly the same [probability](#). The reason for this last requirement is that the cost of hashing-based methods goes up sharply as the number of [collisions](#)—pairs of inputs that are mapped to the same hash value—increases. If some hash values are more likely to occur than others, a larger fraction of the lookup operations will have to search through a larger set of colliding table entries.

Note that this criterion only requires the value to be *uniformly distributed*, not *random* in any sense. A good randomizing function is (barring computational efficiency concerns) generally a good choice as a hash function, but the converse need not be true.

Hash tables often contain only a small subset of the valid inputs. For instance, a club membership list may contain only a hundred or so member names, out of the very large set of all possible names. In these cases, the uniformity criterion should hold for almost all typical subsets of entries that may be found in the table, not just for the global set of all possible entries.

In other words, if a typical set of  $m$  records is hashed to  $n$  table slots, the probability of a bucket receiving many more than  $m/n$  records should be vanishingly small. In particular, if  $m$  is less than  $n$ , very few buckets should have more than one or two records. (In an ideal "perfect hash function", no bucket should have more than one record; but a small number of collisions is virtually inevitable, even if  $n$  is much larger than  $m$  – see the [birthday paradox](#)).

When testing a hash function, the uniformity of the distribution of hash values can be evaluated by the [chi-squared test](#).

### Defined range

It is often desirable that the output of a hash function have fixed size (but see below). If, for example, the output is constrained to 32-bit integer values, the hash values can be used to index into an array. Such hashing is commonly used to accelerate data searches.<sup>[2]</sup> On the other hand, cryptographic hash functions produce much larger hash values, in order to ensure the computational complexity

of brute-force inversion.<sup>[3]</sup> For example **SHA-1**, one of the most widely used cryptographic hash functions, produces a 160-bit value.

Producing fixed-length output from variable length input can be accomplished by breaking the input data into chunks of specific size. Hash functions used for data searches use some arithmetic expression which iteratively processes chunks of the input (such as the characters in a string) to produce the hash value.<sup>[2]</sup> In cryptographic hash functions, these chunks are processed by a **one-way compression function**, with the last chunk being padded if necessary. In this case, their size, which is called **block size**, is much bigger than the size of the hash value.<sup>[3]</sup> For example, in **SHA-1**, the hash value is 160 bits and the block size 512 bits.

**Variable range** In many applications, the range of hash values may be different for each run of the program, or may change along the same run (for instance, when a hash table needs to be expanded). In those situations, one needs a hash function which takes two parameters—the input data  $z$ , and the number  $n$  of allowed hash values.

A common solution is to compute a fixed hash function with a very large range (say, 0 to  $2^{32} - 1$ ), divide the result by  $n$ , and use the division's **remainder**. If  $n$  is itself a power of 2, this can be done by **bit masking** and **bit shifting**. When this approach is used, the hash function must be chosen so that the result has fairly uniform distribution between 0 and  $n - 1$ , for any value of  $n$  that may occur in the application. Depending on the function, the remainder may be uniform only for certain values of  $n$ , e.g. **odd** or **prime numbers**.

We can allow the table size  $n$  to not be a power of 2 and still not have to perform any remainder or division operation, as these computations are sometimes costly. For example, let  $n$  be significantly less than  $2^b$ . Consider a **pseudorandom number generator** (PRNG) function  $P(\text{key})$  that is uniform on the interval  $[0, 2^b - 1]$ . A hash function uniform on the interval  $[0, n-1]$  is  $n P(\text{key})/2^b$ . We can replace the division by a (possibly faster) right bit shift:  $n P(\text{key}) \gg b$ .

**Variable range with minimal movement (dynamic hash function)** When the hash function is used to store values in a hash table that outlives the run of the program, and the hash table needs to be expanded or shrunk, the hash table is referred to as a **dynamic hash table**.

A hash function that will relocate the minimum number of records when the table is – where  $z$  is the key being hashed and  $n$  is the number of allowed hash values – such that  $H(z, n + 1) = H(z, n)$  with probability close to  $n/(n + 1)$ .

**Linear hashing** and **spiral storage** are examples of dynamic hash functions that execute in constant time but relax the property of uniformity to achieve the minimal

movement property.

**Extendible hashing** uses a dynamic hash function that requires space proportional to  $n$  to compute the hash function, and it becomes a function of the previous keys that have been inserted.

Several algorithms that preserve the uniformity property but require time proportional to  $n$  to compute the value of  $H(z, n)$  have been invented.

### Data normalization

In some applications, the input data may contain features that are irrelevant for comparison purposes. For example, when looking up a personal name, it may be desirable to ignore the distinction between upper and lower case letters. For such data, one must use a hash function that is compatible with the data **equivalence** criterion being used: that is, any two inputs that are considered equivalent must yield the same hash value. This can be accomplished by normalizing the input before hashing it, as by upper-casing all letters.

### Continuity

“A hash function that is used to search for similar (as opposed to equivalent) data must be as **continuous** as possible; two inputs that differ by a little should be mapped to equal or nearly equal hash values.”<sup>[4]</sup>

Note that continuity is usually considered a fatal flaw for checksums, **cryptographic hash functions**, and other related concepts. Continuity is desirable for hash functions only in some applications, such as hash tables used in **Nearest neighbor search**.

### Non-invertible

In cryptographic applications, hash functions are typically expected to be **non-invertible**, meaning that it is not possible to reconstruct the input datum  $x$  from its hash value  $h(x)$  alone without spending great amounts of computing time (see also **One-way function**).

## 11.2.3 Hash function algorithms

For most types of hashing functions the choice of the function depends strongly on the nature of the input data, and their probability distribution in the intended application.

### Trivial hash function

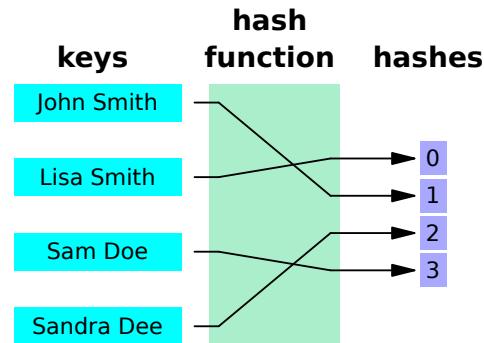
If the datum to be hashed is small enough, one can use the datum itself (reinterpreted as an integer) as the hashed value. The cost of computing this “trivial” (identity) hash

function is effectively zero. This hash function is **perfect**, as it maps each input to a distinct hash value.

The meaning of “small enough” depends on the size of the type that is used as the hashed value. For example, in **Java**, the hash code is a 32-bit integer. Thus the 32-bit integer **Integer** and 32-bit floating-point **Float** objects can simply use the value directly; whereas the 64-bit integer **Long** and 64-bit floating-point **Double** cannot use this method.

Other types of data can also use this perfect hashing scheme. For example, when mapping character strings between **upper** and **lower case**, one can use the binary encoding of each character, interpreted as an integer, to index a table that gives the alternative form of that character (“A” for “a”, “8” for “8”, etc.). If each character is stored in 8 bits (as in **ASCII** or **ISO Latin 1**), the table has only  $2^8 = 256$  entries; in the case of **Unicode** characters, the table would have  $17 \times 2^{16} = 1114112$  entries.

The same technique can be used to map **two-letter country codes** like “us” or “za” to country names ( $26^2 = 676$  table entries), 5-digit zip codes like 13083 to city names (100000 entries), etc. Invalid data values (such as the country code “xx” or the zip code 00000) may be left undefined in the table, or mapped to some appropriate “null” value.



A minimal perfect hash function for the four names shown

### Minimal perfect hashing

A perfect hash function for  $n$  keys is said to be **minimal** if its range consists of  $n$  consecutive integers, usually from 0 to  $n-1$ . Besides providing single-step lookup, a minimal perfect hash function also yields a compact hash table, without any vacant slots. Minimal perfect hash functions are much harder to find than perfect ones with a wider range.

### Hashing uniformly distributed data

If the inputs are bounded-length strings and each input may independently occur with uniform probability (such as **telephone numbers**, **car license plates**, **invoice numbers**, etc.), then a hash function needs to map roughly the same number of inputs to each hash value. For instance, suppose that each input is an integer  $z$  in the range 0 to  $N-1$ , and the output must be an integer  $h$  in the range 0 to  $n-1$ , where  $N$  is much larger than  $n$ . Then the hash function could be  $h = z \bmod n$  (the remainder of  $z$  divided by  $n$ ), or  $h = (z \times n) \div N$  (the value  $z$  scaled down by  $n/N$  and truncated to an integer), or many other formulas.

$h = z \bmod n$  was used in many of the original random number generators, but was found to have a number of issues. One of which is that as  $n$  approaches  $N$ , this function becomes less and less uniform.

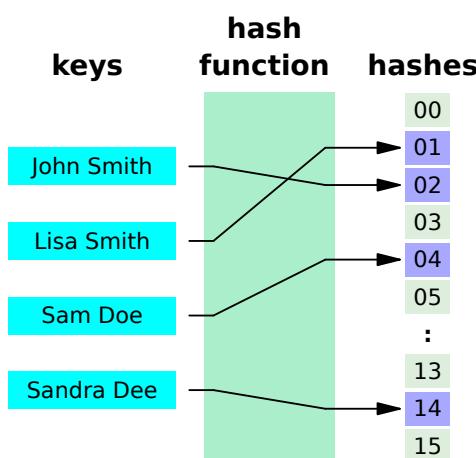
### Hashing data with other distributions

These simple formulas will not do if the input values are not equally likely, or are not independent. For instance, most patrons of a **supermarket** will live in the same geographic area, so their telephone numbers are likely to begin with the same 3 to 4 digits. In that case, if  $m$  is 10000 or so, the division formula  $(z \times m) \div M$ , which depends mainly on the leading digits, will generate a lot of collisions; whereas the remainder formula  $z \bmod m$ , which is quite sensitive to the trailing digits, may still yield a fairly even distribution.

### Perfect hashing

Main article: **Perfect hash function**

A hash function that is **injective**—that is, maps each valid



A perfect hash function for the four names shown

input to a different hash value—is said to be **perfect**. With such a function one can directly locate the desired entry in a hash table, without any additional searching.

## Hashing variable-length data

When the data values are long (or variable-length) character strings—such as personal names, web page addresses, or mail messages—their distribution is usually very uneven, with complicated dependencies. For example, text in any natural language has highly non-uniform distributions of characters, and character pairs, very characteristic of the language. For such data, it is prudent to use a hash function that depends on all characters of the string—and depends on each character in a different way.

In cryptographic hash functions, a Merkle–Damgård construction is usually used. In general, the scheme for hashing such data is to break the input into a sequence of small units (bits, bytes, words, etc.) and combine all the units  $b[1], b[2], \dots, b[m]$  sequentially, as follows

```
S ← S0; // Initialize the state. for k in 1, 2, ..., m do // Scan the input data units: S ← F(S, b[k]); // Combine data unit k into the state. return G(S, n) // Extract the hash value from the state.
```

This schema is also used in many text checksum and fingerprint algorithms. The state variable  $S$  may be a 32- or 64-bit unsigned integer; in that case,  $S0$  can be 0, and  $G(S, n)$  can be just  $S \bmod n$ . The best choice of  $F$  is a complex issue and depends on the nature of the data. If the units  $b[k]$  are single bits, then  $F(S, b)$  could be, for instance

```
if highbit(S) = 0 then return 2 * S + b else return (2 * S + b) ^ P
```

Here  $\text{highbit}(S)$  denotes the most significant bit of  $S$ ; the '\*' operator denotes unsigned integer multiplication with lost overflow; '^' is the bitwise exclusive or operation applied to words; and  $P$  is a suitable fixed word.<sup>[5]</sup>

## Special-purpose hash functions

In many cases, one can design a special-purpose (heuristic) hash function that yields many fewer collisions than a good general-purpose hash function. For example, suppose that the input data are file names such as FILE0000.CHK, FILE0001.CHK, FILE0002.CHK, etc., with mostly sequential numbers. For such data, a function that extracts the numeric part  $k$  of the file name and returns  $k \bmod n$  would be nearly optimal. Needless to say, a function that is exceptionally good for a specific kind of data may have dismal performance on data with different distribution.

## Rolling hash

Main article: rolling hash

In some applications, such as substring search, one must compute a hash function  $h$  for every  $k$ -character substring

of a given  $n$ -character string  $t$ ; where  $k$  is a fixed integer, and  $n$  is  $k$ . The straightforward solution, which is to extract every such substring  $s$  of  $t$  and compute  $h(s)$  separately, requires a number of operations proportional to  $k \cdot n$ . However, with the proper choice of  $h$ , one can use the technique of rolling hash to compute all those hashes with an effort proportional to  $k + n$ .

## Universal hashing

A universal hashing scheme is a randomized algorithm that selects a hashing function  $h$  among a family of such functions, in such a way that the probability of a collision of any two distinct keys is  $1/n$ , where  $n$  is the number of distinct hash values desired—Independently of the two keys. Universal hashing ensures (in a probabilistic sense) that the hash function application will behave as well as if it were using a random function, for any distribution of the input data. It will however have more collisions than perfect hashing, and may require more operations than a special-purpose hash function. See also Unique Permutation Hashing.<sup>[6]</sup>

## Hashing with checksum functions

One can adapt certain checksum or fingerprinting algorithms for use as hash functions. Some of those algorithms will map arbitrary long string data  $z$ , with any typical real-world distribution—no matter how non-uniform and dependent—to a 32-bit or 64-bit string, from which one can extract a hash value in 0 through  $n - 1$ .

This method may produce a sufficiently uniform distribution of hash values, as long as the hash range size  $n$  is small compared to the range of the checksum or fingerprint function. However, some checksums fare poorly in the avalanche test, which may be a concern in some applications. In particular, the popular CRC32 checksum provides only 16 bits (the higher half of the result) that are usable for hashing. Moreover, each bit of the input has a deterministic effect on each bit of the CRC32, that is one can tell without looking at the rest of the input, which bits of the output will flip if the input bit is flipped; so care must be taken to use all 32 bits when computing the hash from the checksum.<sup>[7]</sup>

## Hashing with cryptographic hash functions

Some cryptographic hash functions, such as SHA-1, have even stronger uniformity guarantees than checksums or fingerprints, and thus can provide very good general-purpose hashing functions.

In ordinary applications, this advantage may be too small to offset their much higher cost.<sup>[8]</sup> However, this method can provide uniformly distributed hashes even when the keys are chosen by a malicious agent. This feature may help to protect services against denial of service attacks.

### Hashing By Nonlinear Table Lookup

Tables of random numbers (such as 256 random 32 bit integers) can provide high-quality nonlinear functions to be used as hash functions or for other purposes such as cryptography. The key to be hashed would be split into 8-bit (one byte) parts and each part will be used as an index for the nonlinear table. The table values will be added by arithmetic or XOR addition to the hash output value. Because the table is just 1024 bytes in size, it will fit into the cache of modern microprocessors and allow for very fast execution of the hashing algorithm. As the table value is on average much longer than 8 bits, one bit of input will affect nearly all output bits. This is different from multiplicative hash functions where higher-value input bits do not affect lower-value output bits.

This algorithm has proven to be very fast and of high quality for hashing purposes (especially hashing of integer number keys).

### Efficient Hashing Of Strings

- See also Universal hashing of strings

Modern microprocessors will allow for much faster processing, if 8-bit character strings are not hashed by processing one character at a time, but by interpreting the string as an array of 32 bit or 64 bit integers and hashing/accumulating these “wide word” integer values by means of arithmetic operations (e.g. multiplication by constant and bit-shifting). The remaining characters of the string which are smaller than the word length of the CPU must be handled differently (e.g. being processed one character at a time).

This approach has proven to speed up hash code generation by a factor of five or more on modern microprocessors of a word size of 64 bit.

Another approach<sup>[9]</sup> is to convert strings to a 32 or 64 bit numeric value and then apply a hash function. One method that avoids the problem of strings having great similarity (“Aaaaaaaaaa” and “Aaaaaaaaaab”) is to use a **Cyclic redundancy check** (CRC) of the string to compute a 32- or 64-bit value. While it is possible that two different strings will have the same CRC, the likelihood is very small and only requires that one check the actual string found to determine whether one has an exact match. CRCs will be different for strings such as “Aaaaaaaaaa” and “Aaaaaaaaaab”. Although, CRC codes can be used as hash values<sup>[10]</sup> they are not cryptographically secure since they are not **collision resistant**.<sup>[11]</sup>

### 11.2.4 Locality-sensitive hashing

Locality-sensitive hashing (LSH) is a method of performing probabilistic dimension reduction of high-dimensional data. The basic idea is to hash the input

items so that similar items are mapped to the same buckets with high probability (the number of buckets being much smaller than the universe of possible input items). This is different from the conventional hash functions, such as those used in cryptography, as in this case the goal is to maximize the probability of “collision” of similar items rather than to avoid collisions.<sup>[12]</sup>

One example of LSH is **MinHash** algorithm used for finding similar documents (such as web-pages):

Let  $h$  be a hash function that maps the members of  $A$  and  $B$  to distinct integers, and for any set  $S$  define  $h_{\min}(S)$  to be the member  $x$  of  $S$  with the minimum value of  $h(x)$ . Then  $h_{\min}(A) = h_{\min}(B)$  exactly when the minimum hash value of the union  $A \cup B$  lies in the intersection  $A \cap B$ . Therefore,

$$\Pr[h_{\min}(A) = h_{\min}(B)] = J(A, B). \text{ where } J \text{ is Jaccard index.}$$

In other words, if  $r$  is a random variable that is one when  $h_{\min}(A) = h_{\min}(B)$  and zero otherwise, then  $r$  is an **unbiased estimator** of  $J(A, B)$ , although it has too high a **variance** to be useful on its own. The idea of the MinHash scheme is to reduce the variance by averaging together several variables constructed in the same way.

### 11.2.5 Origins of the term

The term “hash” comes by way of analogy with its non-technical meaning, to “chop and mix”. Indeed, typical hash functions, like the **mod** operation, “chop” the input domain into many sub-domains that get “mixed” into the output range to improve the uniformity of the key distribution.

Donald Knuth notes that Hans Peter Luhn of IBM appears to have been the first to use the concept, in a memo dated January 1953, and that Robert Morris used the term in a survey paper in **CACM** which elevated the term from technical jargon to formal terminology.<sup>[13]</sup>

### 11.2.6 List of hash functions

Main article: [List of hash functions](#)

- NIST hash function competition
- Bernstein hash<sup>[14]</sup>
- Fowler-Noll-Vo hash function (32, 64, 128, 256, 512, or 1024 bits)
- Jenkins hash function (32 bits)
- Pearson hashing (64 bits)
- Zobrist hashing
- Almost linear hash function

### 11.2.7 See also

Computer science portal

- Bloom filter
- Coalesced hashing
- Cuckoo hashing
- Hopscotch hashing
- Cryptographic hash function
- Distributed hash table
- Geometric hashing
- Hash Code cracker
- Hash table
- HMAC
- Identicon
- Linear hash
- List of hash functions
- Locality sensitive hashing
- MD5
- Perfect hash function
- PhotoDNA
- Rabin–Karp string search algorithm
- Rolling hash
- Transposition table
- Universal hashing
- MinHash
- Low-discrepancy sequence

### 11.2.8 References

- [1] “Robust Audio Hashing for Content Identification by Jaap Haitsma, Ton Kalker and Job Oostveen”
- [2] Sedgewick, Robert (2002). “14. Hashing”. *Algorithms in Java* (3 ed.). Addison Wesley. ISBN 978-0201361209.
- [3] Menezes, Alfred J.; van Oorschot, Paul C.; Vanstone, Scott A (1996). *Handbook of Applied Cryptography*. CRC Press. ISBN 0849385237.
- [4] “Fundamental Data Structures – Josiang p.132”. Retrieved May 19, 2014.

- [5] Broder, A. Z. (1993). “Some applications of Rabin’s fingerprinting method”. *Sequences II: Methods in Communications, Security, and Computer Science*. Springer-Verlag. pp. 143–152.
- [6] Shlomi Dolev, Limor Lahiani, Yinnon Haviv, “Unique permutation hashing,” *Theoretical Computer Science* Volume 475, 4 March 2013, Pages 59–65
- [7] Bret Mulvey, *Evaluation of CRC32 for Hash Tables*, in *Hash Functions*. Accessed April 10, 2009.
- [8] Bret Mulvey, *Evaluation of SHA-1 for Hash Tables*, in *Hash Functions*. Accessed April 10, 2009.
- [9] <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.18.7520> Performance in Practice of String Hashing Functions
- [10] Peter Kankowski. “Hash functions: An empirical comparison”.
- [11] Cam-Winget, Nancy; Housley, Russ; Wagner, David; Walker, Jesse (May 2003). “Security Flaws in 802.11 Data Link Protocols”. *Communications of the ACM* **46** (5): 35–39. doi:10.1145/769800.769823.
- [12] A. Rajaraman and J. Ullman (2010). “Mining of Massive Datasets, Ch. 3.”.
- [13] Knuth, Donald (1973). *The Art of Computer Programming, volume 3, Sorting and Searching*. pp. 506–542.
- [14] “Hash Functions”. *cse.yorku.ca*. September 22, 2003. Retrieved November 1, 2012. the djb2 algorithm (k=33) was first reported by dan bernstein many years ago in comp.lang.c.

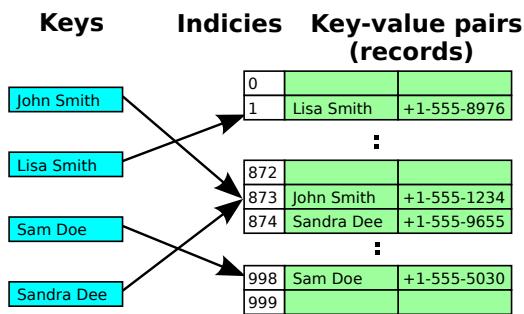
### 11.2.9 External links

- Hash Functions and Block Ciphers by Bob Jenkins
- The Goulburn Hashing Function (PDF) by Mayur Patel
- Hash Function Construction for Textual and Geometrical Data Retrieval Latest Trends on Computers, Vol.2, pp. 483–489, CSCC conference, Corfu, 2010

## 11.3 Open addressing

**Open addressing**, or **closed hashing**, is a method of collision resolution in hash tables. With this method a hash collision is resolved by **probing**, or searching through alternate locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.<sup>[1]</sup> Well known probe sequences include:

**Linear probing** in which the interval between probes is fixed — often at 1.



Hash collision resolved by linear probing (interval=1).

**Quadratic probing** in which the interval between probes increases linearly (hence, the indices are described by a quadratic function).

**Double hashing** in which the interval between probes is fixed for each record but is computed by another hash function.

The main tradeoffs between these methods are that linear probing has the best cache performance but is most sensitive to clustering, while double hashing has poor cache performance but exhibits virtually no clustering; quadratic probing falls in-between in both areas. Double hashing can also require more computation than other forms of probing. Some open addressing methods, such as *last-come-first-served hashing* and *cuckoo hashing* move existing keys around in the array to make room for the new key. This gives better maximum search times than the methods based on probing.

A critical influence on performance of an open addressing hash table is the *load factor*; that is, the proportion of the slots in the array that are used. As the load factor increases towards 100%, the number of probes that may be required to find or insert a given key rises dramatically. Once the table becomes full, probing algorithms may even fail to terminate. Even with good hash functions, load factors are normally limited to 80%. A poor hash function can exhibit poor performance even at very low load factors by generating significant clustering. What causes hash functions to cluster is not well understood, and it is easy to unintentionally write a hash function which causes severe clustering.

### 11.3.1 Example pseudo code

The following pseudocode is an implementation of an open addressing hash table with linear probing and single-slot stepping, a common approach that is effective if the hash function is good. Each of the **lookup**, **set** and **remove** functions use a common internal function **find\_slot** to locate the array slot that either does or should contain a given key.

```

record pair { key, value } var pair array
slot[0..num_slots-1] function find_slot(key) i := hash(key) modulo num_slots // search until we either find the key, or find an empty slot. while (slot[i] is occupied) and (slot[i].key ≠ key) i = (i + 1) modulo num_slots
return i function lookup(key) i := find_slot(key) if slot[i] is occupied // key is in table return slot[i].value else // key is not in table return not found function set(key, value) i := find_slot(key) if slot[i] is occupied // we found our key slot[i].value = value return if the table is almost full rebuild the table larger (note 1) i = find_slot(key) slot[i].key = key slot[i].value = value

```

**note 1** Rebuilding the table requires allocating a larger array and recursively using the **set** operation to insert all the elements of the old array into the new larger array. It is common to increase the array size exponentially, for example by doubling the old array size.

```

function remove(key) i := find_slot(key) if slot[i] is unoccupied return // key is not in the table j := i loop mark slot[i] as unoccupied r2: (note 2) j := (j+1) modulo num_slots if slot[j] is unoccupied exit loop k := hash(slot[j].key) modulo num_slots // determine if k lies cyclically in [i,j] // | i.k.j | // | ... j.i.k. | or | k..j.i... | if ((i<=j) ? ((i<k)&&(k<=j)) : ((i<k)||((k<=j))) goto r2; slot[i] := slot[j] i := j

```

**note 2** For all records in a cluster, there must be no vacant slots between their natural hash position and their current position (else lookups will terminate before finding the record). At this point in the pseudocode, *i* is a vacant slot that might be invalidating this property for subsequent records in the cluster. *j* is such a subsequent record. *k* is the raw hash where the record at *j* would naturally land in the hash table if there were no collisions. This test is asking if the record at *j* is invalidly positioned with respect to the required properties of a cluster now that *i* is vacant.

Another technique for removal is simply to mark the slot as deleted. However this eventually requires rebuilding the table simply to remove deleted records. The methods above provide O(1) updating and removal of existing records, with occasional rebuilding if the high-water mark of the table size grows.

The O(1) remove method above is only possible in linearly probed hash tables with single-slot stepping. In the case where many records are to be deleted in one operation, marking the slots for deletion and later rebuilding may be more efficient.

### 11.3.2 See also

- Lazy deletion – a method of deleting from a hash table using open addressing.

### 11.3.3 References

- [1] Tenenbaum, Aaron M.; Langsam, Yedidyah; Augenstein, Moshe J. (1990), *Data Structures Using C*, Prentice Hall, pp. 456–461, pp. 472, ISBN 0-13-199746-7

## 11.4 Lazy deletion

In computer science, **lazy deletion** refers to a method of deleting elements from a **hash table** that uses **open addressing**. In this method, deletions are done by marking an element as deleted, rather than erasing it entirely. Deleted locations are treated as empty when inserting and as occupied during a search.

The problem with this scheme is that as the number of delete/insert operations increases, the cost of a successful search increases. To improve this, when an element is searched and found in the table, the element is relocated to the first location marked for deletion that was probed during the search. Instead of finding an element to relocate when the deletion occurs, the relocation occurs lazily during the next search.<sup>[1][2]</sup>

### 11.4.1 References

- [1] Celis, Pedro; Franco, John (1995), *The Analysis of Hashing with Lazy Deletions*, Computer Science Department, Indiana University, Technical Report CS-86-14
- [2] Celis, Pedro; Franco, John (1992), “The analysis of hashing with lazy deletions”, *Information Sciences* **62**: 13, doi:10.1016/0020-0255(92)90022-Z

## 11.5 Linear probing

**Linear probing** is a scheme in computer programming for resolving **hash collisions** of values of **hash functions** by sequentially searching the hash table for a free location.<sup>[1]</sup>

### 11.5.1 Algorithm

Linear probing is accomplished using two values - one as a starting value and one as an interval between successive values in **modular arithmetic**. The second value, which is the same for all keys and known as the **stepsize**, is repeatedly added to the starting value until a free space is found, or the entire table is traversed. (In order to traverse the entire table the stepsize should be **relatively prime** to the arraysize, which is why the array size is often chosen to be a prime number.)

```
newLocation = (startingValue +
stepSize) % arraySize
```

Given an ordinary hash function  $H(x)$ , a linear probing function ( $H(x, i)$ ) would be:

$$H(x, i) = (H(x) + i) \pmod{n}.$$

Here  $H(x)$  is the starting value,  $n$  the size of the hash table, and the **stepsize** is  $i$  in this case.

Often, the step size is one; that is, the array cells that are probed are consecutive in the hash table. Double hashing is a variant of the same method in which the step size is itself computed by a hash function.

### 11.5.2 Properties

This algorithm, which is used in open-addressed **hash tables**, provides good memory caching (if stepsize is equal to one), through good locality of reference, but also results in clustering, an unfortunately high **probability** that where there has been one collision there will be more. The performance of linear probing is also more sensitive to input distribution when compared to **double hashing**, where the stepsize is determined by another hash function applied to the value instead of a fixed stepsize as in linear probing.

### 11.5.3 Dictionary operation in constant time

Using linear probing, dictionary operation can be implemented in constant time. In other words, insert, remove and search operations can be implemented in  $O(1)$ , as long as the **load factor** of the hash table is a constant strictly less than one.<sup>[2]</sup> This analysis makes the (unrealistic) assumption that the hash function is completely random, but can be extended also to **5-independent** hash functions.<sup>[3]</sup> Weaker properties, such as **universal hashing**, are not strong enough to ensure the constant-time operation of linear probing,<sup>[4]</sup> but one practical method of hash function generation, **tabulation hashing**, again leads to a guaranteed constant expected time performance despite not being 5-independent.<sup>[5]</sup>

### 11.5.4 See also

- **Quadratic probing**
- **Collision resolution**

### 11.5.5 References

- [1] Dale, Nell (2003). *C++ Plus Data Structures*. Sudbury, MA: Jones and Bartlett Computer Science. ISBN 0-7637-0481-4.

- [2] Knuth, Donald (1963), *Notes on “Open” Addressing*
- [3] Pagh, Anna; Pagh, Rasmus; Ružić, Milan (2009), “Linear probing with constant independence”, *SIAM Journal on Computing* **39** (3): 1107–1120, doi:10.1137/070702278, MR 2538852
- [4] Pătrașcu, Mihai; Thorup, Mikkel (2010), “On the  $k$ -independence required by linear probing and minwise independence”, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6–10, 2010, Proceedings, Part I* (PDF), Lecture Notes in Computer Science **6198**, Springer, pp. 715–726, doi:10.1007/978-3-642-14165-2\_60
- [5] Pătrașcu, Mihai; Thorup, Mikkel (2011), “The power of simple tabulation hashing”, *Proceedings of the 43rd annual ACM Symposium on Theory of Computing (STOC ’11)*, pp. 1–10, arXiv:1011.5200, doi:10.1145/1993636.1993638

### 11.5.6 External links

- How Caching Affects Hashing by Gregory L. Heileman and Wenbin Luo 2005.
- Open Data Structures - Section 5.2 - LinearHashTable: Linear Probing

## 11.6 Quadratic probing

**Quadratic probing** is an open addressing scheme in computer programming for resolving collisions in **hash tables**—when an incoming data’s hash value indicates it should be stored in an already-occupied slot or bucket. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

For a given hash value, the indices generated by linear probing are as follows:

$$H + 1, H + 2, H + 3, H + 4, \dots, H + k$$

This method results in **primary clustering**, and as the cluster grows larger, the search for those items hashing within the cluster becomes less efficient.

An example sequence using quadratic probing is:

$$H + 1^2, H + 2^2, H + 3^2, H + 4^2, \dots, H + k^2$$

Quadratic probing can be a more efficient algorithm in a closed hash table, since it better avoids the clustering problem that can occur with linear probing, although it is not immune. It also provides good memory caching because it preserves some **locality of reference**; however, linear probing has greater locality and, thus, better cache performance.

Quadratic probing is used in the **Berkeley Fast File System** to allocate free blocks. The allocation routine chooses a new cylinder-group when the current is nearly

full using quadratic probing, because of the speed it shows in finding unused cylinder-groups.

### 11.6.1 Quadratic function

Let  $h(k)$  be a **hash function** that maps an element  $k$  to an integer in  $[0, m-1]$ , where  $m$  is the size of the table. Let the  $i^{\text{th}}$  probe position for a value  $k$  be given by the function

$$h(k, i) = (h(k) + c_1 i + c_2 i^2) \pmod{m}$$

where  $c_2 \neq 0$ . If  $c_2 = 0$ , then  $h(k, i)$  degrades to a linear probe. For a given **hash table**, the values of  $c_1$  and  $c_2$  remain constant.

#### Examples:

- If  $h(k, i) = (h(k) + i + i^2) \pmod{m}$ , then the probe sequence will be  $h(k), h(k) + 2, h(k) + 6, \dots$
- For  $m = 2^n$ , a good choice for the constants are  $c_1 = c_2 = 1/2$ , as the values of  $h(k, i)$  for  $i$  in  $[0, m-1]$  are all distinct. This leads to a probe sequence of  $h(k), h(k) + 1, h(k) + 3, h(k) + 6, \dots$  where the values increase by 1, 2, 3, ...
- For prime  $m > 2$ , most choices of  $c_1$  and  $c_2$  will make  $h(k, i)$  distinct for  $i$  in  $[0, (m-1)/2]$ . Such choices include  $c_1 = c_2 = 1/2$ ,  $c_1 = c_2 = 1$ , and  $c_1 = 0, c_2 = 1$ . Because there are only about  $m/2$  distinct probes for a given element, it is difficult to guarantee that insertions will succeed when the load factor is  $> 1/2$ .

### 11.6.2 Quadratic probing insertion

The problem, here, is to insert a key at an available key space in a given Hash Table using quadratic probing.<sup>[1]</sup>

#### Algorithm to insert key in hash table

1. Get the key  $k$
2. Set counter  $j = 0$
3. Compute hash function  $h[k] = k \% \text{SIZE}$
4. If  $\text{hashtable}[h[k]]$  is empty
  - (4.1) Insert key  $k$  at  $\text{hashtable}[h[k]]$
  - (4.2) Stop Else
  - (4.3) The key space at  $\text{hashtable}[h[k]]$  is occupied, so we need to find the next available key space
  - (4.4) Increment  $j$
  - (4.5) Compute new hash function  $h[k] = (k + j * j) \% \text{SIZE}$
  - (4.6) Repeat Step 4 till  $j$  is equal to the **SIZE** of hash table
5. The hash table is full
6. Stop

#### C function for key insertion

```
int quadratic_probing_insert(int *hashtable, int key, int
empty){ / hashtable[] is an integer hash table; empty[] is another array which indicates whether the key space is occupied; If an empty key space is found, the function
```

```

returns the index of the bucket where the key is inserted,
otherwise it returns (-1) if no empty key space is found
/* int j = 0, hk; hk = key % SIZE; while(j < SIZE) {
if(empty[hk] == 1){ hashtable[hk] = key; empty[hk] =
0; return (hk); } j++; hk = (key + j * j) % SIZE; } return
(-1); }

```

### 11.6.3 Quadratic probing search

#### Algorithm to search element in hash table

1. Get the key  $k$  to be searched
2. Set counter  $j = 0$
3. Compute hash function  $h[k] = k \% \text{SIZE}$
4. If the key space at  $\text{hashtable}[h[k]]$  is occupied (4.1) Compare the element at  $\text{hashtable}[h[k]]$  with the key  $k$ .
- (4.2) If they are equal (4.2.1) The key is found at the bucket  $h[k]$  (4.2.2) Stop Else (4.3) The element might be placed at the next location given by the quadratic function (4.4) Increment  $j$
- (4.5) Compute new hash function  $h[k] = (k + j * j) \% \text{SIZE}$
- (4.6) Repeat Step 4 till  $j$  is greater than  $\text{SIZE}$  of hash table 5. The key was not found in the hash table 6. Stop

#### C function for key searching

```

int quadratic_probing_search(int *hashtable, int key, int
empty) { / If the key is found in the hash table, the
function returns the index of the hashtable where the key
is inserted, otherwise it returns (-1) if the key is not
found */ int j = 0, hk; hk = key % SIZE; while(j < SIZE)
{ if((empty[hk] == 0) && (hashtable[hk] == key)) re-
turn (hk); j++; hk = (key + j * j) % SIZE; } return (-1); }

```

### 11.6.4 Limitations

<sup>[2]</sup> For linear probing it is a bad idea to let the hash table get nearly full, because performance is degraded as the hash table gets filled. In the case of quadratic probing, the situation is even more drastic. With the exception of the triangular number case for a power-of-two-sized hash table, there is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve collisions. If the hash table size is  $b$  (a prime greater than 3), it can be proven that the first  $b/2$  alternative locations including the initial location  $h(k)$  are all distinct and unique. Suppose, we assume two of the alternative locations to be given by  $h(k) + x^2 \pmod{b}$  and  $h(k) + y^2 \pmod{b}$ , where  $0 \leq x, y \leq (b/2)$ . If these two locations point to the same key space, but  $x \neq y$ . Then the following would have to be true,

$$h(k) + x^2 \equiv h(k) + y^2 \pmod{b} \quad x^2 \equiv y^2 \pmod{b} \quad x^2 - y^2 \equiv 0 \pmod{b} \quad (x - y)(x + y) \equiv 0 \pmod{b}$$

As  $b$  (table size) is a prime greater than 3, either  $(x - y)$  or  $(x + y)$  has to be equal to zero. Since  $x$  and  $y$  are unique,  $(x - y)$  cannot be zero. Also, since  $0 \leq x, y \leq (b/2)$ ,  $(x + y)$  cannot be zero.

Thus, by contradiction, it can be said that the first  $(b/2)$  alternative locations after  $h(k)$  are unique. So an empty key space can always be found as long as at most  $(b/2)$  locations are filled, i.e., the hash table is not more than half full.

#### Alternating sign

If the sign of the offset is alternated (e.g. +1, -4, +9, -16 etc.), and if the number of buckets is a prime number  $p$  congruent to 3 modulo 4 (i.e. one of 3, 7, 11, 19, 23, 31 and so on), then the first  $p$  offsets will be unique modulo  $p$ .

In other words, a permutation of 0 through  $p-1$  is obtained, and, consequently, a free bucket will always be found as long as there exists at least one.

The insertion algorithm only receives a minor modification (but do note that  $\text{SIZE}$  has to be a suitable prime number as explained above):

1. Get the key  $k$
2. Set counter  $j = 0$
3. Compute hash function  $h[k] = k \% \text{SIZE}$
4. If  $\text{hashtable}[h[k]]$  is empty (4.1) Insert key  $k$  at  $\text{hashtable}[h[k]]$  (4.2) Stop Else (4.3) The key space at  $\text{hashtable}[h[k]]$  is occupied, so we need to find the next available key space (4.4) Increment  $j$
- (4.5) Compute new hash function  $h[k]$ . If  $j$  is odd, then  $h[k] = (k + j * j) \% \text{SIZE}$ , else  $h[k] = (k - j * j) \% \text{SIZE}$  (4.6) Repeat Step 4 till  $j$  is equal to the  $\text{SIZE}$  of hash table 5. The hash table is full 6. Stop

The search algorithm is modified likewise.

### 11.6.5 See also

- Hash tables
- Hash collision
- Double hashing
- Linear probing
- Hash function

### 11.6.6 References

- [1] Horowitz, Sahni, Anderson-Freed (2011). *Fundamentals of Data Structures in C*. University Press. ISBN 978-81-7371-605-8.
- [2] *Data Structures and Algorithm Analysis in C++*. Pearson Education. 2009. ISBN 978-81-317-1474-4. If `first1=missing` `last1=` in Authors list (help)

### 11.6.7 External links

- Tutorial/quadratic probing

## 11.7 Double hashing

**Double hashing** is a computer programming technique used in hash tables to resolve hash collisions, cases when two different values to be searched for produce the same hash key. It is a popular collision-resolution technique in open-addressed hash tables. Double hashing is implemented in many popular libraries.

Like linear probing, it uses one hash value as a starting point and then repeatedly steps forward an interval until the desired value is located, an empty location is reached, or the entire table has been searched; but this interval is decided using a second, independent hash function (hence the name double hashing). Unlike linear probing and quadratic probing, the interval depends on the data, so that even values mapping to the same location have different bucket sequences; this minimizes repeated collisions and the effects of clustering.

Given two randomly, uniformly, and independently selected hash functions  $h_1$  and  $h_2$ , the  $i$ th location in the bucket sequence for value  $k$  in a hash table  $T$  is:  $h(i, k) = (h_1(k) + i \cdot h_2(k)) \bmod |T|$ . Generally,  $h_1$  and  $h_2$  are selected from a set of universal hash functions.

### 11.7.1 Classical applied data structure

Double hashing with open addressing is a classical data structure on a table  $T$ . Let  $n$  be the number of elements stored in  $T$ , then  $T$ 's load factor is  $\alpha = \frac{n}{|T|}$ .

Double hashing approximates uniform open address hashing. That is, start by randomly, uniformly and independently selecting two universal hash functions  $h_1$  and  $h_2$  to build a double hashing table  $T$ .

All elements are put in  $T$  by double hashing using  $h_1$  and  $h_2$ . Given a key  $k$ , determining the  $(i + 1)$ -st hash location is computed by:

$$h(i, k) = (h_1(k) + i \cdot h_2(k)) \bmod |T|.$$

Let  $T$  have fixed load factor  $\alpha : 1 > \alpha > 0$ . Bradford and Katehakis<sup>[1]</sup> showed the expected number of probes for an unsuccessful search in  $T$ , still using these initially chosen hash functions, is  $\frac{1}{1-\alpha}$  regardless of the distribution of the inputs. More precisely, these two uniformly, randomly and independently chosen hash functions are chosen from a set of universal hash functions where pairwise independence suffices.

Previous results include: Guibas and Szemerédi<sup>[2]</sup> showed  $\frac{1}{1-\alpha}$  holds for unsuccessful search for load factors  $\alpha < 0.319$ . Also, Lueker and Molodowitch<sup>[3]</sup> showed this held assuming ideal randomized functions. Schmidt

and Siegel<sup>[4]</sup> showed this with  $k$ -wise independent and uniform functions (for  $k = c \log n$ , and suitable constant  $c$ ).

### 11.7.2 Implementation details for caching

Linear probing and, to a lesser extent, quadratic probing are able to take advantage of the data cache by accessing locations that are close together. Double hashing has, on average, larger intervals and is not able to achieve this advantage.

Like all other forms of open addressing, double hashing becomes linear as the hash table approaches maximum capacity. The only solution to this is to rehash to a larger size, as with all other open addressing schemes.

On top of that, it is possible for the secondary hash function to evaluate to zero. For example, if we choose  $k=5$  with the following function:

$$h_2(k) = 5 - (k \bmod 7)$$

The resulting sequence will always remain at the initial hash value. One possible solution is to change the secondary hash function to:

$$h_2(k) = (k \bmod 7) + 1$$

This ensures that the secondary hash function will always be non zero.

### 11.7.3 See also

- Hash\_table#Collision\_resolution
- Hash function
- Linear probing
- Cuckoo hashing

### 11.7.4 Notes

- [1] P. G. Bradford and M. Katehakis: *A Probabilistic Study on Combinatorial Expanders and Hashing*, SIAM Journal on Computing 2007 (37:1), 83-111. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.2647>
- [2] L. Guibas and E. Szemerédi: *The Analysis of Double Hashing*, Journal of Computer and System Sciences, 1978, 16, 226-274.
- [3] G. S. Lueker and M. Molodowitch: *More Analysis of Double Hashing*, Combinatorica, 1993, 13(1), 83-96.
- [4] J. P. Schmidt and A. Siegel: *Double Hashing is Computable and Randomizable with Universal Hash Functions*, manuscript.

### 11.7.5 External links

- How Caching Affects Hashing by Gregory L. Heileman and Wenbin Luo 2005.
- Hash Table Animation
- klib a C library that includes double hashing functionality.

## 11.8 Cuckoo hashing

**Cuckoo hashing** is a scheme in computer programming for resolving **hash collisions** of values of hash functions in a **table**, with **worst-case constant** lookup time. The name derives from the behavior of some species of **cuckoo**, where the cuckoo chick pushes the other eggs or young out of the nest when it hatches; analogously, inserting a new key into a cuckoo hashing table may push an older key to a different location in the table.

### 11.8.1 History

Cuckoo hashing was first described by Rasmus Pagh and Flemming Friche Rodler in 2001.<sup>[1]</sup>

### 11.8.2 Theory

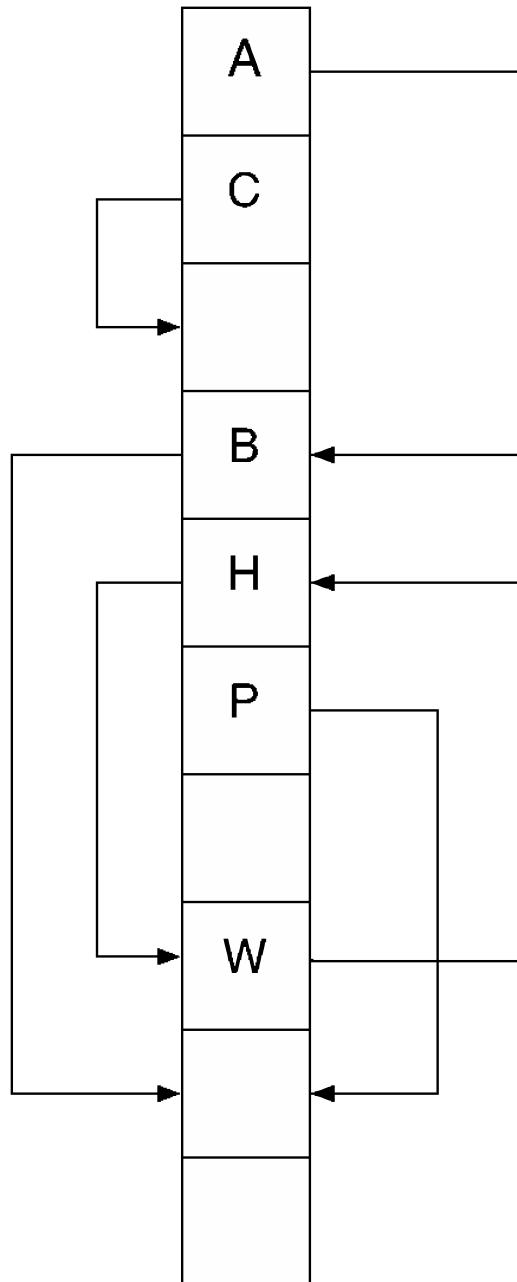
The basic idea is to use two hash functions instead of only one. This provides two possible locations in the hash table for each **key**. In one of the commonly used variants of the algorithm, the hash table is split into two smaller tables of equal size, and each hash function provides an index into one of these two tables.

When a new key is inserted, a **greedy algorithm** is used: The new key is inserted in one of its two possible locations, “kicking out”, that is, displacing, any key that might already reside in this location. This displaced key is then inserted in its alternative location, again kicking out any key that might reside there, until a vacant position is found, or the procedure would enter an **infinite loop**. In the latter case, the **hash table** is rebuilt in-place using new **hash functions**:

There is no need to allocate new tables for the rehashing: We may simply run through the tables to delete and perform the usual insertion procedure on all keys found not to be at their intended position in the table.

—Pagh & Rodler, “Cuckoo Hashing”<sup>[1]</sup>

Lookup requires inspection of just two locations in the hash table, which takes constant time in the worst case (see **Big O** notation). This is in contrast to many other



*Cuckoo hashing example. The arrows show the alternative location of each key. A new item would be inserted in the location of A by moving A to its alternative location, currently occupied by B, and moving B to its alternative location which is currently vacant. Insertion of a new item in the location of H would not succeed: Since H is part of a cycle (together with W), the new item would get kicked out again.*

hash table algorithms, which may not have a constant worst-case bound on the time to do a lookup.

It can also be shown that insertions succeed in expected constant time,<sup>[1]</sup> even considering the possibility of having to rebuild the table, as long as the number of keys is kept below half of the capacity of the hash table, i.e., the load factor is below 50%. One method of proving

this uses the theory of [random graphs](#): one may form an [undirected graph](#) called the “Cuckoo Graph” that has a vertex for each hash table location, and an edge for each hashed value, with the endpoints of the edge being the two possible locations of the value. Then, the greedy insertion algorithm for adding a set of values to a cuckoo hash table succeeds if and only if the Cuckoo Graph for this set of values is a [pseudoforest](#), a graph with at most one cycle in each of its [connected components](#), as any vertex-induced subgraph with more edges than vertices corresponds to a set of keys for which there are an insufficient number of slots in the hash table. This property is true with high probability for a random graph in which the number of edges is less than half the number of vertices.<sup>[2]</sup>

### 11.8.3 Example

The following hashfunctions are given:

$$h(k) = k \bmod 11$$

$$h'(k) = \lfloor \frac{k}{11} \rfloor \bmod 11$$

Columns in the following two tables show the state of the hash tables over time as the elements are inserted.

#### Cycle

If you now wish to insert the element 6, then you get into a cycle. In the last row of the table we find the same initial situation as at the beginning again.

$$h(6) = 6 \bmod 11 = 6$$

$$h'(6) = \lfloor \frac{6}{11} \rfloor \bmod 11 = 0$$

### 11.8.4 Generalizations and applications

Generalizations of cuckoo hashing that use more than 2 alternative hash functions can be expected to utilize a larger part of the capacity of the hash table efficiently while sacrificing some lookup and insertion speed. Using just three hash functions increases the load to 91%.<sup>[3]</sup> Another generalization of cuckoo hashing consists in using more than one key per bucket. Using just 2 keys per bucket permits a load factor above 80%.

Other algorithms that use multiple hash functions include the [Bloom filter](#). A Cuckoo Filter employs Cuckoo hashing principles to implement a data structure equivalent to a Bloom filter.<sup>[4]</sup> Some people recommend a simplified generalization of cuckoo hashing called [skewed-associative cache](#) in some CPU caches.<sup>[5]</sup>

A study by Zukowski et al.<sup>[6]</sup> has shown that cuckoo hashing is much faster than [chained hashing](#) for small, [cache-resident](#) hash tables on modern processors. Kenneth Ross<sup>[7]</sup> has shown bucketized versions of cuckoo hashing (variants that use buckets that contain more than

one key) to be faster than conventional methods also for large hash tables, when space utilization is high. The performance of the bucketized cuckoo hash table was investigated further by Askitis,<sup>[8]</sup> with its performance compared against alternative hashing schemes.

A survey by Mitzenmacher<sup>[3]</sup> presents open problems related to cuckoo hashing as of 2009.

### 11.8.5 See also

- Perfect hashing
- Linear probing
- Double hashing
- Hash collision
- Hash function
- Quadratic probing
- Hopscotch hashing

### 11.8.6 References

- [1] Pagh, Rasmus; Rodler, Flemming Friche (2001). “Cuckoo Hashing”. *Algorithms — ESA 2001*. Lecture Notes in Computer Science **2161**. pp. 121–133. doi:10.1007/3-540-44676-1\_10. ISBN 978-3-540-42493-2.
- [2] Kutzelnigg, Reinhard (2006). *Bipartite random graphs and cuckoo hashing*. Fourth Colloquium on Mathematics and Computer Science. Discrete Mathematics and Theoretical Computer Science **AG**. pp. 403–406
- [3] Mitzenmacher, Michael (2009-09-09). “Some Open Questions Related to Cuckoo Hashing | Proceedings of ESA 2009” (PDF). Retrieved 2010-11-10.
- [4] Fan, Bin; Kaminsky, Michael; Andersen, David (August 2013). “Cuckoo Filter: Better Than Bloom” (PDF). *login*: (USENIX) **38** (4): 36–40. Retrieved 12 June 2014.
- [5] “Micro-Architecture”.
- [6] Zukowski, Marcin; Heman, Sandor; Boncz, Peter (June 2006). “Architecture-Conscious Hashing” (PDF). Proceedings of the International Workshop on Data Management on New Hardware (DaMoN). Retrieved 2008-10-16.
- [7] Ross, Kenneth (2006-11-08). “Efficient Hash Probes on Modern Processors” (PDF). IBM Research Report RC24100. RC24100. Retrieved 2008-10-16.
- [8] Askitis, Nikolas (2009). *Fast and Compact Hash Tables for Integer Keys* (PDF). Proceedings of the 32nd Australasian Computer Science Conference (ACSC 2009) **91**. pp. 113–122. ISBN 978-1-920682-72-9.

### 11.8.7 External links

- A cool and practical alternative to traditional hash tables, U. Erlingsson, M. Manasse, F. Mcsherry, 2006.
- Cuckoo Hashing for Undergraduates, 2006, R. Pagh, 2006.
- Cuckoo Hashing, Theory and Practice (Part 1, Part 2 and Part 3), Michael Mitzenmacher, 2007.
- Naor, Moni; Segev, Gil; Wieder, Udi (2008). “History-Independent Cuckoo Hashing”. *International Colloquium on Automata, Languages and Programming (ICALP)*. Reykjavik, Iceland. Retrieved 2008-07-21.
- Algorithmic Improvements for Fast Concurrent Cuckoo Hashing, X. Li, D. Andersen, M. Kaminsky, M. Freedman. EuroSys 2014.

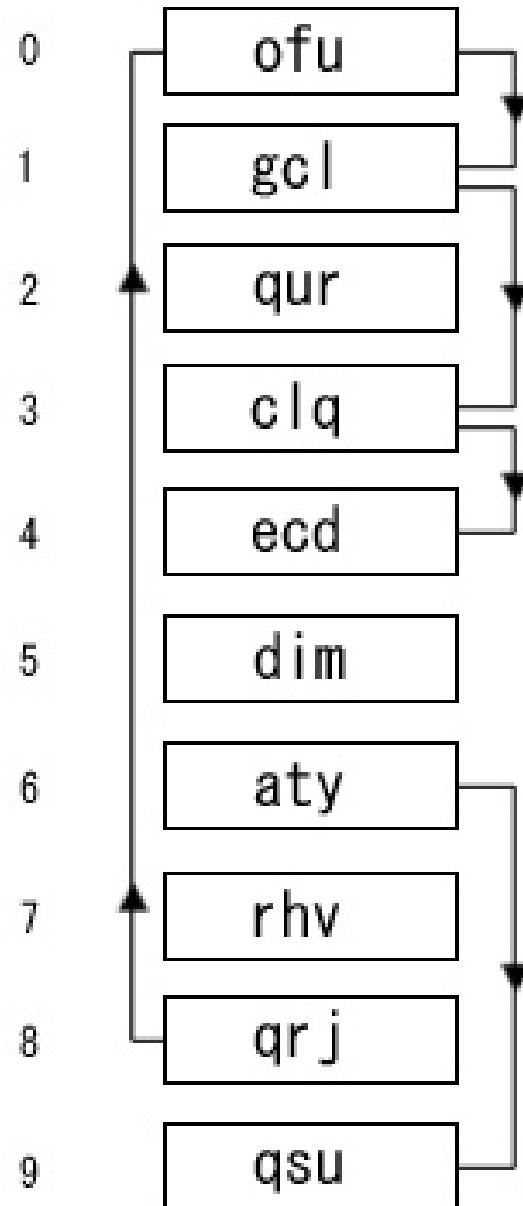
### Examples

- Concurrent high-performance Cuckoo hashtable written in C++
- Cuckoo hash map written in C++
- Static cuckoo hashtable generator for C/C++
- Cuckoo hashtable written in Java
- Generic Cuckoo hashmap in Java
- Cuckoo hash table written in Haskell
- Cuckoo hashing for Go

## 11.9 Coalesced hashing

**Coalesced hashing**, also called **coalesced chaining**, is a strategy of collision resolution in a hash table that forms a hybrid of **separate chaining** and **open addressing**. In a separate chaining hash table, items that hash to the same address are placed on a list (or “chain”) at that address. This technique can result in a great deal of wasted memory because the table itself must be large enough to maintain a load factor that performs well (typically twice the expected number of items), and extra memory must be used for all but the first item in a chain (unless list headers are used, in which case extra memory must be used for all items in a chain).

Given a sequence “qrj,” “aty,” “qur,” “dim,” “ofu,” “gcl,” “rvh,” “clq,” “ecd,” “qsu” of randomly generated three character long strings, the following table would be generated (using Bob Jenkins’ One-at-a-Time hash algorithm) with a table of size 10:



*Coalesced Hashing example. For purposes of this example, collision buckets are allocated in increasing order, starting with bucket 0.*

This strategy is effective, efficient, and very easy to implement. However, sometimes the extra memory use might be prohibitive, and the most common alternative, open addressing, has uncomfortable disadvantages that decrease performance. The primary disadvantage of open addressing is primary and secondary clustering, in which searches may access long sequences of used buckets that contain items with different hash addresses; items with one hash address can thus lengthen searches for items with other hash addresses.

One solution to these issues is coalesced hashing. Coalesced hashing uses a similar technique as separate chaining, but instead of allocating new nodes for the linked

list, buckets in the actual table are used. The first empty bucket in the table at the time of a collision is considered the collision bucket. When a collision occurs anywhere in the table, the item is placed in the collision bucket and a link is made between the chain and the collision bucket. It is possible for a newly inserted item to collide with items with a different hash address, such as the case in the example above when item “clq” is inserted. The chain for “clq” is said to “coalesce” with the chain of “qrj,” hence the name of the algorithm. However, the extent of coalescing is minor compared with the clustering exhibited by open addressing. For example, when coalescing occurs, the length of the chain grows by only 1, whereas in open addressing, search sequences of arbitrary length may combine.

An important optimization, to reduce the effect of coalescing, is to restrict the address space of the hash function to only a subset of the table. For example, if the table has size  $M$  with buckets numbered from 0 to  $M - 1$ , we can restrict the address space so that the hash function only assigns addresses to the first  $N$  locations in the table. The remaining  $M - N$  buckets, called the *cellar*, are used exclusively for storing items that collide during insertion. No coalescing can occur until the cellar is exhausted.

The optimal choice of  $N$  relative to  $M$  depends upon the load factor (or fullness) of the table. A careful analysis shows that the value  $N = 0.86 \times M$  yields near-optimum performance for most load factors.<sup>[1]</sup> Other variants for insertion are also possible that have improved search time. Deletion algorithms have been developed that preserve randomness, and thus the average search time analysis still holds after deletions.<sup>[1]</sup>

Insertion in C:

```
/* htab is the hash table, N is the size of the address
space of the hash function, and M is the size of the entire
table including the cellar. Collision buckets are allocated
in decreasing order, starting with bucket M-1. */ int
insert (char key[]) { unsigned h = hash (key, strlen (key)
) % N; if (htab[h] == NULL) { /* Make a new chain
*/ htab[h] = make_node (key, NULL); } else { struct
node *it; int cursor = M-1; /* Find the first empty bucket
*/ while (cursor >= 0 && htab[cursor] != NULL)
--cursor; /* The table is full, terminate unsuccessfully */
if (cursor == -1) return -1; htab[cursor] = make_node
(key, NULL); /* Find the last node in the chain and
point to it */ it = htab[h]; while (it->next != NULL)
it = it->next; it->next = htab[cursor]; } return 0; }
```

One benefit of this strategy is that the search algorithm for separate chaining can be used without change in a coalesced hash table.

Lookup in C:

```
char *find (char key[]) { unsigned h = hash (key, strlen
(key)) % N; if (htab[h] != NULL) { struct node *it;
/* Search the chain at index h */ for (it = htab[h]; it !=
```

```
NULL; it = it->next) { if (strcmp (key, it->data) == 0
) return it->data; } } return NULL; }
```

## 11.9.1 Performance

Coalesced chaining avoids the effects of primary and secondary clustering, and as a result can take advantage of the efficient search algorithm for separate chaining. If the chains are short, this strategy is very efficient and can be highly condensed, memory-wise. As in open addressing, deletion from a coalesced hash table is awkward and potentially expensive, and resizing the table is terribly expensive and should be done rarely, if ever.

## 11.9.2 References

- [1] J. S. Vitter and W.-C. Chen, *Design and Analysis of Coalesced Hashing*, Oxford University Press, New York, NY, 1987, ISBN 0-19-504182-8

# 11.10 Perfect hash function

A **perfect hash function** for a set  $S$  is a **hash function** that maps distinct elements in  $S$  to a set of integers, with no collisions. A perfect hash function has many of the same **applications** as other hash functions, but with the advantage that no collision resolution has to be implemented. In mathematical terms, it is a **total injective function**.

## 11.10.1 Properties and uses

A perfect hash function for a specific set  $S$  that can be evaluated in constant time, and with values in a small range, can be found by a **randomized algorithm** in a number of operations that is proportional to the size of  $S$ .<sup>[1]</sup> Any perfect hash functions suitable for use with a **hash table** require at least a number of bits that is proportional to the size of  $S$ .

A perfect hash function with values in a limited **range** can be used for efficient lookup operations, by placing keys from  $S$  (or other associated values) in a **table** indexed by the output of the function. Using a perfect hash function is best in situations where there is a frequently queried large set,  $S$ , which is seldom updated. This is because any modification of the set leads to a non-perfect hash function. Solutions which update the hash function any time the set is modified are known as **dynamic perfect hashing**, but these methods are relatively complicated to implement. A simple alternative to perfect hashing, which also allows dynamic updates, is **cuckoo hashing**.

### 11.10.2 Minimal perfect hash function

A **minimal perfect hash function** is a perfect hash function that maps  $n$  keys to  $n$  consecutive integers—usually  $[0..n-1]$  or  $[1..n]$ . A more formal way of expressing this is: Let  $j$  and  $k$  be elements of some finite set  $\mathbf{K}$ .  $F$  is a minimal perfect hash function iff  $F(j) = F(k)$  implies  $j=k$  (injectivity) and there exists an integer  $a$  such that the range of  $F$  is  $a..a+|\mathbf{K}|-1$ . It has been proved that a general purpose minimal perfect hash scheme requires at least  $1.44$  bits/key.<sup>[2]</sup> The best currently known minimal perfect hashing schemes use around  $2.6$  bits/key.<sup>[3]</sup>

A minimal perfect hash function  $F$  is **order preserving** if keys are given in some order  $a_1, a_2, \dots, a_n$  and for any keys  $a_j$  and  $a_k$ ,  $j < k$  implies  $F(a_j) < F(a_k)$ .<sup>[4]</sup> Order-preserving minimal perfect hash functions require necessarily  $\Omega(n \log n)$  bits to be represented.<sup>[5]</sup>

A minimal perfect hash function  $F$  is **monotone** if it preserves the **lexicographical order** of the keys. In this case, the function value is just the position of each key in the sorted ordering of all of the keys. If the keys to be hashed are themselves stored in a sorted array, it is possible to store a small number of additional bits per key in a data structure that can be used to compute hash values quickly.<sup>[6]</sup>

### 11.10.3 See also

- Dynamic perfect hashing
- Pearson hashing
- Universal hashing

### 11.10.4 References

- [1] Fredman, M. L.; Komlós, J. N.; Szemerédi, E. (1984). “Storing a Sparse Table with  $O(1)$  Worst Case Access Time”. *Journal of the ACM* **31** (3): 538. doi:10.1145/828.1884.
- [2] Belazzougui, D.; Botelho, F. C.; Dietzfelbinger, M. (2009). “Hash, Displace, and Compress”. *Algorithms - ESA 2009* (PDF). LNCS **5757**. p. 682. doi:10.1007/978-3-642-04128-0\_61. ISBN 978-3-642-04127-3.
- [3] Baeza-Yates, Ricardo; Poblete, Patricio V. (2010), “Searching”, in Atallah, Mikhail J.; Blanton, Marina, *Algorithms and Theory of Computation Handbook: General Concepts and Techniques* (2nd ed.), CRC Press, ISBN 9781584888239. See in particular p. 2-10.
- [4] Jenkins, Bob (14 April 2009), “order-preserving minimal perfect hashing”, in Black, Paul E., *Dictionary of Algorithms and Data Structures*, U.S. National Institute of Standards and Technology, retrieved 2013-03-05
- [5] Fox, E. A.; Chen, Q. F.; Daoud, A. M.; Heath, L. S. (1990). “Order preserving minimal perfect hash functions and information retrieval”. *Proceedings of the 13th annual*

*international ACM SIGIR conference on Research and development in information retrieval - SIGIR '90.* p. 279. doi:10.1145/96749.98233. ISBN 0897914082.

- [6] Belazzougui, Djamal; Boldi, Paolo; Pagh, Rasmus; Vigna, Sebastiano (November 2008), “Theory and practice of monotone minimal perfect hashing”, *Journal of Experimental Algorithms* **16**, Art. no. 3.2, 26pp, doi:10.1145/1963190.2025378.

### 11.10.5 Further reading

- Richard J. Cichelli. *Minimal Perfect Hash Functions Made Simple*, Communications of the ACM, Vol. 23, Number 1, January 1980.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 11.5: Perfect hashing, pp. 245–249.
- Fabiano C. Botelho, Rasmus Pagh and Nivio Ziviani. “Perfect Hashing for Data Management Applications”.
- Fabiano C. Botelho and Nivio Ziviani. “External perfect hashing for very large key sets”. 16th ACM Conference on Information and Knowledge Management (CIKM07), Lisbon, Portugal, November 2007.
- Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. “Monotone minimal perfect hashing: Searching a sorted table with  $O(1)$  accesses”. In Proceedings of the 20th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA), New York, 2009. ACM Press.
- Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. “Theory and practise of monotone minimal perfect hashing”. In Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, 2009.
- Douglas C. Schmidt, **GPERF: A Perfect Hash Function Generator**, C++ Report, SIGS, Vol. 10, No. 10, November/December, 1998.

### 11.10.6 External links

- Minimal Perfect Hashing by Bob Jenkins
- **gperf** is an Open Source C and C++ perfect hash generator
- **cmph** is Open Source implementing many perfect hashing methods
- **Sux4J** is Open Source implementing perfect hashing, including monotone minimal perfect hashing in Java

- MPHSharp is Open Source implementing many perfect hashing methods in C#

$\forall x, y \in U, x \neq y$  , when  $h$  is drawn randomly from the family  $H$  , the difference  $h(x) - h(y) \bmod m$  is uniformly distributed in  $[m]$  .

## 11.11 Universal hashing

Using **universal hashing** (in a randomized algorithm or data structure) refers to selecting a **hash function** at random from a family of hash functions with a certain mathematical property (see definition below). This guarantees a low number of collisions in **expectation**, even if the data is chosen by an adversary. Many universal families are known (for hashing integers, vectors, strings), and their evaluation is often very efficient. Universal hashing has numerous uses in computer science, for example in implementations of **hash tables**, randomized algorithms, and **cryptography**.

### 11.11.1 Introduction

See also: **Hash function**

Assume we want to map keys from some universe  $U$  into  $m$  bins (labelled  $[m] = \{0, \dots, m-1\}$  ). The algorithm will have to handle some data set  $S \subseteq U$  of  $|S| = n$  keys, which is not known in advance. Usually, the goal of hashing is to obtain a low number of collisions (keys from  $S$  that land in the same bin). A deterministic hash function cannot offer any guarantee in an adversarial setting if the size of  $U$  is greater than  $m \cdot n$  , since the adversary may choose  $S$  to be precisely the **preimage** of a bin. This means that all data keys land in the same bin, making hashing useless. Furthermore, a deterministic hash function does not allow for **rehashing**: sometimes the input data turns out to be bad for the hash function (e.g. there are too many collisions), so one would like to change the hash function.

The solution to these problems is to pick a function randomly from a family of hash functions. A family of functions  $H = \{h : U \rightarrow [m]\}$  is called a **universal family** if,  $\forall x, y \in U, x \neq y : \Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}$  .

In other words, any two keys of the universe collide with probability at most  $1/m$  when the hash function  $h$  is drawn randomly from  $H$  . This is exactly the probability of collision we would expect if the hash function assigned truly random hash codes to every key. Sometimes, the definition is relaxed to allow collision probability  $O(1/m)$  . This concept was introduced by Carter and Wegman<sup>[1]</sup> in 1977, and has found numerous applications in computer science (see, for example <sup>[2]</sup>). If we have an upper bound of  $\epsilon < 1$  on the collision probability, we say that we have  $\epsilon$  -almost universality.

Many, but not all, universal families have the following stronger **uniform difference property**:

Note that the definition of universality is only concerned with whether  $h(x) - h(y) = 0$  , which counts collisions. The uniform difference property is stronger.

(Similarly, a universal family can be XOR universal if  $\forall x, y \in U, x \neq y$  , the value  $h(x) \oplus h(y) \bmod m$  is uniformly distributed in  $[m]$  where  $\oplus$  is the bitwise exclusive or operation. This is only possible if  $m$  is a power of two.)

An even stronger condition is **pairwise independence**: we have this property when  $\forall x, y \in U, x \neq y$  we have the probability that  $x, y$  will hash to any pair of hash values  $z_1, z_2$  is as if they were perfectly random:  $P(h(x) = z_1 \wedge h(y) = z_2) = 1/m^2$  . Pairwise independence is sometimes called **strong universality**.

Another property is **uniformity**. We say that a family is uniform if all hash values are equally likely:  $P(h(x) = z) = 1/m$  for any hash value  $z$  . Universality does not imply uniformity. However, strong universality does imply uniformity.

Given a family with the uniform distance property, one can produce a pairwise independent or strongly universal hash family by adding a uniformly distributed random constant with values in  $[m]$  to the hash functions. (Similarly, if  $m$  is a power of two, we can achieve pairwise independence from an XOR universal hash family by doing an exclusive or with a uniformly distributed random constant.) Since a shift by a constant is sometimes irrelevant in applications (e.g. hash tables), a careful distinction between the uniform distance property and pairwise independent is sometimes not made.<sup>[3]</sup>

For some applications (such as hash tables), it is important for the least significant bits of the hash values to be also universal. When a family is strongly universal, this is guaranteed: if  $H$  is a strongly universal family with  $m = 2^L$  , then the family made of the functions  $h \bmod 2^{L'}$  for all  $h \in H$  is also strongly universal for  $L' \leq L$  . Unfortunately, the same is not true of (merely) universal families. For example the family made of the identity function  $h(x) = x$  is clearly universal, but the family made of the function  $h(x) = x \bmod 2^{L'}$  fails to be universal.

### 11.11.2 Mathematical guarantees

For any fixed set  $S$  of  $n$  keys, using a universal family guarantees the following properties.

1. For any fixed  $x$  in  $S$  , the expected number of keys in the bin  $h(x)$  is  $n/m$  . When implementing hash tables by **chaining**, this number is proportional to the expected running time of an operation involving the key  $x$  (for example a query, insertion or deletion).

2. The expected number of pairs of keys  $x, y$  in  $S$  with  $x \neq y$  that collide ( $h(x) = h(y)$ ) is bounded above by  $n(n-1)/2m$ , which is of order  $O(n^2/m)$ . When the number of bins,  $m$ , is  $O(n)$ , the expected number of collisions is  $O(n)$ . When hashing into  $n^2$  bins, there are no collisions at all with probability at least a half.
3. The expected number of keys in bins with at least  $t$  keys in them is bounded above by  $2n/(t-2(n/m)+1)$ .<sup>[4]</sup> Thus, if the capacity of each bin is capped to three times the average size ( $t = 3n/m$ ), the total number of keys in overflowing bins is at most  $O(m)$ . This only holds with a hash family whose collision probability is bounded above by  $1/m$ . If a weaker definition is used, bounding it by  $O(1/m)$ , this result is no longer true.<sup>[4]</sup>

As the above guarantees hold for any fixed set  $S$ , they hold if the data set is chosen by an adversary. However, the adversary has to make this choice before (or independent of) the algorithm's random choice of a hash function. If the adversary can observe the random choice of the algorithm, randomness serves no purpose, and the situation is the same as deterministic hashing.

The second and third guarantee are typically used in conjunction with **rehashing**. For instance, a randomized algorithm may be prepared to handle some  $O(n)$  number of collisions. If it observes too many collisions, it chooses another random  $h$  from the family and repeats. Universality guarantees that the number of repetitions is a **geometric random variable**.

### 11.11.3 Constructions

Since any computer data can be represented as one or more machine words, one generally needs hash functions for three types of domains: machine words (“integers”); fixed-length vectors of machine words; and variable-length vectors (“strings”).

#### Hashing integers

This section refers to the case of hashing integers that fit in machine words; thus, operations like multiplication, addition, division, etc. are cheap machine-level instructions. Let the universe to be hashed be  $U = \{0, \dots, u-1\}$ .

The original proposal of Carter and Wegman<sup>[1]</sup> was to pick a prime  $p \geq u$  and define

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$$

where  $a, b$  are randomly chosen integers modulo  $p$  with  $a \neq 0$ . Technically, adding  $b$  is not needed for universality (but it does make the hash function 2-independent).

(This is a single iteration of a linear congruential generator).

To see that  $H = \{h_{a,b}\}$  is a universal family, note that  $h(x) = h(y)$  only holds when

$$ax + b \equiv ay + b + i \cdot m \pmod{p}$$

for some integer  $i$  between 0 and  $p/m$ . If  $x \neq y$ , their difference,  $x - y$  is nonzero and has an inverse modulo  $p$ . Solving for  $a$ ,

$$a \equiv i \cdot m \cdot (x - y)^{-1} \pmod{p}$$

There are  $p-1$  possible choices for  $a$  (since  $a=0$  is excluded) and, varying  $i$  in the allowed range,  $\lfloor p/m \rfloor$  possible values for the right hand side. Thus the collision probability is

$$\lfloor p/m \rfloor / (p-1)$$

which tends to  $1/m$  for large  $p$  as required. This analysis also shows that  $b$  does not have to be randomised in order to have universality.

Another way to see  $H$  is a universal family is via the notion of **statistical distance**. Write the difference  $h(x) - h(y)$  as

$$h(x) - h(y) \equiv (a(x - y) \bmod p) \pmod{m}$$

Since  $x - y$  is nonzero and  $a$  is uniformly distributed in  $\{1, \dots, p\}$ , it follows that  $a(x - y)$  modulo  $p$  is also uniformly distributed in  $\{1, \dots, p\}$ . The distribution of  $(h(x) - h(y)) \bmod m$  is thus almost uniform, up to a difference in probability of  $\pm 1/p$  between the samples. As a result, the statistical distance to a uniform family is  $O(m/p)$ , which becomes negligible when  $p \gg m$ .

**Avoiding modular arithmetic** The state of the art for hashing integers is the **multiply-shift** scheme described by Dietzfelbinger et al. in 1997.<sup>[5]</sup> By avoiding modular arithmetic, this method is much easier to implement and also runs significantly faster in practice (usually by at least a factor of four<sup>[6]</sup>). The scheme assumes the number of bins is a power of two,  $m = 2^M$ . Let  $w$  be the number of bits in a machine word. Then the hash functions are parametrised over odd positive integers  $a < 2^w$  (that fit in a word of  $w$  bits). To evaluate  $h_a(x)$ , multiply  $x$  by  $a$  modulo  $2^w$  and then keep the high order  $M$  bits as the hash code. In mathematical notation, this is

$$h_a(x) = (a \cdot x \bmod 2^w) \text{ div } 2^{w-M}$$

and it can be implemented in C-like programming languages by

$$h_a(x) = (\text{unsigned}) (a*x) \gg (w-M)$$

This scheme does *not* satisfy the uniform difference property and is only  $2/m$  -*almost-universal*; for any  $x \neq y$  ,  $\Pr\{h_a(x) = h_a(y)\} \leq 2/m$  .

To understand the behavior of the hash function, notice that, if  $ax \bmod 2^w$  and  $ay \bmod 2^w$  have the same highest-order 'M' bits, then  $a(x - y) \bmod 2^w$  has either all 1's or all 0's as its highest order M bits (depending on whether  $ax \bmod 2^w$  or  $ay \bmod 2^w$  is larger). Assume that the least significant set bit of  $x - y$  appears on position  $w - c$  . Since  $a$  is a random odd integer and odd integers have inverses in the **ring**  $Z_{2^w}$  , it follows that  $a(x - y) \bmod 2^w$  will be uniformly distributed among  $w$  -bit integers with the least significant set bit on position  $w - c$  . The probability that these bits are all 0's or all 1's is therefore at most  $2/2^M = 2/m$  . On the other hand, if  $c < M$  , then higher-order M bits of  $a(x - y) \bmod 2^w$  contain both 0's and 1's, so it is certain that  $h(x) \neq h(y)$  . Finally, if  $c = M$  then bit  $w - M$  of  $a(x - y) \bmod 2^w$  is 1 and  $h_a(x) = h_a(y)$  if and only if bits  $w - 1, \dots, w - M + 1$  are also 1, which happens with probability  $1/2^{M-1} = 2/m$  .

This analysis is tight, as can be shown with the example  $x = 2^{w-M-2}$  and  $y = 3x$  . To obtain a truly 'universal' hash function, one can use the multiply-add-shift scheme

$$h_{a,b}(x) = ((ax + b) \bmod 2^w) \div 2^{w-M}$$

which can be implemented in C-like programming languages by

$$h_{a,b}(x) = (\text{unsigned}) (a*x + b) \gg (w-M)$$

where  $a$  is a random odd positive integer with  $a < 2^w$  and  $b$  is a random non-negative integer with  $b < 2^{w-M}$  . With these choices of  $a$  and  $b$  ,  $\Pr\{h_{a,b}(x) = h_{a,b}(y)\} \leq 1/m$  for all  $x \not\equiv y \pmod{2^w}$  .<sup>[7]</sup> This differs slightly but importantly from the mistranslation in the English paper.<sup>[8]</sup>

### Hashing vectors

This section is concerned with hashing a fixed-length vector of machine words. Interpret the input as a vector  $\bar{x} = (x_0, \dots, x_{k-1})$  of  $k$  machine words (integers of  $w$  bits each). If  $H$  is a universal family with the uniform difference property, the following family (dating back to Carter and Wegman<sup>[1]</sup>) also has the uniform difference property (and hence is universal):

$$h(\bar{x}) = \left( \sum_{i=0}^{k-1} h_i(x_i) \right) \bmod m, \text{ where each } h_i \in H \text{ is chosen independently at random.}$$

If  $m$  is a power of two, one may replace summation by exclusive or.<sup>[9]</sup>

In practice, if double-precision arithmetic is available, this is instantiated with the multiply-shift hash family of.<sup>[10]</sup> Initialize the hash function with a vector  $\bar{a} = (a_0, \dots, a_{k-1})$  of random **odd** integers on  $2w$  bits each. Then if the number of bins is  $m = 2^M$  for  $M \leq w$  :

$$h_{\bar{a}}(\bar{x}) = \left( \left( \sum_{i=0}^{k-1} x_i \cdot a_i \right) \bmod 2^{2w} \right) \div 2^{2w-M}$$

It is possible to halve the number of multiplications, which roughly translates to a two-fold speed-up in practice.<sup>[9]</sup> Initialize the hash function with a vector  $\bar{a} = (a_0, \dots, a_{k-1})$  of random **odd** integers on  $2w$  bits each. The following hash family is universal:<sup>[11]</sup>

$$h_{\bar{a}}(\bar{x}) = \left( \left( \sum_{i=0}^{\lceil k/2 \rceil} (x_{2i} + a_{2i}) \cdot (x_{2i+1} + a_{2i+1}) \right) \bmod 2^{2w} \right) \div 2^{2w-M}$$

If double-precision operations are not available, one can interpret the input as a vector of half-words (  $w/2$  -bit integers). The algorithm will then use  $\lceil k/2 \rceil$  multiplications, where  $k$  was the number of half-words in the vector. Thus, the algorithm runs at a "rate" of one multiplication per word of input.

The same scheme can also be used for hashing integers, by interpreting their bits as vectors of bytes. In this variant, the vector technique is known as **tabulation hashing** and it provides a practical alternative to multiplication-based universal hashing schemes.<sup>[12]</sup>

Strong universality at high speed is also possible.<sup>[13]</sup> Initialize the hash function with a vector  $\bar{a} = (a_0, \dots, a_k)$  of random integers on  $2w$  bits. Compute

$$h_{\bar{a}}(\bar{x})^{\text{strong}} = (a_0 + \sum_{i=0}^{k-1} a_{i+1} x_i \bmod 2^{2w}) \div 2^w$$

The result is strongly universal on  $w$  bits. Experimentally, it was found to run at 0.2 CPU cycle per byte on recent Intel processors for  $w = 32$  .

### Hashing strings

This refers to hashing a *variable-sized* vector of machine words. If the length of the string can be bounded by a small number, it is best to use the vector solution from above (conceptually padding the vector with zeros up to the upper bound). The space required is the maximal length of the string, but the time to evaluate  $h(s)$  is just the length of  $s$  . As long as zeroes are forbidden in the string, the zero-padding can be ignored when evaluating the hash function without affecting universality<sup>[9]</sup>). Note that if zeroes are allowed in the string, then it might be best to append a fictitious non-zero (e.g., 1) character to

all strings prior to padding: this will ensure that universality is not affected.<sup>[13]</sup>

Now assume we want to hash  $\bar{x} = (x_0, \dots, x_\ell)$ , where a good bound on  $\ell$  is not known a priori. A universal family proposed by<sup>[10]</sup> treats the string  $x$  as the coefficients of a polynomial modulo a large prime. If  $x_i \in [u]$ , let  $p \geq \max\{u, m\}$  be a prime and define:

$$h_a(\bar{x}) = h_{\text{int}}\left(\left(\sum_{i=0}^{\ell} x_i \cdot a^i\right) \bmod p\right),$$

where  $a \in [p]$  is uniformly random and  $h_{\text{int}}$  is chosen randomly from a universal family mapping integer domain  $[p] \mapsto [m]$ .

Using properties of modular arithmetic, above can be computed without producing large numbers for large strings as follows:<sup>[14]</sup>

```
int hash(String x, int a, int p) int h=x[0] for (int i=1 ; i < x.length ; i++) h = ((h*a) + x[i]) mod p return h
```

Consider two strings  $\bar{x}, \bar{y}$  and let  $\ell$  be length of the longer one; for the analysis, the shorter string is conceptually padded with zeros up to length  $\ell$ . A collision before applying  $h_{\text{int}}$  implies that  $a$  is a root of the polynomial with coefficients  $\bar{x} - \bar{y}$ . This polynomial has at most  $\ell$  roots modulo  $p$ , so the collision probability is at most  $\ell/p$ . The probability of collision through the random  $h_{\text{int}}$  brings the total collision probability to  $\frac{1}{m} + \frac{\ell}{p}$ . Thus, if the prime  $p$  is sufficiently large compared to the length of strings hashed, the family is very close to universal (in statistical distance).

To mitigate the computational penalty of modular arithmetic, two tricks are used in practice:<sup>[19]</sup>

1. One chooses the prime  $p$  to be close to a power of two, such as a **Mersenne prime**. This allows arithmetic modulo  $p$  to be implemented without division (using faster operations like addition and shifts). For instance, on modern architectures one can work with  $p = 2^{61} - 1$ , while  $x_i$ 's are 32-bit values.
2. One can apply vector hashing to blocks. For instance, one applies vector hashing to each 16-word block of the string, and applies string hashing to the  $\lceil k/16 \rceil$  results. Since the slower string hashing is applied on a substantially smaller vector, this will essentially be as fast as vector hashing.

#### 11.11.4 See also

- K-independent hashing
- Rolling hashing
- Tabulation hashing
- Min-wise independence

- Universal one-way hash function
- Low-discrepancy sequence
- Perfect hashing

#### 11.11.5 References

- [1] Carter, Larry; Wegman, Mark N. (1979). “Universal Classes of Hash Functions”. *Journal of Computer and System Sciences* **18** (2): 143–154. doi:10.1016/0022-0000(79)90044-8. Conference version in STOC'77.
- [2] Miltersen, Peter Bro. “Universal Hashing”. Archived from the original (PDF) on 24 June 2009.
- [3] Motwani, Rajeev; Raghavan, Prabhakar (1995). *Randomized Algorithms*. Cambridge University Press. p. 221. ISBN 0-521-47465-5.
- [4] Baran, Ilya; Demaine, Erik D.; Pătrașcu, Mihai (2008). “Subquadratic Algorithms for 3SUM” (PDF). *Algorithmica* **50** (4): 584–596. doi:10.1007/s00453-007-9036-3.
- [5] Dietzfelbinger, Martin; Hagerup, Torben; Katajainen, Jyrki; Penttonen, Martti (1997). “A Reliable Randomized Algorithm for the Closest-Pair Problem” (POSTSCRIPT). *Journal of Algorithms* **25** (1): 19–51. doi:10.1006/jagm.1997.0873. Retrieved 10 February 2011.
- [6] Thorup, Mikkel. “Text-book algorithms at SODA”.
- [7] Woelfel, Philipp (2003). *Über die Komplexität der Multiplikation in eingeschränkten Branchingprogrammmodellen* (PDF) (Ph.D.). Universität Dortmund. Retrieved 18 September 2012.
- [8] Woelfel, Philipp (1999). *Efficient Strongly Universal and Optimally Universal Hashing* (PDF). Mathematical Foundations of Computer Science 1999. LNCS **1672**. pp. 262–272. doi:10.1007/3-540-48340-3\_24. Retrieved 17 May 2011.
- [9] Thorup, Mikkel (2009). *String hashing for linear probing* (PDF). Proc. 20th ACM-SIAM Symposium on Discrete Algorithms (SODA): 655–664., section 5.3
- [10] Dietzfelbinger, Martin; Gil, Joseph; Matias, Yossi; Pippenger, Nicholas (1992). *Polynomial Hash Functions Are Reliable (Extended Abstract)*. Proc. 19th International Colloquium on Automata, Languages and Programming (ICALP): 235–246.
- [11] Black, J.; Halevi, S.; Krawczyk, H.; Krovetz, T. (1999). *UMAC: Fast and Secure Message Authentication* (PDF). *Advances in Cryptology (CRYPTO '99)*, Equation 1
- [12] Pătrașcu, Mihai; Thorup, Mikkel (2011). *The power of simple tabulation hashing*. *Proceedings of the 43rd annual ACM Symposium on Theory of Computing (STOC '11)*: 1–10. arXiv:1011.5200. doi:10.1145/1993636.1993638.
- [13] Kaser, Owen; Lemire, Daniel (2013). “Strongly universal string hashing is fast”. *Computer Journal* (Oxford University Press). arXiv:1202.4961. doi:10.1093/comjnl/bxt070.
- [14] “Hebrew University Course Slides” (PDF).

### 11.11.6 Further reading

- Knuth, Donald Ervin (1998). *The Art of Computer Programming, Vol. III: Sorting and Searching* (3rd ed.). Reading, Mass; London: Addison-Wesley. ISBN 0-201-89685-0.

### 11.11.7 External links

- Open Data Structures - Section 5.1.1 - Multiplicative Hashing

## 11.12 Linear hashing

**Linear hashing** is a dynamic hash table algorithm invented by Witold Litwin (1980),<sup>[1]</sup> and later popularized by Paul Larson. Linear hashing allows for the expansion of the hash table one slot at a time. The frequent single slot expansion can very effectively control the length of the collision chain. The cost of hash table expansion is spread out across each hash table insertion operation, as opposed to being incurred all at once.<sup>[2]</sup> Linear hashing is therefore well suited for interactive applications.

### 11.12.1 Algorithm Details

First the initial hash table is set up with some arbitrary initial number of buckets. The following values need to be kept track of:

- $N$  : The initial number of buckets.
- $L$  : The current level which is an integer that indicates on a logarithmic scale approximately how many buckets the table has grown by. This is initially 0 .
- $S$  : The step pointer which points to a bucket. It initially points to the first bucket in the table.

Bucket collisions can be handled in a variety of ways but it is typical to have space for two items in each bucket and to add more buckets whenever a bucket overflows. More than two items can be used once the implementation is debugged. Addresses are calculated in the following way:

- Apply a hash function to the key and call the result  $H$  .
- If  $H \bmod N \times 2^L$  is an address that comes before  $S$  , the address is  $H \bmod N \times 2^{L+1}$  .
- If  $H \bmod N \times 2^L$  is  $S$  or an address that comes after  $S$  , the address is  $H \bmod N \times 2^L$  .

To add a bucket:

- Allocate a new bucket at the end of the table.
- If  $S$  points to the  $N \times 2^L$  th bucket in the table, reset  $S$  and increment  $L$  .
- Otherwise increment  $S$  .

The effect of all of this is that the table is split into three sections; the section before  $S$  , the section from  $S$  to  $N \times 2^L$  , and the section after  $N \times 2^L$  . The first and last sections are stored using  $H \bmod N \times 2^{L+1}$  and the middle section is stored using  $H \bmod N \times 2^L$  . Each time  $S$  reaches  $N \times 2^L$  the table has doubled in size.

### Points to ponder over

- Full buckets are not necessarily split, and an overflow space for temporary overflow buckets is required. In external storage, this could mean a second file.
- Buckets split are not necessarily full
- Every bucket will be split sooner or later and so all Overflows will be reclaimed and rehashed.
- Split pointer  $s$  decides which bucket to split
  - $s$  is independent to overflowing bucket
  - At level  $i$ ,  $s$  is between 0 and  $2^i$
  - $s$  is incremented and if at end, is reset to 0.
  - since a bucket at  $s$  is split then  $s$  is incremented, only buckets before  $s$  have the second doubled hash space .
  - a large good pseudo random number is first obtained , and then is bit-masked with either  $(2^i) - 1$  or  $(2^{i+1}) - 1$ , but the latter only applies if  $x$ , the random number, bit-masked with the former ,  $(2^i) - 1$ , is less than  $S$ , so the larger range of hash values only apply to buckets that have already been split.
  - e.g. To bit-mask a number , use  $x \& 0111$  , where  $\&$  is the AND operator, 111 is binary 7 , where 7 = 8 - 1 and 8 is  $2^3$  and  $i = 3$ .
  - what if  $s$  lands on a bucket which has 1 or more full overflow buckets ? The split will only reduce the overflow bucket count by 1, and the remaining overflow buckets will have to be recreated by seeing which of the new 2 buckets, or their overflow buckets, the overflow entries belong.
  - $h_i(k) = h(k) \bmod (2^i n)$
  - $h_{i+1}$  doubles the range of  $h_i$

### Algorithm for inserting ‘k’ and checking overflow condition

- $b = h_0(k)$
- if  $b <$  split-pointer then
- $b = h_1(k)$

### Searching in the hash table for ‘k’

- $b = h_0(k)$
- if  $b <$  split-pointer then
- $b = h_1(k)$
- read bucket  $b$  and search there

### 11.12.2 Adoption in language systems

Griswold and Townsend [3] discussed the adoption of linear hashing in the **Icon** language. They discussed the implementation alternatives of **dynamic array** algorithm used in linear hashing, and presented performance comparisons using a list of **Icon** benchmark applications.

### 11.12.3 Adoption in database systems

Linear hashing is used in the **BDB** Berkeley database system, which in turn is used by many software systems such as **OpenLDAP**, using a C implementation derived from the **CACM** article and first published on the **Usenet** in 1988 by Esmond Pitt.

### 11.12.4 References

- [1] Litwin, Witold (1980), “Linear hashing: A new tool for file and table addressing” (PDF), *Proc. 6th Conference on Very Large Databases*: 212–223
- [2] Larson, Per-Åke (April 1988), “Dynamic Hash Tables”, *Communications of the ACM* **31** (4): 446–457, doi:10.1145/42404.42410
- [3] Griswold, William G.; Townsend, Gregg M. (April 1993), “The Design and Implementation of Dynamic Hashing for Sets and Tables in Icon”, *Software - Practice and Experience* **23** (4): 351–367

### 11.12.5 External links

- [Sorted Linear Hash Table, C++ implementation of a Linear Hashtable](#)
- [TommyDS, C implementation of a Linear Hashtable](#)
- [An in Memory Go Implementation with Explanation](#)

- Black, Paul E. “linear hashing”. *Dictionary of Algorithms and Data Structures*. NIST.

- [A C++ Implementation of Linear Hashtable which Supports Both Filesystem and In-Memory Storage](#)

### 11.12.6 See also

- [Extendible hashing](#)
- [Consistent hashing](#)

## 11.13 Extendible hashing

**Extendible hashing** is a type of hash system which treats a hash as a bit string, and uses a trie for bucket lookup.<sup>[1]</sup> Because of the hierarchical nature of the system, rehashing is an incremental operation (done one bucket at a time, as needed). This means that time-sensitive applications are less affected by table growth than by standard full-table rehashes.

### 11.13.1 Example

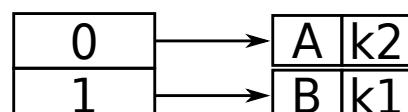
This is an example from Fagin et al. (1979).

Assume that the hash function  $h(k)$  returns a binary number. The first  $i$  bits of each string will be used as indices to figure out where they will go in the “directory” (hash table). Additionally,  $i$  is the smallest number such that the first  $i$  bits of all keys are different.

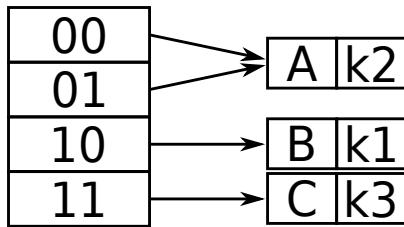
Keys to be used:

$$\begin{aligned} h(k_1) &= 100100 \\ h(k_2) &= 010110 \\ h(k_3) &= 110110 \end{aligned}$$

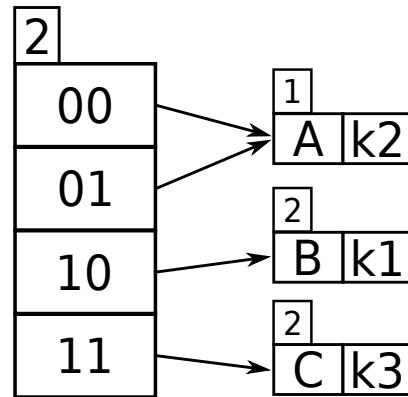
Let’s assume that for this particular example, the bucket size is 1. The first two keys to be inserted,  $k_1$  and  $k_2$ , can be distinguished by the most significant bit, and would be inserted into the table as follows:



Now, if  $k_3$  were to be hashed to the table, it wouldn’t be enough to distinguish all three keys by one bit (because  $k_3$  and  $k_1$  have 1 as their leftmost bit). Also, because the bucket size is one, the table would overflow. Because comparing the first two most significant bits would give each key a unique location, the directory size is doubled as follows:



And so now  $k_1$  and  $k_3$  have a unique location, being distinguished by the first two leftmost bits. Because  $k_2$  is in the top half of the table, both 00 and 01 point to it because there is no other key to compare to that begins with a 0.



### Further detail

$$h(k_4) = 011110$$

Now,  $k_4$  needs to be inserted, and it has the first two bits as 01..(1110), and using a 2 bit depth in the directory, this maps from 01 to Bucket A. Bucket A is full (max size 1), so it must be split; because there is more than one pointer to Bucket A, there is no need to increase the directory size.

What is needed is information about:

1. The key size that maps the directory (the global depth), and
2. The key size that has previously mapped the bucket (the local depth)

In order to distinguish the two action cases:

1. Doubling the directory when a bucket becomes full
2. Creating a new bucket, and re-distributing the entries between the old and the new bucket

Examining the initial case of an extendible hash structure, if each directory entry points to one bucket, then the local depth should be equal to the global depth.

The number of directory entries is equal to  $2^{\text{global depth}}$ , and the initial number of buckets is equal to  $2^{\text{local depth}}$ .

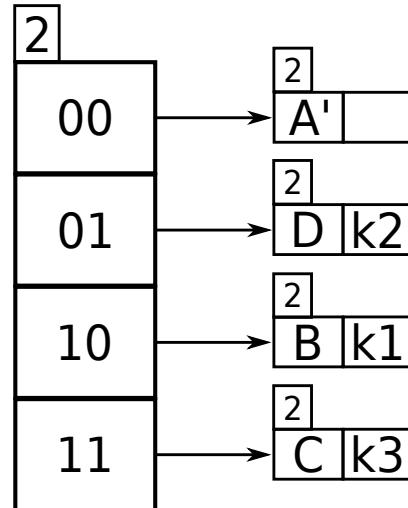
Thus if global depth = local depth = 0, then  $2^0 = 1$ , so an initial directory of one pointer to one bucket.

Back to the two action cases:

If the local depth is equal to the global depth, then there is only one pointer to the bucket, and there is no other directory pointers that can map to the bucket, so the directory must be doubled (case 1).

If the bucket is full, if the local depth is less than the global depth, then there exists more than one pointer from the directory to the bucket, and the bucket can be split (case 2).

Key 01 points to Bucket A, and Bucket A's local depth of 1 is less than the directory's global depth of 2, which means keys hashed to Bucket A have only used a 1 bit prefix (i.e. 0), and the bucket needs to have its contents split using keys 1 + 1 = 2 bits in length; in general, for any local depth  $d$  where  $d$  is less than  $D$ , the global depth, then  $d$  must be incremented after a bucket split, and the new  $d$  used as the number of bits of each entry's key to redistribute the entries of the former bucket into the new buckets.



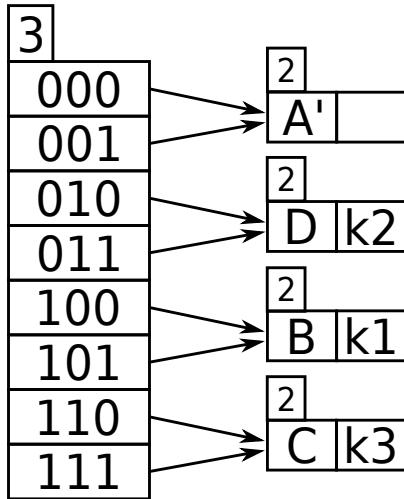
$$\text{Now, } h(k_4) = 011110$$

is tried again, with 2 bits 01.., and now key 01 points to a new bucket but there is still  $k_2$  in it ( $h(k_2) = 010110$  and also begins with 01).

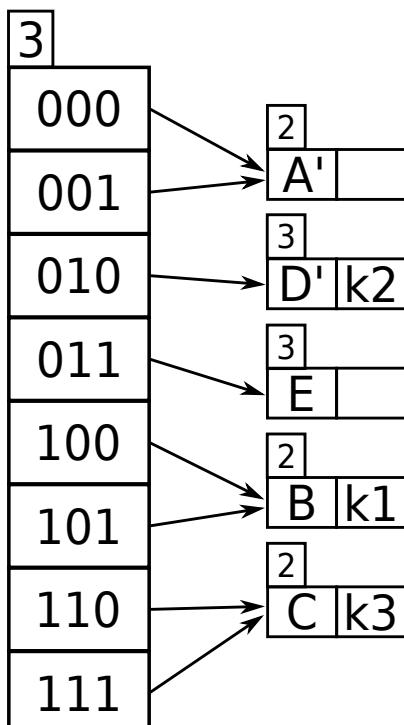
If  $k_2$  had been 000110, with key 00, there would have been no problem, because  $k_2$  would have remained in the new bucket  $A'$  and bucket  $D$  would have been empty.

(This would have been the most likely case by far when buckets are of greater size than 1 and the newly split buckets would be exceedingly unlikely to overflow, unless all the entries were all rehashed to one bucket again. But just to emphasize the role of the depth information, the example will be pursued logically to the end.)

So Bucket D needs to be split, but a check of its local depth, which is 2, is the same as the global depth, which is 2, so the directory must be split again, in order to hold keys of sufficient detail, e.g. 3 bits.



1. Bucket D needs to split due to being full.
2. As D's local depth = the global depth, the directory must double to increase bit detail of keys.
3. Global depth has incremented after directory split to 3.
4. The new entry k4 is rekeyed with global depth 3 bits and ends up in D which has local depth 2, which can now be incremented to 3 and D can be split to D' and E.
5. The contents of the split bucket D, k2, has been rekeyed with 3 bits, and it ends up in D.
6. K4 is retried and it ends up in E which has a spare slot.



Now,  $h(k2) = 010110$  is in D and  $h(k4) = 011110$  is tried again, with 3 bits 011.., and it points to bucket D which already contains k2 so is full; D's local depth is 2 but now the global depth is 3 after the directory doubling, so now D can be split into bucket's D' and E, the contents of D, k2 has its  $h(k2)$  retried with a new global depth bitmask of 3 and k2 ends up in D', then the new entry k4 is retried with  $h(k4)$  bitmasked using the new global depth bit count of 3 and this gives 011 which now points to a new bucket E which is empty. So K4 goes in Bucket E.

### 11.13.2 Example implementation

Below is the extendible hashing algorithm in Python, with the disc block / memory page association, caching and consistency issues removed. Note a problem exists if the depth exceeds the bit size of an integer, because then doubling of the directory or splitting of a bucket won't allow entries to be rehashed to different buckets.

The code uses the *least significant bits*, which makes it more efficient to expand the table, as the entire directory can be copied as one block (Ramakrishnan & Gehrke (2003)).

#### Python example

```
PAGE_SZ = 20
class Page(object):
 def __init__(self):
 self.m = {}
 def full(self):
 return len(self.m) >= PAGE_SZ
 def put(self, k, v):
 self.m[k] = v
 def get(self, k):
 return self.m.get(k)
class EH(object):
 def __init__(self):
 self.gd = 0
 self.p = Page()
 self.pp = [self.p]
 def get_page(self, k):
 h = hash(k)
 p = self.pp[h & ((1 << self.gd) - 1)]
 return p
 def put(self, k, v):
 p = self.get_page(k)
 if p.full():
 p.d = self.gd
 self.pp *= 2
 self.gd += 1
 if p.d < self.gd:
 p.put(k, v)
 else:
 p1 = Page()
 p2 = Page()
 for k2, v2 in p.m.items():
 h = hash(k2)
 h = h & ((1 << self.gd) - 1)
 if (h >> p.d) & 1 == 1:
 p2.put(k2, v2)
 else:
 p1.put(k2, v2)
 p.m = p1.m
 p.d = p1.d
 p1 = p2
 p2.d = p1.d + 1
 p2.put(k, v)
 def get(self, k):
 p = self.get_page(k)
 return p.get(k)
if __name__ == "__main__":
 eh = EH()
 N = 10088
 l = list(range(N))
 random.shuffle(l)
 for x in l:
 eh.put(x, x)
 print 1 for i in range(N)
 print eh.get(i)
```

### 11.13.3 Notes

- [1] Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R. (September 1979), "Extendible Hashing - A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems* 4 (3): 315–344, doi:10.1145/320083.320092

### 11.13.4 See also

- Trie
- Hash table
- Stable hashing
- Consistent hashing
- Linear hashing

### 11.13.5 References

- Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R. (September 1979), “Extendible Hashing - A Fast Access Method for Dynamic Files”, *ACM Transactions on Database Systems* **4** (3): 315–344, doi:10.1145/320083.320092
- Ramakrishnan, R.; Gehrke, J. (2003), *Database Management Systems, 3rd Edition: Chapter 11, Hash-Based Indexing*, pp. 373–378

### 11.13.6 External links

- Black, Paul E. “Extendible hashing”. *Dictionary of Algorithms and Data Structures*. NIST.
- Extendible Hashing at University of Nebraska
- Extendible Hashing notes at Arkansas State University
- Extendible hashing notes

## 11.14 2-choice hashing

**2-choice hashing**, also known as **2-choice chaining**, is a variant of a **hash table** in which keys are added by hashing with two hash functions. The key is put in the array position with the fewer (colliding) keys. Some collision resolution scheme is needed, unless keys are kept in buckets. The average-case cost of a successful search is  $O(2 + (m-1)/n)$ , where  $m$  is the number of keys and  $n$  is the size of the array. The most collisions is  $\log_2 \ln n + \theta(m/n)$  with high probability.”<sup>[1]</sup>

### 11.14.1 How It Works

2-choice hashing utilizes two hash functions  $h_1(x)$  and  $h_2(x)$  which work as hash functions are expected to work (i.e. mapping integers from the universe into a specified range). The two hash functions should be independent and have no correlation to each other. Having two hash functions allows any integer  $x$  to have up to two potential locations to be stored based on the values of the respective outputs,  $h_1(x)$  and  $h_2(x)$ . It is important to note that,

although there are two hash functions, there is only one table; both hash functions map to locations on that table.

### 11.14.2 Implementation

The most important functions of the hashing implementation in this case are insertion and search.

**Insertion:** When inserting the values of both hash functions are computed for the to-be-inserted object. The object is then placed in the bucket which contains fewer objects. If the buckets are equal in size, the default location is the  $h_1(x)$  value.

**Search:** Effective searches are done by looking in both buckets, that is, the bucket locations which  $h_1(x)$  and  $h_2(x)$  mapped to for the desire value.

### 11.14.3 Performance

As is true with all hash tables, the performance is based on the largest bucket. Although there are instances where bucket sizes happen to be large based on the values and the hash functions used, this is rare. Having two hash functions and, therefore, two possible locations for any one value, makes the possibility of large buckets even more unlikely to happen.

The expected bucket size while using 2-choice hashing is:  $\theta(\log(\log(n)))$ . This improvement is due to the randomized concept known as **The Power of Two Choices**.

Using two hash functions offers substantial benefits over a single hash function. There is little improvement (and no change to the expected order statistics) if more than two hash functions are used: “Additional hash functions only decrease the maximum by a constant factor.”<sup>[2]</sup>

### 11.14.4 See also

- 2-left hashing

### 11.14.5 References

[1] Paul E. Black, NIST, DADS

[2] Paul E. Black, DADS, retrieved 29 January 2015.

Black, Paul E. “2-choice hashing”. *Dictionary of Algorithms and Data Structures*. NIST.

### 11.14.6 Further reading

- Azar, Yossi; Broder, Andrei Z.; Karlin, Anna R.; Upfal, Eli (1999), “Balanced Allocations”, *SIAM J. Comput.* **29** (1): 180–200

## 11.15 Pearson hashing

**Pearson hashing**<sup>[1][2]</sup> is a hash function designed for fast execution on processors with 8-bit registers. Given an input consisting of any number of bytes, it produces as output a single byte that is strongly dependent<sup>[1]</sup> on every byte of the input. Its implementation requires only a few instructions, plus a 256-byte **lookup table** containing a permutation of the values 0 through 255.

This hash function is a **CBC-MAC** that uses an 8-bit **substitution cipher** implemented via the substitution table. An 8-bit cipher has negligible cryptographic security, so the Pearson hash function is not **cryptographically strong**; but it offers these benefits:

- It is extremely simple.
- It executes quickly on resource-limited processors.
- There is no simple class of inputs for which **collisions** (identical outputs) are especially likely.
- Given a small, privileged set of inputs (e.g., **reserved words** for a **compiler**), the permutation table can be adjusted so that those inputs yield distinct hash values, producing what is called a **perfect hash function**.

One of its drawbacks when compared with other hashing algorithms designed for 8-bit processors is the suggested 256 byte lookup table, which can be prohibitively large for a small **microcontroller** with a program memory size on the order of hundreds of bytes. A workaround to this is to use a simple permutation function instead of a table stored in program memory. However, using a too simple function, such as  $T[i] = 255 - i$  partly defeats the usability as a hash function as **anagrams** will result in the same hash value; using a too complex function, on the other hand, will affect speed negatively.

The algorithm can be described by the following **pseudocode**, which computes the hash of message  $C$  using the permutation table  $T$ :

```
h := 0
for each c in C
 loop index := h xor c
 h := T[index]
end loop
return h
```

### 11.15.1 C implementation to generate 64-bit (16 hex chars) hash

1. void Pearson16(const unsigned char \*x, size\_t len,  
char \*hex, size\_t hexlen) {
2. size\_t i, j;
3. unsigned char hh[8];
4. static const unsigned char T[256] = {
5. // 256 values 0-255 in any (random) order suffices

6. 98, 6, 85, 150, 36, 23, 112, 164, 135, 207, 169, 5, 26,  
64, 165, 219, // 1
7. 61, 20, 68, 89, 130, 63, 52, 102, 24, 229, 132, 245,  
80, 216, 195, 115, // 2
8. 90, 168, 156, 203, 177, 120, 2, 190, 188,  
7, 100, 185, 174, 243, 162, 10, // 3
9. 237, 18, 253, 225, 8, 208, 172, 244, 255, 126, 101,  
79, 145, 235, 228, 121, // 4
10. 123, 251, 67, 250, 161, 0, 107, 97, 241, 111, 181,  
82, 249, 33, 69, 55, // 5
11. 59, 153, 29, 9, 213, 167, 84, 93, 30, 46, 94,  
75, 151, 114, 73, 222, // 6
12. 197, 96, 210, 45, 16, 227, 248, 202, 51, 152, 252, 125,  
81, 206, 215, 186, // 7
13. 39, 158, 178, 187, 131, 136, 1, 49, 50, 17, 141, 91,  
47, 129, 60, 99, // 8
14. 154, 35, 86, 171, 105, 34, 38, 200, 147, 58,  
77, 118, 173, 246, 76, 254, // 9
15. 133, 232, 196, 144, 198, 124, 53, 4, 108,  
74, 223, 234, 134, 230, 157, 139, // 10
16. 189, 205, 199, 128, 176, 19, 211, 236, 127, 192, 231,  
70, 233, 88, 146, 44, // 11
17. 183, 201, 22, 83, 13, 214, 116, 109, 159, 32,  
95, 226, 140, 220, 57, 12, // 12
18. 221, 31, 209, 182, 143, 92, 149, 184, 148, 62, 113, 65,  
37, 27, 106, 166, // 13
19. 3, 14, 204, 72, 21, 41, 56, 66, 28, 193, 40, 217, 25,  
54, 179, 117, // 14
20. 238, 87, 240, 155, 180, 170, 242, 212, 191, 163,  
78, 218, 137, 194, 175, 110, // 15
21. 43, 119, 224, 71, 122, 142, 42, 160, 104, 48, 247, 103,  
15, 11, 138, 239 // 16
22. };
- 23.
24. for (j = 0; j < 8; j++) {
25. unsigned char h = T[(x[0] + j) % 256];
26. for (i = 1; i < len; i++) {
27. h = T[h ^ x[i]];
28. }
29. hh[j] = h;
30. }
- 31.

```

32. snprintf(hex, hexlen, "%02X%02X%02X%02X%02X%02X%02X%02X")
33. hh[0], hh[1], hh[2], hh[3],
34. hh[4], hh[5], hh[6], hh[7];
35. }

```

For a given string or chunk of data, Pearson's original algorithm produces only an 8 bit byte or integer, 0-255. But the algorithm makes it extremely easy to generate whatever length of hash is desired. The scheme used above is a very straightforward implementation of the algorithm. As Pearson noted: a change to any bit in the string causes his algorithm to create a completely different hash (0-255). In the code above, following every completion of the inner loop, the first byte of the string is incremented by one.

Every time that simple change to the first byte of the data is made, a different Pearson hash,  $h$ , is generated. xPear16 builds a 16 hex character hash by concatenating a series of 8-bit Pearson ( $h$ ) hashes. Instead of producing a value from 0 to 255, it generates a value from 0 to 18,446,744,073,709,551,615.

Pearson's algorithm can be made to generate hashes of any desired length, simply by adding 1 to the first byte of the string, re-computing  $h$  for the string, and concatenating the results. Thus the same core logic can be made to generate 32-bit or 128-bit hashes.

## 11.15.2 References

- [1] Pearson, Peter K. (June 1990), “Fast Hashing of Variable-Length Text Strings”, *Communications of the ACM* **33** (6): 677, doi:10.1145/78973.78978
- [2] Online PDF file of the CACM paper.

## 11.16 Fowler–Noll–Vo hash function

**Fowler–Noll–Vo** is a non-cryptographic hash function created by Glenn Fowler, Landon Curt Noll, and Phong Vo.

The basis of the FNV hash algorithm was taken from an idea sent as reviewer comments to the **IEEE POSIX P1003.2** committee by Glenn Fowler and Phong Vo in 1991. In a subsequent ballot round, Landon Curt Noll improved on their algorithm. In an email message to Landon, they named it the *Fowler/Noll/Vo* or FNV hash.<sup>[1]</sup>

### 11.16.1 Overview

The current versions are FNV-1 and FNV-1a, which supply a means of creating non-zero **FNV offset basis**. FNV

currently comes in 32-, 64-, 128-, 256-, 512-, and 1024-bit flavors. For pure FNV implementations, this is determined solely by the availability of **FNV primes** for the desired bit length; however, the FNV webpage discusses methods of adapting one of the above versions to a smaller length that may or may not be a power of two.<sup>[2][3]</sup>

The FNV hash algorithms and **sample FNV source code** have been released into the public domain.<sup>[4]</sup>

FNV is not a **cryptographic hash**.

### 11.16.2 The hash

One of FNV's key advantages is that it is very simple to implement. Start with an initial hash value of **FNV offset basis**. For each byte in the input, **multiply** **hash** by the **FNV prime**, then **XOR** it with the byte from the input. The alternate algorithm, FNV-1a, reverses the multiply and XOR steps.

#### FNV-1 hash

The FNV-1 hash algorithm is as follows:<sup>[5]</sup>

*hash* = **FNV\_offset\_basis** for each *octet\_of\_data* to be hashed *hash* = *hash* × **FNV\_prime** *hash* = *hash* XOR *octet\_of\_data* return *hash*

In the above pseudocode, all variables are **unsigned integers**. All variables, except for *octet\_of\_data*, have the same number of **bits** as the FNV hash. The variable, *octet\_of\_data*, is an 8 bit unsigned **integer**.

As an example, consider the 64-bit FNV-1 hash:

- All variables, except for *octet\_of\_data*, are 64-bit **unsigned integers**.
- The variable, *octet\_of\_data*, is an 8 bit **unsigned integer**.
- The **FNV\_offset\_basis** is the 64-bit **FNV offset basis** value: 14695981039346656037 (in hex, 0xcbf29ce484222325).
- The **FNV\_prime** is the 64-bit **FNV prime** value: 1099511628211 (in hex, 0x100000001b3).
- The **multiply** returns the lower 64-bits of the product.
- The **XOR** is an 8-bit operation that modifies only the lower 8-bits of the hash value.
- The *hash* value returned is a 64-bit **unsigned integer**.

The values for **FNV prime** and **FNV offset basis** may be found in this table.<sup>[6]</sup>

### FNV-1a hash

The FNV-1a hash differs from the FNV-1 hash by only the order in which the **multiply** and **XOR** is performed:<sup>[7]</sup>

*hash = FNV\_offset\_basis* for each *octet\_of\_data* to be hashed  
*hash = hash XOR octet\_of\_data* *hash = hash × FNV\_prime* return *hash*

The above **pseudocode** has the same assumptions that were noted for the FNV-1 pseudocode. The small change in order leads to much better avalanche characteristics.<sup>[8]</sup>

### 11.16.3 Non-cryptographic hash

The FNV hash was designed for fast **hash table** and **checksum** use, not **cryptography**. The authors have identified the following properties as making the algorithm unsuitable as a cryptographic hash function:<sup>[9]</sup>

- **Speed of Computation** - As a hash designed primarily for hashtable and checksum use, FNV-1 and 1a were designed to be fast to compute. However, this same speed makes finding specific hash values (collisions) by brute force faster.
- **Sticky State** - Being an iterative hash based primarily on multiplication and XOR, the algorithm is sensitive to the number zero. Specifically, if the hash value were to become zero at any point during calculation, and the next byte hashed were also all zeroes, the hash would not change. This makes colliding messages trivial to create given a message that results in a hash value of zero at some point in its calculation. Additional operations, such as the addition of a third constant prime on each step, can mitigate this but may have detrimental effects on **avalanche effect** or random distribution of hash values.
- **Diffusion** - The ideal secure hash function is one in which each byte of input has an equally-complex effect on every bit of the hash. In the FNV hash, the ones place (the rightmost bit) is always the XOR of the rightmost bit of every input byte. This can be mitigated by XOR-folding (computing a hash twice the desired length, and then XORing the bits in the “upper half” with the bits in the “lower half”).

### 11.16.4 See also

- **Pearson hashing** (uses a constant linear permutation instead of a constant prime seed)
- **Jenkins hash function**
- **MurmurHash**

### 11.16.5 Notes

- [1] FNV hash history
- [2] Changing the FNV hash size - xor-folding
- [3] Changing the FNV hash size - non-powers of 2
- [4] FNV put into the public domain
- [5] The core of the FNV hash
- [6] Parameters of the FNV-1 hash
- [7] FNV-1a alternate algorithm
- [8] Avalanche
- [9] The FNV Non-Cryptographic Hash Algorithm - Why is FNV Non-Cryptographic?

### 11.16.6 External links

- Landon Curt Noll’s webpage on **FNV** (with table of base & prime parameters)
- Internet draft by Fowler, Noll, Vo, and Eastlake (2011, work in progress)
- FNV-1 and FNV-1a Calculator in-browser (all bit lengths supported for verification)

## 11.17 Bitstate hashing

**Bitstate hashing** is a **hashing** method invented in 1968 by Morris.<sup>[1]</sup> It is used for state hashing, where each state (e.g. of an automaton) is represented by a number and it is passed to some **hash function**.

The result of the function is then taken as the index to an array of bits (a **bit-field**), where one looks for 1 if the state was already seen before or stores 1 by itself if not.

It usually serves as a yes–no technique without a need of storing whole state bit representation.

A shortcoming of this framework is losing precision like in other hashing techniques. Hence some tools use this technique with more than one hash function so that the bit-field gets widened by the number of used functions, each having its own row. And even after all functions return values (the indices) point to fields with contents equal to 1, the state may be uttered as visited with much higher probability.

### 11.17.1 Use

- It is utilized in **SPIN** model checker for decision whether a state was already visited by nested-**depth-first search** searching algorithm or not. They mention savings of 98% of memory in the case of using one hash function (175 MB to 3 MB) and 92% when

two hash functions are used (13 MB). The state coverage dropped to 97% in the former case. [2]

## 11.17.2 References

- [1] Morris, R. (1968). *Scatter Storage Techniques*
- [2] Holzmann, G. J. (2003) Addison Wesley. *Spin Model Checker, The: Primer and Reference Manual*

## 11.18 Bloom filter

Not to be confused with Bloom shader effect.

A **Bloom filter** is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an **element** is a member of a **set**. **False positive** matches are possible, but **false negatives** are not, thus a Bloom filter has a 100% **recall rate**. In other words, a query returns either “possibly in set” or “definitely not in set”. Elements can be added to the set, but not removed (though this can be addressed with a “counting” filter). The more elements that are added to the set, the larger the probability of false positives.

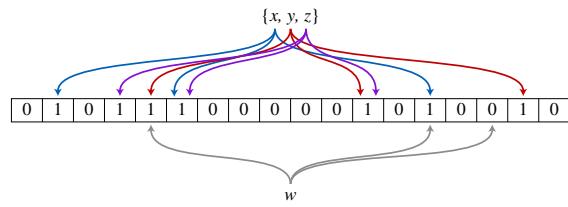
Bloom proposed the technique for applications where the amount of source data would require an impractically large hash area in memory if “conventional” error-free hashing techniques were applied. He gave the example of a **hyphenation algorithm** for a dictionary of 500,000 words, out of which 90% follow simple hyphenation rules, but the remaining 10% require expensive disk accesses to retrieve specific hyphenation patterns. With sufficient core memory, an error-free hash could be used to eliminate all unnecessary disk accesses; on the other hand, with limited core memory, Bloom’s technique uses a smaller hash area but still eliminates most unnecessary accesses. For example, a hash area only 15% of the size needed by an ideal error-free hash still eliminates 85% of the disk accesses (Bloom (1970)).

More generally, fewer than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set (Bonomi et al. (2006)).

### 11.18.1 Algorithm description

An **empty Bloom filter** is a **bit array** of  $m$  bits, all set to 0. There must also be  $k$  different **hash functions** defined, each of which **maps** or hashes some set element to one of the  $m$  array positions with a uniform random distribution.

To **add** an element, feed it to each of the  $k$  hash functions to get  $k$  array positions. Set the bits at all these positions to 1.



An example of a Bloom filter, representing the set  $\{x, y, z\}$ . The colored arrows show the positions in the bit array that each set element is mapped to. The element  $w$  is not in the set  $\{x, y, z\}$ , because it hashes to one bit-array position containing 0. For this figure,  $m = 18$  and  $k = 3$ .

To **query** for an element (test whether it is in the set), feed it to each of the  $k$  hash functions to get  $k$  array positions. If any of the bits at these positions is 0, the element is definitely not in the set – if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a **false positive**. In a simple Bloom filter, there is no way to distinguish between the two cases, but more advanced techniques can address this problem.

The requirement of designing  $k$  different independent hash functions can be prohibitive for large  $k$ . For a good **hash function** with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple “different” hash functions by slicing its output into multiple bit fields. Alternatively, one can pass  $k$  different initial values (such as 0, 1, ...,  $k - 1$ ) to a hash function that takes an initial value; or add (or append) these values to the key. For larger  $m$  and/or  $k$ , independence among the hash functions can be relaxed with negligible increase in false positive rate (Dillinger & Manolios (2004a), Kirsch & Mitzenmacher (2006)). Specifically, Dillinger & Manolios (2004b) show the effectiveness of deriving the  $k$  indices using enhanced double hashing or triple hashing, variants of **double hashing** that are effectively simple random number generators seeded with the two or three hash values.

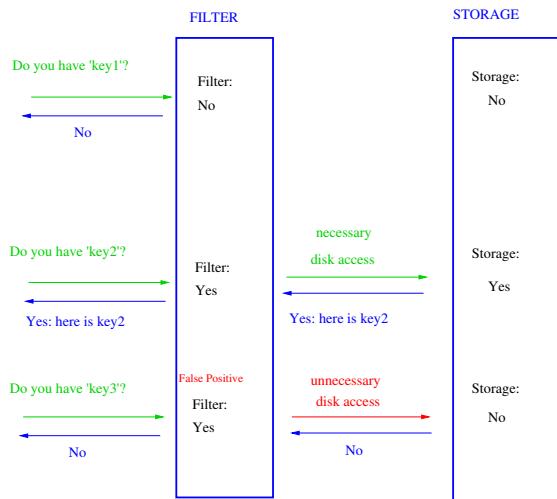
Removing an element from this simple Bloom filter is impossible because false negatives are not permitted. An element maps to  $k$  bits, and although setting any one of those  $k$  bits to zero suffices to remove the element, it also results in removing any other elements that happen to map onto that bit. Since there is no way of determining whether any other elements have been added that affect the bits for an element to be removed, clearing any of the bits would introduce the possibility for false negatives.

One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which may be undesirable. In this approach re-adding a previously removed item is not possible, as

one would have to remove it from the “removed” filter.

It is often the case that all the keys are available but are expensive to enumerate (for example, requiring many disk reads). When the false positive rate gets too high, the filter can be regenerated; this should be a relatively rare event.

### 11.18.2 Space and time advantages



*Bloom filter used to speed up answers in a key-value storage system. Values are stored on a disk which has slow access times. Bloom filter decisions are much faster. However some unnecessary disk accesses are made when the filter reports a positive (in order to weed out the false positives). Overall answer speed is better with the Bloom filter than without the Bloom filter. Use of a Bloom filter for this purpose, however, does increase memory usage.*

While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked lists of the entries. Most of these require storing at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). Linked structures incur an additional linear space overhead for pointers. A Bloom filter with 1% error and an optimal value of  $k$ , in contrast, requires only about 9.6 bits per element — regardless of the size of the elements. This advantage comes partly from its compactness, inherited from arrays, and partly from its probabilistic nature. The 1% false-positive rate can be reduced by a factor of ten by adding only about 4.8 bits per element.

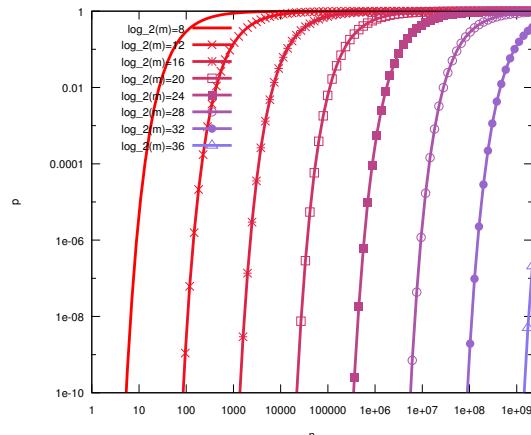
However, if the number of potential values is small and many of them can be in the set, the Bloom filter is easily surpassed by the deterministic bit array, which requires only one bit for each potential element. Note also that hash tables gain a space and time advantage if they be-

gin ignoring collisions and store only whether each bucket contains an entry; in this case, they have effectively become Bloom filters with  $k = 1$ .<sup>[1]</sup>

Bloom filters also have the unusual property that the time needed either to add items or to check whether an item is in the set is a fixed constant,  $O(k)$ , completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters. In a hardware implementation, however, the Bloom filter shines because its  $k$  lookups are independent and can be parallelized.

To understand its space efficiency, it is instructive to compare the general Bloom filter with its special case when  $k = 1$ . If  $k = 1$ , then in order to keep the false positive rate sufficiently low, a small fraction of bits should be set, which means the array must be very large and contain long runs of zeros. The information content of the array relative to its size is low. The generalized Bloom filter ( $k$  greater than 1) allows many more bits to be set while still maintaining a low false positive rate; if the parameters ( $k$  and  $m$ ) are chosen well, about half of the bits will be set,<sup>[2]</sup> and these will be apparently random, minimizing redundancy and maximizing information content.

### 11.18.3 Probability of false positives



*The false positive probability  $p$  as a function of number of elements  $n$  in the filter and the filter size  $m$ . An optimal number of hash functions  $k = (m/n) \ln 2$  has been assumed.*

Assume that a hash function selects each array position with equal probability. If  $m$  is the number of bits in the array, the probability that a certain bit is not set to 1 by a certain hash function during the insertion of an element is

$$1 - \frac{1}{m}.$$

If  $k$  is the number of hash functions, the probability that the bit is not set to 1 by any of the hash functions is

$$\left(1 - \frac{1}{m}\right)^k.$$

If we have inserted  $n$  elements, the probability that a certain bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn};$$

the probability that it is 1 is therefore

$$1 - \left(1 - \frac{1}{m}\right)^{kn}.$$

Now test membership of an element that is not in the set. Each of the  $k$  array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is often given as

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

This is not strictly correct as it assumes independence for the probabilities of each bit being set. However, assuming it is a close approximation we have that the probability of false positives decreases as  $m$  (the number of bits in the array) increases, and increases as  $n$  (the number of inserted elements) increases.

An alternative analysis arriving at the same approximation without the assumption of independence is given by Mitzenmacher and Upfal.<sup>[3]</sup> After all  $n$  items have been added to the Bloom filter, let  $q$  be the fraction of the  $m$  bits that are set to 0. (That is, the number of bits still set to 0 is  $qm$ .) Then, when testing membership of an element not in the set, for the array position given by any of the  $k$  hash functions, the probability that the bit is found set to 1 is  $1 - q$ . So the probability that all  $k$  hash functions find their bit set to 1 is  $(1 - q)^k$ . Further, the expected value of  $q$  is the probability that a given array position is left untouched by each of the  $k$  hash functions for each of the  $n$  items, which is (as above)

$$E[q] = \left(1 - \frac{1}{m}\right)^{kn}$$

It is possible to prove, without the independence assumption, that  $q$  is very strongly concentrated around its expected value. In particular, from the Azuma–Hoeffding inequality, they prove that<sup>[4]</sup>

$$\Pr(|q - E[q]| \geq \frac{\lambda}{m}) \leq 2 \exp(-2\lambda^2/m)$$

Because of this, we can say that the exact probability of false positives is

$$\sum_t \Pr(q = t)(1-t)^k \approx (1 - E[q])^k = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

as before.

### Optimal number of hash functions

For a given  $m$  and  $n$ , the value of  $k$  (the number of hash functions) that minimizes the false positive probability is

$$k = \frac{m}{n} \ln 2,$$

which gives

$$2^{-k} \approx 0.6185^{m/n}.$$

The required number of bits  $m$ , given  $n$  (the number of inserted elements) and a desired false positive probability  $p$  (and assuming the optimal value of  $k$  is used) can be computed by substituting the optimal value of  $k$  in the probability expression above:

$$p = \left(1 - e^{-(m/n \ln 2)n/m}\right)^{(m/n \ln 2)}$$

which can be simplified to:

$$\ln p = -\frac{m}{n} (\ln 2)^2.$$

This results in:

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$

This means that for a given false positive probability  $p$ , the length of a Bloom filter  $m$  is proportionate to the number of elements being filtered  $n$ .<sup>[5]</sup> While the above formula is asymptotic (i.e. applicable as  $m, n \rightarrow \infty$ ), the agreement with finite values of  $m, n$  is also quite good; the false positive probability for a finite bloom filter with  $m$  bits,  $n$  elements, and  $k$  hash functions is at most

$$\left(1 - e^{-k(n+0.5)/(m-1)}\right)^k.$$

So we can use the asymptotic formula if we pay a penalty for at most half an extra element and at most one fewer bit.<sup>[6]</sup>

### 11.18.4 Approximating the number of items in a Bloom filter

Swamidass & Baldi (2007) showed that the number of items in a Bloom filter can be approximated with the following formula,

$$n^* = -\frac{m \ln \left[ 1 - \frac{X}{m} \right]}{k}$$

where  $n^*$  is an estimate of the number of items in the filter,  $m$  is length of the filter,  $k$  is the number of hash functions per item, and  $X$  is the number of bits set to one.

### 11.18.5 The union and intersection of sets

Bloom filters are a way of compactly representing a set of items. It is common to try to compute the size of the intersection or union between two sets. Bloom filters can be used to approximate the size of the intersection and union of two sets. Swamidass & Baldi (2007) showed that for two bloom filters of length  $m$ , their counts, respectively can be estimated as

$$n(A^*) = -m \ln [1 - n(A)/m] / k$$

and

$$n(B^*) = -m \ln [1 - n(B)/m] / k$$

The size of their union can be estimated as

$$n(A^* \cup B^*) = -m \ln [1 - n(A \cup B)/m] / k$$

where  $n(A \cup B)$  is the number of bits set to one in either of the two bloom filters. Finally, the intersection can be estimated as

$$n(A^* \cap B^*) = n(A^*) + n(B^*) - n(A^* \cup B^*)$$

using the three formulas together.

### 11.18.6 Interesting properties

- Unlike a standard **hash table**, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements; adding an element never fails due to the data structure “filling up.” However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point *all* queries yield a positive result.

- Union and intersection of Bloom filters with the same size and set of hash functions can be implemented with bitwise OR and AND operations, respectively. The union operation on Bloom filters is lossless in the sense that the resulting Bloom filter is the same as the Bloom filter created from scratch using the union of the two sets. The intersect operation satisfies a weaker property: the false positive probability in the resulting Bloom filter is at most the false-positive probability in one of the constituent Bloom filters, but may be larger than the false positive probability in the Bloom filter created from scratch using the intersection of the two sets.

- Some kinds of **superimposed code** can be seen as a Bloom filter implemented with physical **edge-notched cards**. An example is **Zatocoding**, invented by Calvin Mooers in 1947, in which the set of categories associated with a piece of information is represented by notches on a card, with a random pattern of four notches for each category.

### 11.18.7 Examples

- Google **BigTable** and Apache **Cassandra** use Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.<sup>[7]</sup>
- The **Google Chrome** web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result).<sup>[8][9]</sup>
- The **Squid Web Proxy Cache** uses Bloom filters for cache digests.<sup>[10]</sup>
- **Bitcoin** uses Bloom filters to speed up wallet synchronization.<sup>[11][12]</sup>
- The **Venti** archival storage system uses Bloom filters to detect previously stored data.<sup>[13]</sup>
- The **SPIN model checker** uses Bloom filters to track the reachable state space for large verification problems.<sup>[14]</sup>
- The **Cascading** analytics framework uses Bloom filters to speed up asymmetric joins, where one of the joined data sets is significantly larger than the other (often called Bloom join<sup>[15]</sup> in the database literature).<sup>[16]</sup>
- The **Exim** mail transfer agent (MTA) uses bloom filters in its rate-limit feature.<sup>[17]</sup>

### 11.18.8 Alternatives

Classic Bloom filters use  $1.44 \log_2(1/\epsilon)$  bits of space per inserted key, where  $\epsilon$  is the false positive rate of the Bloom filter. However, the space that is strictly necessary for any data structure playing the same role as a Bloom filter is only  $\log_2(1/\epsilon)$  per key (Pagh, Pagh & Rao 2005). Hence Bloom filters use 44% more space than a hypothetical equivalent optimal data structure. The number of hash functions used to achieve a given false positive rate  $\epsilon$  is proportional to  $\log(1/\epsilon)$  which is not optimal as it has been proved that an optimal data structure would need only a constant number of hash functions independent of the false positive rate.

Stern & Dill (1996) describe a probabilistic structure based on hash tables, **hash compaction**, which Dillinger & Manolios (2004b) identify as significantly more accurate than a Bloom filter when each is configured optimally. Dillinger and Manolios, however, point out that the reasonable accuracy of any given Bloom filter over a wide range of numbers of additions makes it attractive for probabilistic enumeration of state spaces of unknown size. Hash compaction is, therefore, attractive when the number of additions can be predicted accurately; however, despite being very fast in software, hash compaction is poorly suited for hardware because of worst-case linear access time.

Putze, Sanders & Singler (2007) have studied some variants of Bloom filters that are either faster or use less space than classic Bloom filters. The basic idea of the fast variant is to locate the  $k$  hash values associated with each key into one or two blocks having the same size as processor's memory cache blocks (usually 64 bytes). This will presumably improve performance by reducing the number of potential memory cache misses. The proposed variants have however the drawback of using about 32% more space than classic Bloom filters.

The space efficient variant relies on using a single hash function that generates for each key a value in the range  $[0, n/\epsilon]$  where  $\epsilon$  is the requested false positive rate. The sequence of values is then sorted and compressed using **Golomb coding** (or some other compression technique) to occupy a space close to  $n \log_2(1/\epsilon)$  bits. To query the Bloom filter for a given key, it will suffice to check if its corresponding value is stored in the Bloom filter. Decompressing the whole Bloom filter for each query would make this variant totally unusable. To overcome this problem the sequence of values is divided into small blocks of equal size that are compressed separately. At query time only half a block will need to be decompressed on average. Because of decompression overhead, this variant may be slower than classic Bloom filters but this may be compensated by the fact that a single hash function need to be computed.

Another alternative to classic Bloom filter is the one based on space efficient variants of **cuckoo hashing**. In this case

once the hash table is constructed, the keys stored in the hash table are replaced with short signatures of the keys. Those signatures are strings of bits computed using a hash function applied on the keys.

### 11.18.9 Extensions and applications

#### Counting filters

Counting filters provide a way to implement a *delete* operation on a Bloom filter without recreating the filter afresh. In a counting filter the array positions (buckets) are extended from being a single bit to being an  $n$ -bit counter. In fact, regular Bloom filters can be considered as counting filters with a bucket size of one bit. Counting filters were introduced by Fan et al. (1998).

The insert operation is extended to *increment* the value of the buckets and the lookup operation checks that each of the required buckets is non-zero. The delete operation, obviously, then consists of decrementing the value of each of the respective buckets.

Arithmetic overflow of the buckets is a problem and the buckets should be sufficiently large to make this case rare. If it does occur then the increment and decrement operations must leave the bucket set to the maximum possible value in order to retain the properties of a Bloom filter.

The size of counters is usually 3 or 4 bits. Hence counting Bloom filters use 3 to 4 times more space than static Bloom filters. In theory, an optimal data structure equivalent to a counting Bloom filter should not use more space than a static Bloom filter.

Another issue with counting filters is limited scalability. Because the counting Bloom filter table cannot be expanded, the maximal number of keys to be stored simultaneously in the filter must be known in advance. Once the designed capacity of the table is exceeded, the false positive rate will grow rapidly as more keys are inserted.

Bonomi et al. (2006) introduced a data structure based on d-left hashing that is functionally equivalent but uses approximately half as much space as counting Bloom filters. The scalability issue does not occur in this data structure. Once the designed capacity is exceeded, the keys could be reinserted in a new hash table of double size.

The space efficient variant by Putze, Sanders & Singler (2007) could also be used to implement counting filters by supporting insertions and deletions.

Rottenstreich, Kanizo & Keslassy (2012) introduced a new general method based on variable increments that significantly improves the false positive probability of counting Bloom filters and their variants, while still supporting deletions. Unlike counting Bloom filters, at each element insertion, the hashed counters are incremented by a hashed variable increment instead of a unit increment. To query an element, the exact values of the coun-

ters are considered and not just their positiveness. If a sum represented by a counter value cannot be composed of the corresponding variable increment for the queried element, a negative answer can be returned to the query.

### Decentralized aggregation

Bloom filters can be organized in distributed data structures to perform fully decentralized computations of aggregate functions. Decentralized aggregation makes collective measurements locally available in every node of a distributed network without involving a centralized computational entity for this purpose.<sup>[18][19]</sup>

### Data synchronization

Bloom filters can be used for approximate data synchronization as in Byers et al. (2004). Counting Bloom filters can be used to approximate the number of differences between two sets and this approach is described in Agarwal & Trachtenberg (2006).

### Bloomier filters

Chazelle et al. (2004) designed a generalization of Bloom filters that could associate a value with each element that had been inserted, implementing an associative array. Like Bloom filters, these structures achieve a small space overhead by accepting a small probability of false positives. In the case of “Bloomier filters”, a *false positive* is defined as returning a result when the key is not in the map. The map will never return the wrong value for a key that *is* in the map.

### Compact approximators

Boldi & Vigna (2005) proposed a lattice-based generalization of Bloom filters. A **compact approximator** associates to each key an element of a lattice (the standard Bloom filters being the case of the Boolean two-element lattice). Instead of a bit array, they have an array of lattice elements. When adding a new association between a key and an element of the lattice, they compute the maximum of the current contents of the  $k$  array locations associated to the key with the lattice element. When reading the value associated to a key, they compute the minimum of the values found in the  $k$  locations associated to the key. The resulting value approximates from above the original value.

### Stable Bloom filters

Deng & Rafiei (2006) proposed Stable Bloom filters as a variant of Bloom filters for streaming data. The idea is that since there is no way to store the entire history of a

stream (which can be infinite), Stable Bloom filters continuously evict stale information to make room for more recent elements. Since stale information is evicted, the Stable Bloom filter introduces false negatives, which do not appear in traditional bloom filters. The authors show that a tight upper bound of false positive rates is guaranteed, and the method is superior to standard bloom filters in terms of false positive rates and time efficiency when a small space and an acceptable false positive rate are given.

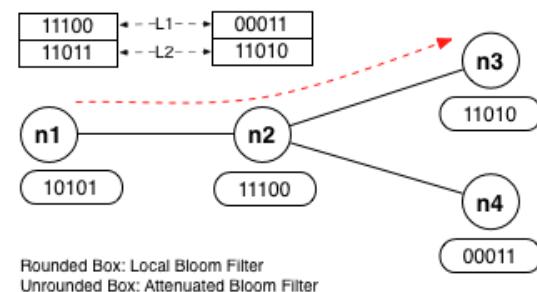
### Scalable Bloom filters

Almeida et al. (2007) proposed a variant of Bloom filters that can adapt dynamically to the number of elements stored, while assuring a minimum false positive probability. The technique is based on sequences of standard bloom filters with increasing capacity and tighter false positive probabilities, so as to ensure that a maximum false positive probability can be set beforehand, regardless of the number of elements to be inserted.

### Layered Bloom filters

A layered bloom filter consists of multiple bloom filter layers. Layered bloom filters allow keeping track of how many times an item was added to the bloom filter by checking how many layers contain the item. With a layered bloom filter a check operation will normally return the deepest layer number the item was found in.<sup>[20][21]</sup>

### Attenuated Bloom filters



Attenuated Bloom Filter Example

An attenuated bloom filter of depth  $D$  can be viewed as an array of  $D$  normal bloom filters. In the context of service discovery in a network, each node stores regular and attenuated bloom filters locally. The regular or local bloom filter indicates which services are offered by the node itself. The attenuated filter of level  $i$  indicates which services can be found on nodes that are  $i$ -hops away from the current node. The  $i$ -th value is constructed by taking a union of local bloom filters for nodes  $i$ -hops away from the node.<sup>[22]</sup>

Let's take a small network shown on the graph below as an example. Say we are searching for a service A whose id hashes to bits 0,1, and 3 (pattern 11010). Let n1 node to be the starting point. First, we check whether service A is offered by n1 by checking its local filter. Since the patterns don't match, we check the attenuated bloom filter in order to determine which node should be the next hop. We see that n2 doesn't offer service A but lies on the path to nodes that do. Hence, we move to n2 and repeat the same procedure. We quickly find that n3 offers the service, and hence the destination is located.<sup>[23]</sup>

By using attenuated Bloom filters consisting of multiple layers, services at more than one hop distance can be discovered while avoiding saturation of the Bloom filter by attenuating (shifting out) bits set by sources further away.<sup>[22]</sup>

### Chemical structure searching

Bloom filters are often used to search large chemical structure databases (see [chemical similarity](#)). In the simplest case, the elements added to the filter (called a fingerprint in this field) are just the atomic numbers present in the molecule, or a hash based on the atomic number of each atom and the number and type of its bonds. This case is too simple to be useful. More advanced filters also encode atom counts, larger substructure features like carboxyl groups, and graph properties like the number of rings. In hash-based fingerprints, a hash function based on atom and bond properties is used to turn a subgraph into a [PRNG](#) seed, and the first output values used to set bits in the Bloom filter.

Molecular fingerprints arose as a way to screen out obvious rejects in [molecular subgraph searches](#). They are more often used in calculating molecular similarity, computed as the [Tanimoto](#) similarity between their respective fingerprints. The Daylight and Indigo fingerprints, use a variable length (which are restricted to be powers of two) that adapts to the number of items to ensure the final filter is not oversaturated. A length for the filter is chosen to keep the saturation of the filter approximately at a fixed value (for example 30%).

Strictly speaking, the MACCS keys and CACTVS keys are not Bloom filters, rather they are deterministic bit-arrays. Similarly, they are often incorrectly considered to be molecular fingerprints, but they are actually "structural keys". They do not use hash functions, but use a dictionary to map specific substructures to specific indices in the filter. In contrast with Bloom filters, this is a one-to-one mapping that does not use hash functions at all.

### 11.18.10 See also

- Count–min sketch
- Feature hashing

- MinHash
- Quotient filter
- Skip list

### 11.18.11 Notes

- [1] Mitzenmacher & Upfal (2005).
- [2] Blustein & El-Maazawi (2002), pp. 21–22
- [3] Mitzenmacher & Upfal (2005), pp. 109–111, 308.
- [4] Mitzenmacher & Upfal (2005), p. 308.
- [5] Starobinski, Trachtenberg & Agarwal (2003)
- [6] Goel & Gupta (2010).
- [7] (Chang et al. 2006).
- [8] Yakunin, Alex (2010-03-25). "Alex Yakunin's blog: Nice Bloom filter application". Blog.alexyakunin.com. Retrieved 2014-05-31.
- [9] "Issue 10896048: Transition safe browsing from bloom filter to prefix set. - Code Review". Chromiumcodereview.appspot.com. Retrieved 2014-07-03.
- [10] Wessels, Duane (January 2004), "10.7 Cache Digests", *Squid: The Definitive Guide* (1st ed.), O'Reilly Media, p. 172, ISBN 0-596-00162-2, Cache Digests are based on a technique first published by Pei Cao, called Summary Cache. The fundamental idea is to use a Bloom filter to represent the cache contents.
- [11] Bitcoin 0.8.0
- [12] Core Development Status Report #1
- [13] "Plan 9 /sys/man/8/venti". Plan9.bell-labs.com. Retrieved 2014-05-31.
- [14] <http://spinroot.com/>
- [15] Mullin (1990)
- [16] "BloomJoin: BloomFilter + CoGroup | LiveRamp Blog". Blog.liveramp.com. Retrieved 2014-05-31.
- [17] "Exim source code". github. Retrieved 2014-03-03.
- [18] Pournaras, Warnier & Brazier (2013)
- [19] "DIAS - Dynamic Intelligent Aggregation Service".
- [20] "pmylund/go-bloom". Github.com. Retrieved 2014-06-13.
- [21] First Name Middle Name Last Name (2010-08-22). "IEEE Xplore Abstract - A multi-layer bloom filter for duplicated URL detection". Ieeexplore.ieee.org. doi:10.1109/ICACTE.2010.5578947. Retrieved 2014-06-13.
- [22] Koucheryavy et al. (2009)
- [23] Kubiatowicz et al. (2000)

### 11.18.12 References

- Koucheryavy, Y.; Giambene, G.; Staehle, D.; Barcelo-Arroyo, F.; Braun, T.; Siris, V. (2009), “Traffic and QoS Management in Wireless Multi-media Networks”, *COST 290 Final Report* (USA): 111
- Kubiakowicz, J.; Bindel, D.; Czerwinski, Y.; Geels, S.; Eaton, D.; Gummadi, R.; Rhea, S.; Weatherspoon, H. et al. (2000), “Oceanstore: An architecture for global-scale persistent storage” (PDF), *ACM SIGPLAN Notices* (USA): 190–201
- Agarwal, Sachin; Trachtenberg, Ari (2006), “Approximating the number of differences between remote sets” (PDF), *IEEE Information Theory Workshop* (Punta del Este, Uruguay): 217, doi:10.1109/ITW.2006.1633815, ISBN 1-4244-0035-X
- Ahmadi, Mahmood; Wong, Stephan (2007), “A Cache Architecture for Counting Bloom Filters”, *15th international Conference on Networks (ICON-2007)*, p. 218, doi:10.1109/ICON.2007.4444089, ISBN 978-1-4244-1229-7
- Almeida, Paulo; Baquero, Carlos; Preguica, Nuno; Hutchison, David (2007), “Scalable Bloom Filters” (PDF), *Information Processing Letters* **101** (6): 255–261, doi:10.1016/j.ipl.2006.10.007
- Byers, John W.; Considine, Jeffrey; Mitzenmacher, Michael; Rost, Stanislav (2004), “Informed content delivery across adaptive overlay networks”, *IEEE/ACM Transactions on Networking* **12** (5): 767, doi:10.1109/TNET.2004.836103
- Bloom, Burton H. (1970), “Space/Time Trade-offs in Hash Coding with Allowable Errors”, *Communications of the ACM* **13** (7): 422–426, doi:10.1145/362686.362692
- Boldi, Paolo; Vigna, Sebastiano (2005), “Mutable strings in Java: design, implementation and lightweight text-search algorithms”, *Science of Computer Programming* **54** (1): 3–23, doi:10.1016/j.scico.2004.05.003
- Bonomi, Flavio; Mitzenmacher, Michael; Panigrahy, Rina; Singh, Sushil; Varghese, George (2006), “An Improved Construction for Counting Bloom Filters”, *Algorithms – ESA 2006, 14th Annual European Symposium* (PDF), Lecture Notes in Computer Science **4168**, pp. 684–695, doi:10.1007/11841036\_61, ISBN 978-3-540-38875-3
- Broder, Andrei; Mitzenmacher, Michael (2005), “Network Applications of Bloom Filters: A Survey” (PDF), *Internet Mathematics* **1** (4): 485–509, doi:10.1080/15427951.2004.10129096
- Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson; Wallach, Deborah; Burrows, Mike; Chandra, Tushar; Fikes, Andrew; Gruber, Robert (2006), “Bigtable: A Distributed Storage System for Structured Data”, *Seventh Symposium on Operating System Design and Implementation*
- Charles, Denis; Chellapilla, Kumar (2008), “Bloomier Filters: A second look”, *The Computing Research Repository (CoRR)*, arXiv:0807.0928
- Chazelle, Bernard; Kilian, Joe; Rubinfeld, Ronitt; Tal, Ayellet (2004), “The Bloomier filter: an efficient data structure for static support lookup tables”, *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (PDF), pp. 30–39
- Cohen, Saar; Matias, Yossi (2003), “Spectral Bloom Filters”, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (PDF), pp. 241–252, doi:10.1145/872757.872787, ISBN 158113634X
- Deng, Fan; Rafiei, Davood (2006), “Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters”, *Proceedings of the ACM SIGMOD Conference* (PDF), pp. 25–36
- Dharmapurikar, Sarang; Song, Haoyu; Turner, Jonathan; Lockwood, John (2006), “Fast packet classification using Bloom filters”, *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems* (PDF), pp. 61–70, doi:10.1145/1185347.1185356, ISBN 1595935800
- Dietzfelbinger, Martin; Pagh, Rasmus (2008), “Succinct Data Structures for Retrieval and Approximate Membership”, *The Computing Research Repository (CoRR)*, arXiv:0803.3693
- Swamidass, S. Joshua; Baldi, Pierre (2007), “Mathematical correction for fingerprint similarity measures to improve chemical retrieval”, *Journal of chemical information and modeling* (ACS Publications) **47** (3): 952–964, doi:10.1021/ci600526a, PMID 17444629
- Dillinger, Peter C.; Manolios, Panagiotis (2004a), “Fast and Accurate Bitstate Verification for SPIN”, *Proceedings of the 11th International Spin Workshop on Model Checking Software*, Springer-Verlag, Lecture Notes in Computer Science 2989
- Dillinger, Peter C.; Manolios, Panagiotis (2004b), “Bloom Filters in Probabilistic Verification”, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*, Springer-Verlag, Lecture Notes in Computer Science 3312

- Donnet, Benoit; Baynat, Bruno; Friedman, Timur (2006), “Retouched Bloom Filters: Allowing Networked Applications to Flexibly Trade Off False Positives Against False Negatives”, *CoNEXT 06 – 2nd Conference on Future Networking Technologies*
- Eppstein, David; Goodrich, Michael T. (2007), “Space-efficient straggler identification in round-trip data streams via Newton’s identities and invertible Bloom filters”, *Algorithms and Data Structures, 10th International Workshop, WADS 2007*, Springer-Verlag, Lecture Notes in Computer Science 4619, pp. 637–648, [arXiv:0704.3313](https://arxiv.org/abs/0704.3313)
- Fan, Li; Cao, Pei; Almeida, Jussara; Broder, Andrei (2000), “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol”, *IEEE/ACM Transactions on Networking* **8** (3): 281–293, doi:10.1109/90.851975. A preliminary version appeared at SIGCOMM ’98.
- Goel, Ashish; Gupta, Pankaj (2010), “Small subset queries and bloom filters using ternary associative memories, with applications”, *ACM Sigmetrics 2010* **38**: 143, doi:10.1145/1811099.1811056
- Kirsch, Adam; Mitzenmacher, Michael (2006), “Less Hashing, Same Performance: Building a Better Bloom Filter”, in Azar, Yossi; Erlebach, Thomas, *Algorithms – ESA 2006, 14th Annual European Symposium* (PDF), Lecture Notes in Computer Science **4168**, Springer-Verlag, Lecture Notes in Computer Science 4168, pp. 456–467, doi:10.1007/11841036, ISBN 978-3-540-38875-3
- Mortensen, Christian Worm; Pagh, Rasmus; Pătrașcu, Mihai (2005), “On dynamic range reporting in one dimension”, *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, pp. 104–111, doi:10.1145/1060590.1060606, ISBN 1581139608
- Pagh, Anna; Pagh, Rasmus; Rao, S. Srinivasa (2005), “An optimal Bloom filter replacement”, *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (PDF), pp. 823–829
- Porat, Ely (2008), “An Optimal Bloom Filter Replacement Based on Matrix Solving”, *The Computing Research Repository (CoRR)*, arXiv:0804.1845
- Putze, F.; Sanders, P.; Singler, J. (2007), “Cache-, Hash- and Space-Efficient Bloom Filters”, in Demetrescu, Camil, *Experimental Algorithms, 6th International Workshop, WEA 2007* (PDF), Lecture Notes in Computer Science **4525**, Springer-Verlag, Lecture Notes in Computer Science 4525, pp. 108–121, doi:10.1007/978-3-540-72845-0, ISBN 978-3-540-72844-3
- Sethumadhavan, Simha; Desikan, Rajagopalan; Burger, Doug; Moore, Charles R.; Keckler, Stephen W. (2003), “Scalable hardware memory disambiguation for high ILP processors”, *36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003, MICRO-36* (PDF), pp. 399–410, doi:10.1109/MICRO.2003.1253244, ISBN 0-7695-2043-X
- Shanmugasundaram, Kulesh; Brönnimann, Hervé; Memon, Nasir (2004), “Payload attribution via hierarchical Bloom filters”, *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pp. 31–41, doi:10.1145/1030083.1030089, ISBN 1581139616
- Starobinski, David; Trachtenberg, Ari; Agarwal, Sachin (2003), “Efficient PDA Synchronization”, *IEEE Transactions on Mobile Computing* **2** (1): 40, doi:10.1109/TMC.2003.1195150
- Stern, Ulrich; Dill, David L. (1996), “A New Scheme for Memory-Efficient Probabilistic Verification”, *Proceedings of Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification: IFIP TC6/WG6.1 Joint International Conference*, Chapman & Hall, IFIP Conference Proceedings, pp. 333–348, [CiteSeerX: 10.1.1.47.4101](https://www.citeSeerX.info/10.1.1.47.4101)
- Haghagh, Mohammad Hashem; Tavakoli, Mehdi; Kharrazi, Mehdi (2013), “Payload Attribution via Character Dependent Multi-Bloom Filters”, *Transaction on Information Forensics and Security, IEEE* **99** (5): 705, doi:10.1109/TIFS.2013.2252341
- Mitzenmacher, Michael; Upfal, Eli (2005), *Probability and computing: Randomized algorithms and probabilistic analysis*, Cambridge University Press, pp. 107–112, ISBN 9780521835404
- Mullin, James K. (1990), “Optimal semijoins for distributed database systems”, *Software Engineering, IEEE Transactions on* **16** (5): 558–560, doi:10.1109/32.52778
- Rottenstreich, Ori; Kanizo, Yossi; Keslassy, Isaac (2012), “The Variable-Increment Counting Bloom Filter”, *31st Annual IEEE International Conference on Computer Communications, 2012, Infocom 2012* (PDF), pp. 1880–1888, doi:10.1109/INFCOM.2012.6195563, ISBN 978-1-4673-0773-4
- Blustein, James; El-Maazawi, Amal (2002), “optimal case for general Bloom filters”, *Bloom Filters — A Tutorial, Analysis, and Survey*, Dalhousie University Faculty of Computer Science, pp. 1–31
- Pournaras, E.; Warnier, M.; Brazier, F.M.T.. (2013), “A generic and adaptive aggregation service

for large-scale decentralized networks”, *Complex Adaptive Systems Modeling* 1:19, doi:10.1186/2194-3206-1-19

### 11.18.13 External links

- Why Bloom filters work the way they do (Michael Nielsen, 2012)
- Bloom Filters — A Tutorial, Analysis, and Survey (Blustein & El-Maazawi, 2002) at Dalhousie University
- Table of false-positive rates for different configurations from a University of Wisconsin–Madison website
- Interactive Processing demonstration from ashcan.org
- “More Optimal Bloom Filters,” Ely Porat (Nov/2007) Google TechTalk video on YouTube
- “Using Bloom Filters” Detailed Bloom Filter explanation using Perl
- “A Garden Variety of Bloom Filters - Explanation and Analysis of Bloom filter variants
- “Bloom filters, fast and simple” - Explanation and example implementation in Python

## 11.19 Locality preserving hashing

In computer science, a **locality preserving hashing** is a hash function  $f$  that maps a point or points in a multidimensional coordinate space to a scalar value, such that if we have three points  $A$ ,  $B$  and  $C$  such that

$$|A - B| < |B - C| \Rightarrow |f(A) - f(B)| < |f(B) - f(C)|.$$

In other words, these are hash functions where the relative distance between the input values is preserved in the relative distance between the output hash values; input values that are closer to each other will produce output hash values that are closer to each other.

This is in contrast to cryptographic hash functions and checksums, which are designed to have maximum output difference between adjacent inputs.

Locality preserving hashes are related to space-filling curves and locality sensitive hashing.

### 11.19.1 External links

- Indyk, Piotr; Motwani, Rajeev; Raghavan, Prabhakar; Vempala, Santosh (1997). “Locality-preserving hashing in multidimensional spaces”.

*Proceedings of the twenty-ninth annual ACM symposium on Theory of computing.* pp. 618–625. doi:10.1145/258533.258656. ISBN 0-89791-888-6. CiteSeerX: 10.1.1.50.4927.

- Chin, Andrew (1994). “Locality-preserving hash functions for general purpose parallel computation” (PDF). *Algorithmica* 12 (2–3): 170–181. doi:10.1007/BF01185209.

## 11.20 Zobrist hashing

**Zobrist hashing** (also referred to as **Zobrist keys** or **Zobrist signatures**<sup>[1]</sup>) is a hash function construction used in computer programs that play abstract board games, such as chess and Go, to implement transposition tables, a special kind of hash table that is indexed by a board position and used to avoid analyzing the same position more than once. Zobrist hashing is named for its inventor, **Albert Lindsey Zobrist**.<sup>[2]</sup> It has also been applied as a method for recognizing substitutional alloy configurations in simulations of crystalline materials.<sup>[3]</sup>

### 11.20.1 Calculation of the hash value

Zobrist hashing starts by randomly generating bitstrings for each possible element of a board game, i.e. for each combination of a piece and a position (in the game of chess, that's 12 pieces  $\times$  64 board positions, or 14 if a king that may still castle and a pawn that may capture *en passant* are treated separately). Now any board configuration can be broken up into independent piece/position components, which are mapped to the random bitstrings generated earlier. The final Zobrist hash is computed by combining those bitstrings using bitwise **XOR**. Example pseudocode for the game of chess:

```
constant indices white_pawn := 1 white_rook := 2 # etc.
black_king := 12 function init_zobrist(): # fill a table of
random numbers/bitstrings table := a 2-d array of size
64x12 for i from 1 to 64: # loop over the board,
represented as a linear array for j from 1 to 12: # loop
over the pieces table[i][j] = random_bitstring() function
hash(board): h := 0 for i from 1 to 64: # loop over the
board positions if board[i] != empty: j := the piece at
board[i], as listed in the constant indices, above h := h
XOR table[i][j] return h
```

### 11.20.2 Use of the hash value

If the bitstrings are long enough, different board positions will almost certainly hash to different values; however longer bitstrings require proportionally more computer resources to manipulate. Many game engines store only the hash values in the transposition table, omitting the position information itself entirely to reduce memory

usage, and assuming that hash collisions will not occur, or will not greatly influence the results of the table if they do.

Zobrist hashing is the first known instance of tabulation hashing. The result is a 3-wise independent hash family. In particular, it is strongly universal.

As an example, in chess, each of the 64 squares can at any time be empty, or contain one of the 6 game pieces, which are either black or white. That is, each square can be in one of  $1 + 6 \times 2 = 13$  possible states at any time. Thus one needs to generate at most  $13 \times 64 = 832$  random bitstrings. Given a position, one obtains its Zobrist hash by finding out which pieces are on which squares, and combining the relevant bitstrings together.

### 11.20.3 Updating the hash value

Rather than computing the hash for the entire board every time, as the pseudocode above does, the hash value of a board can be updated simply by XORing out the bitstring(s) for positions that have changed, and XORing in the bitstrings for the new positions. For instance, if a pawn on a chessboard square is replaced by a rook from another square, the resulting position would be produced by XORing the existing hash with the bitstrings for:

'pawn at this square' (XORing *out* the pawn at this square)  
 'rook at this square' (XORing *in* the rook at this square)  
 'rook at source square' (XORing *out* the rook at the source square) 'nothing at source square' (XORing *in* nothing at the source square).

This makes Zobrist hashing very efficient for traversing a game tree.

In computer go, this technique is also used for superko detection.

### 11.20.4 Wider usage

The same method has been used to recognize substitutional alloy configurations during Monte Carlo simulations in order to prevent wasting computational effort on states that have already been calculated.<sup>[3]</sup>

### 11.20.5 See also

- Alpha-beta pruning

### 11.20.6 References

- [1] Zobrist keys: a means of enabling position comparison.
- [2] Albert Lindsey Zobrist, *A New Hashing Method with Application for Game Playing*, Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, (1969).

- [3] Mason, D. R.; Hudson, T. S.; Sutton, A. P. (2005). "Fast recall of state-history in kinetic Monte Carlo simulations utilizing the Zobrist key". *Computer Physics Communications* **165**: 37. Bibcode:2005CoPhC.165..37M. doi:10.1016/j.cpc.2004.09.007.

## 11.21 Rolling hash

A **rolling hash** is a **hash function** where the input is hashed in a window that moves through the input.

A few hash functions allow a rolling hash to be computed very quickly—the new hash value is rapidly calculated given only the old hash value, the old value removed from the window, and the new value added to the window—similar to the way a moving average function can be computed much more quickly than other low-pass filters.

One of the main applications is the **Rabin-Karp** string search algorithm, which uses the rolling hash described below.

Another popular application is **rsync** program which uses a checksum based on Mark Adler's **adler-32** as its rolling hash.

Another application is the **Low Bandwidth Network Filesystem** (LBFS), which uses a **Rabin** fingerprint as its rolling hash.

At best, rolling hash values are pairwise independent<sup>[1]</sup> or strongly universal. They cannot be 3-wise independent, for example.

### 11.21.1 Rabin-Karp rolling hash

The **Rabin-Karp** string search algorithm is normally used with a very simple rolling hash function that only uses multiplications and additions:

$$H = c_1 a^{k-1} + c_2 a^{k-2} + c_3 a^{k-3} + \dots + c_k a^0 \text{ where } a \text{ is a constant and } c_1, \dots, c_k \text{ are the input characters.}$$

In order to avoid manipulating huge  $H$  values, all math is done modulo  $n$ . The choice of  $a$  and  $n$  is critical to get good hashing; see **linear congruential generator** for more discussion.

Removing and adding characters simply involves adding or subtracting the first or last term. Shifting all characters by one position to the left requires multiplying the entire sum  $H$  by  $a$ . Shifting all characters by one position to the right requires dividing the entire sum  $H$  by  $a$ . Note that in modulo arithmetic,  $a$  can be chosen to have a multiplicative inverse  $a^{-1}$  by which  $H$  can be multiplied to get the result of the division without actually performing a division.

### 11.21.2 Content based slicing using Rabin-Karp hash

One of the interesting use cases of the rolling hash function is that it can create dynamic, content-based chunks of a stream or file. This is especially useful when it is required to send only the changed chunks of a large file over a network and a simple byte addition at the front of the file would cause all the fixed size windows to become updated, while in reality, only the first ‘chunk’ has been modified.

The simplest approach to calculate the dynamic chunks is to calculate the rolling hash and if it matches a pattern (like the lower  $N$  bits are all zeroes) then it’s a chunk boundary. This approach will ensure that any change in the file will only affect its current and possibly the next chunk, but nothing else.

When the boundaries are known, the chunks need to be compared by their hash values to detect which one was modified and needs transfer across the network.<sup>[2]</sup>

### 11.21.3 Cyclic polynomial

Hashing by cyclic polynomial<sup>[3]</sup>—sometimes called Buzhash—is also simple, but it has the benefit of avoiding multiplications, using barrel shifts instead. It is a form of tabulation hashing: it presumes that there is some hash function  $h$  from characters to integers in the interval  $[0, 2^L]$ . This hash function might be simply an array or a hash table mapping characters to random integers. Let the function  $s$  be a cyclic binary rotation (or barrel shift): it rotates the bits by 1 to the left, pushing the latest bit in the first position. E.g.,  $s(10011) = 00111$ . Let  $\oplus$  be the bit-wise **exclusive or**. The hash values are defined as

$$H = s^{k-1}(h(c_1)) \oplus s^{k-2}(h(c_2)) \oplus \dots \oplus s(h(c_{k-1})) \oplus h(c_k)$$

where the multiplications by powers of two can be implemented by binary shifts. The result is a number in  $[0, 2^L)$

Computing the hash values in a rolling fashion is done as follows. Let  $H$  be the previous hash value. Rotate  $H$  once:  $H \leftarrow s(H)$ . If  $c_1$  is the character to be removed, rotate it  $k$  times:  $s^k(h(c_1))$ . Then simply set

$$H \leftarrow s(H) \oplus s^k(h(c_1)) \oplus h(c_{k+1})$$

where  $c_{k+1}$  is the new character.

Hashing by cyclic polynomials is strongly universal or pairwise independent: simply keep the first  $L - k + 1$  bits. That is, take the result  $H$  and dismiss any  $k - 1$  consecutive bits.<sup>[1]</sup> In practice, this can be achieved by an integer division  $H \rightarrow H \div 2^{k-1}$ .

### 11.21.4 Computational complexity

All rolling hash functions are linear in the number of characters, but their complexity with respect to the length of the window ( $k$ ) varies. Rabin-Karp rolling hash requires the multiplications of two  $k$ -bit numbers, **integer multiplication** is in  $O(k \log k 2^{O(\log^* k)})$ .<sup>[4]</sup> Hashing **ngrams** by cyclic polynomials can be done in linear time.<sup>[1]</sup>

### 11.21.5 Software

- `rollinghashcpp` is a **Free software** C++ implementation of several rolling hash functions
- `rollinghashjava` is an Apache licensed Java implementation of rolling hash functions

### 11.21.6 See also

- `MinHash`
- `w-shingling`

### 11.21.7 External links

- MIT 6.006: Introduction to Algorithms 2011- Lecture Notes - Rolling Hash

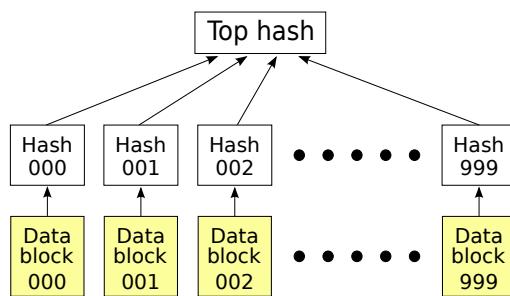
### 11.21.8 Footnotes

- [1] Daniel Lemire, Owen Kaser: Recursive n-gram hashing is pairwise independent, at best, Computer Speech & Language 24 (4), pages 698-710, 2010. [arXiv:0705.4676](https://arxiv.org/abs/0705.4676)
- [2] Horvath, Adam (October 24, 2012). “Rabin Karp rolling hash - dynamic sized chunks based on hashed content”.
- [3] Jonathan D. Cohen, Recursive Hashing Functions for n-Grams, ACM Trans. Inf. Syst. 15 (3), 1997
- [4] M. Fürer, Faster integer multiplication, in: STOC '07, 2007, pp. 57–66.

## 11.22 Hash list

In **computer science**, a **hash list** is typically a **list** of hashes of the data blocks in a file or set of files. Lists of hashes are used for many different purposes, such as fast table lookup (hash tables) and distributed databases (distributed hash tables). This article covers hash lists that are used to guarantee data integrity.

A hash list is an extension of the old concept of hashing an item (for instance, a file). A hash list is usually sufficient for most needs, but a more advanced form of the concept is a **hash tree**.



A hash list with a top hash

Hash lists can be used to protect any kind of data stored, handled and transferred in and between computers. An important use of hash lists is to make sure that data blocks received from other peers in a peer-to-peer network are received undamaged and unaltered, and to check that the other peers do not “lie” and send fake blocks.

Usually a **cryptographic hash function** such as SHA-256 is used for the hashing. If the hash list only needs to protect against unintentional damage less secure checksums such as **CRCs** can be used.

Hash lists are better than a simple hash of the entire file since, in the case of a data block being damaged, this is noticed, and only the damaged block needs to be redownloaded. With only a hash of the file, many undamaged blocks would have to be redownloaded, and the file reconstructed and tested until the correct hash of the entire file is obtained. Hash lists also protect against nodes that try to sabotage by sending fake blocks, since in such a case the damaged block can be acquired from some other source.

### 11.22.1 Root hash

Often, an additional hash of the hash list itself (a *top hash*, also called *root hash* or *master hash*) is used. Before downloading a file on a p2p network, in most cases the top hash is acquired from a trusted source, for instance a friend or a web site that is known to have good recommendations of files to download. When the top hash is available, the hash list can be received from any non-trusted source, like any peer in the p2p network. Then the received hash list is checked against the trusted top hash, and if the hash list is damaged or fake, another hash list from another source will be tried until the program finds one that matches the top hash.

In some systems (like for example BitTorrent), instead of a top hash the whole hash list is available on a web site in a small file. Such a "torrent file" contains a description, file names, a hash list and some additional data.

### 11.22.2 See also

- Hash tree
- Hash table
- Hash chain
- Ed2k: URI scheme, which uses an MD4 top hash of an MD4 hash list to uniquely identify a file
- Cryptographic hash function
- List

## 11.23 Hash tree

Not to be confused with Hash trie.

In computer science, **hash tree** may refer to:

- Hashed array tree
- Hash tree (persistent data structure), an implementation strategy for sets and maps
- Merkle tree

## 11.24 Prefix hash tree

A **prefix hash tree** (PHT) is a distributed data structure that enables more sophisticated queries over a distributed hash table (DHT). The prefix hash tree uses the lookup interface of a DHT to construct a trie-based data structure that is both efficient (updates are doubly logarithmic in the size of the domain being indexed), and resilient (the failure of any given node in a prefix hash tree does not affect the availability of data stored at other nodes).

### 11.24.1 External links

- <https://www.eecs.berkeley.edu/~{}sylvia/papers/pht.pdf> - *Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables*
- <http://pier.cs.berkeley.edu> - PHT was developed as part of work on the PIER project.

### 11.24.2 See also

- Prefix tree
- P-Grid

## 11.25 Hash trie

In computer science, **hash trie** can refer to:

- **Hash tree (persistent data structure)**, a trie used to map hash values to keys
- A space-efficient implementation of a sparse **trie**, in which the descendants of each node may be interleaved in memory. (The name is suggested by a similarity to a closed **hash table**.) <sup>[1]</sup>
- A data structure which “combines features of hash tables and LC-tries (Least Compression tries) in order to perform efficient lookups and updates” <sup>[2]</sup>

### 11.25.1 See also

- Hash array mapped trie
- Hash tree

### 11.25.2 References

- [1] Liang, Frank (June 1983), *Word hy-phen-a-tion by computer* (PDF), Frank M. Liang, Ph.D. thesis, Stanford University., retrieved 2010-03-28 |first2= missing |last2= in Authors list ([help](#))
- [2] Thomas, Roshan; Mark, Brian; Johnson, Tommy; Croall, James (2004), *High-speed Legitimacy-based DDoS Packet Filtering with Network Processors: A Case Study and Implementation on the Intel IXP1200* (PDF), retrieved 2009-05-03

## 11.26 Hash array mapped trie

A **hash array mapped trie**<sup>[1]</sup> (**HAMT**) is an implementation of an **associative array** that combines the characteristics of a **hash table** and an **array mapped trie**.<sup>[1]</sup> It is a refined version of the more general notion of a **hash tree**.

### 11.26.1 Operation

A HAMT is an array mapped trie where the keys are first hashed in order to ensure an even distribution of keys and a constant key length.

In a typical implementation of HAMT’s array mapped trie, each node contains a table with some fixed number  $N$  of slots with each slot containing either a nil pointer or a pointer to another node.  $N$  is commonly 32. As allocating space for  $N$  pointers for each node would be expensive, each node instead contains a bitmap which is  $N$  bits long where each bit indicates the presence of a non-nil pointer. This is followed by an array of pointers equal in length to the number of ones in the bitmap, (its **Hamming weight**).

### 11.26.2 Advantages of HAMTs

The hash array mapped trie achieves almost hash table-like speed while using memory much more economically. Also, a hash table may have to be periodically resized, an expensive operation, whereas HAMTs grow dynamically. Generally, HAMT performance is improved by a larger root table with some multiple of  $N$  slots; some HAMT variants allow the root to grow lazily<sup>[1]</sup> with negligible impact on performance.

### 11.26.3 Implementation details

Implementation of a HAMT involves the use of the **population count** function, which counts the number of ones in the binary representation of a number. This operation is available in **many instruction set architectures**, but it is **available in only some high-level languages**. Although population count can be implemented in software in  $O(1)$  time using a series of **shift** and **add** instructions, doing so may perform the operation an order of magnitude slower.

### 11.26.4 Implementations

The programming languages **Clojure**<sup>[2]</sup> and **Scala** use a persistent variant of hash array mapped tries for their native hash map type. The Haskell library **unordered-containers** uses the same to implement persistent map and set data types.<sup>[3]</sup> A Javascript HAMT library <sup>[4]</sup> based on the Clojure implementation is also available. The **Rubinius**<sup>[5]</sup> implementation of **Ruby** includes a HAMT, mostly written in Ruby but with 3<sup>[6]</sup> primitives.

The concurrent lock-free version<sup>[7]</sup> of the hash trie called **Ctrie** is a mutable thread-safe implementation which ensures progress. The data-structure has been proven to be correct<sup>[8]</sup> - Ctrie operations have shown to have the atomicity, linearizability and lock-freedom properties.

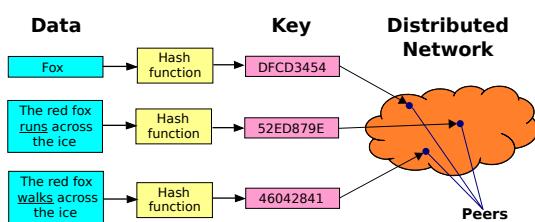
### 11.26.5 References

- [1] Phil Bagwell (2000). *Ideal Hash Trees* (PDF) (Report). Infoscience Department, École Polytechnique Fédérale de Lausanne.
- [2] Java source file of Clojure’s hash map type.
- [3] Johan Tibell, A. Announcing unordered-containers 0.2
- [4] Javascript HAMT library and source
- [5] Ruby source file of Rubinius’s HAMT
- [6]
- [7] Prokopec, A. Implementation of Concurrent Hash Tries on GitHub
- [8] Prokopec, A. et al. (2011) Cache-Aware Lock-Free Concurrent Hash Tries. Technical Report, 2011.

## 11.27 Distributed hash table

A **distributed hash table (DHT)** is a class of a decentralized distributed system that provides a lookup service similar to a **hash table**;  $(key, value)$  pairs are stored in a DHT, and any participating **node** can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to **scale** to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

DHTs form an infrastructure that can be used to build more complex services, such as anycast, cooperative Web caching, distributed file systems, domain name services, instant messaging, multicast, and also peer-to-peer file sharing and content distribution systems. Notable distributed networks that use DHTs include BitTorrent's distributed tracker, the **Coral Content Distribution Network**, the **Kad** network, the **Storm** botnet, the **Tox** instant messenger, Freenet and the **YaCy** search engine.



Distributed hash tables

### 11.27.1 History

DHT research was originally motivated, in part, by peer-to-peer systems such as **Freenet**, **gnutella**, **BitTorrent** and **Napster**, which took advantage of resources distributed across the Internet to provide a single useful application. In particular, they took advantage of increased **bandwidth** and **hard disk capacity** to provide a file-sharing service.

These systems differed in how they *found* the data their peers contained:

- Napster, the first large-scale P2P content delivery system, required a central index server: each node, upon joining, would send a list of locally held files to the server, which would perform searches and refer the querier to the nodes that held the results. This central component left the system vulnerable to attacks and lawsuits.
- Gnutella and similar networks moved to a flooding query model – in essence, each search would result in a message being broadcast to every other machine

in the network. While avoiding a **single point of failure**, this method was significantly less efficient than Napster. Later versions of Gnutella clients moved to a dynamic querying model which vastly improved efficiency.

- Finally, **Freenet** is fully distributed, but employs a heuristic **key-based routing** in which each file is associated with a key, and files with similar keys tend to cluster on a similar set of nodes. Queries are likely to be routed through the network to such a cluster without needing to visit many peers.<sup>[1]</sup> However, Freenet does not guarantee that data will be found.

Distributed hash tables use a more structured key-based routing in order to attain both the decentralization of Freenet and gnutella, and the efficiency and guaranteed results of Napster. One drawback is that, like Freenet, DHTs only directly support exact-match search, rather than keyword search, although Freenet's routing algorithm can be generalized to any key type where a closeness operation can be defined.<sup>[2]</sup>

In 2001, four systems—**CAN**,<sup>[3]</sup> **Chord**,<sup>[4]</sup> **Pastry**, and **Tapestry**—ignited DHTs as a popular research topic. A project called the Infrastructure for Resilient Internet Systems (Iris) was funded by a \$12 million grant from the US **National Science Foundation** in 2002.<sup>[5]</sup> Researchers included Hari Balakrishnan and Scott Shenker.<sup>[6]</sup> Outside academia, DHT technology has been adopted as a component of **BitTorrent** and in the **Coral Content Distribution Network**.

### 11.27.2 Properties

DHTs characteristically emphasize the following properties:

- **Autonomy and Decentralization:** the nodes collectively form the system without any central coordination.
- **Fault tolerance:** the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.
- **Scalability:** the system should function efficiently even with thousands or millions of nodes.

A key technique used to achieve these goals is that any one node needs to coordinate with only a few other nodes in the system – most commonly,  $O(\log n)$  of the  $n$  participants (see below) – so that only a limited amount of work needs to be done for each change in membership.

Some DHT designs seek to be **secure** against malicious participants<sup>[7]</sup> and to allow participants to remain anonymous, though this is less common than in many

other peer-to-peer (especially **file sharing**) systems; see anonymous P2P.

Finally, DHTs must deal with more traditional distributed systems issues such as **load balancing**, **data integrity**, and performance (in particular, ensuring that operations such as routing and data storage or retrieval complete quickly).

### 11.27.3 Structure

The structure of a DHT can be decomposed into several main components.<sup>[8][9]</sup> The foundation is an abstract **keyspace**, such as the set of 160-bit strings. A **keyspace partitioning** scheme splits ownership of this keyspace among the participating nodes. An **overlay network** then connects the nodes, allowing them to find the owner of any given key in the keyspace.

Once these components are in place, a typical use of the DHT for storage and retrieval might proceed as follows. Suppose the keyspace is the set of 160-bit strings. To store a file with given *filename* and *data* in the DHT, the **SHA-1** hash of *filename* is generated, producing a 160-bit key  $k$ , and a message  $put(k, data)$  is sent to any node participating in the DHT. The message is forwarded from node to node through the overlay network until it reaches the single node responsible for key  $k$  as specified by the keyspace partitioning. That node then stores the key and the data. Any other client can then retrieve the contents of the file by again hashing *filename* to produce  $k$  and asking any DHT node to find the data associated with  $k$  with a message  $get(k)$ . The message will again be routed through the overlay to the node responsible for  $k$ , which will reply with the stored *data*.

The keyspace partitioning and overlay network components are described below with the goal of capturing the principal ideas common to most DHTs; many designs differ in the details.

#### Keyspace partitioning

Most DHTs use some variant of **consistent hashing** to map keys to nodes. This technique employs a function  $\delta(k_1, k_2)$  that defines an abstract notion of the *distance* between the keys  $k_1$  and  $k_2$ , which is unrelated to geographical **distance** or network **latency**. Each node is assigned a single key called its *identifier* (ID). A node with ID  $i_x$  owns all the keys  $k_m$  for which  $i_x$  is the closest ID, measured according to  $\delta(k_m, i_x)$ .

**Example.** The Chord DHT treats keys as points on a circle, and  $\delta(k_1, k_2)$  is the distance traveling clockwise around the circle from  $k_1$  to  $k_2$ . Thus, the circular keyspace is split into contiguous segments whose endpoints are the node identifiers. If  $i_1$  and  $i_2$  are two adjacent

IDs, with a shorter clockwise distance from  $i_1$  to  $i_2$ , then the node with ID  $i_2$  owns all the keys that fall between  $i_1$  and  $i_2$ .

Consistent hashing has the essential property that removal or addition of one node changes only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected. Contrast this with a traditional **hash table** in which addition or removal of one bucket causes nearly the entire keyspace to be remapped. Since any change in ownership typically corresponds to bandwidth-intensive movement of objects stored in the DHT from one node to another, minimizing such reorganization is required to efficiently support high rates of churn (node arrival and failure).

Locality-preserving hashing ensures that similar keys are assigned to similar objects. This can enable a more efficient execution of range queries. Self-Chord<sup>[10]</sup> decouples object keys from peer IDs and sorts keys along the ring with a statistical approach based on the **swarm intelligence** paradigm. Sorting ensures that similar keys are stored by neighbour nodes and that discovery procedures, including range queries, can be performed in logarithmic time.

#### Overlay network

Each node maintains a set of **links** to other nodes (its **neighbors** or **routing table**). Together, these links form the **overlay network**. A node picks its neighbors according to a certain structure, called the **network's topology**.

All DHT topologies share some variant of the most essential property: for any key  $k$ , each node either has a node ID that owns  $k$  or has a link to a node whose node ID is *closer* to  $k$ , in terms of the keyspace distance defined above. It is then easy to route a message to the owner of any key  $k$  using the following **greedy algorithm** (that is not necessarily globally optimal): at each step, forward the message to the neighbor whose ID is closest to  $k$ . When there is no such neighbor, then we must have arrived at the closest node, which is the owner of  $k$  as defined above. This style of routing is sometimes called **key-based routing**.

Beyond basic routing correctness, two important constraints on the topology are to guarantee that the maximum number of **hops** in any route (route length) is low, so that requests complete quickly; and that the maximum number of neighbors of any node (maximum node **degree**) is low, so that maintenance overhead is not excessive. Of course, having shorter routes requires higher maximum degree. Some common choices for maximum degree and route length are as follows, where  $n$  is the number of nodes in the DHT, using **Big O** notation:

The most common choice,  $O(\log n)$  degree/route length, is not optimal in terms of degree/route length tradeoff, as such topologies typically allow more flexibility in choice

of neighbors. Many DHTs use that flexibility to pick neighbors that are close in terms of latency in the physical underlying network.

Maximum route length is closely related to **diameter**: the maximum number of hops in any shortest path between nodes. Clearly, the network's worst case route length is at least as large as its diameter, so DHTs are limited by the degree/diameter tradeoff<sup>[11]</sup> that is fundamental in graph theory. Route length can be greater than diameter, since the greedy routing algorithm may not find shortest paths.<sup>[12]</sup>

### Algorithms for overlay networks

Aside from routing, there exist many algorithms that exploit the structure of the overlay network for sending a message to all nodes, or a subset of nodes, in a DHT.<sup>[13]</sup> These algorithms are used by applications to do **overlay multicast**, range queries, or to collect statistics. Two systems that are based on this approach are Structella,<sup>[14]</sup> which implements flooding and random walks on a Pastry overlay, and DQ-DHT,<sup>[15]</sup> which implements a dynamic querying search algorithm over a Chord network.

#### 11.27.4 DHT implementations

Most notable differences encountered in practical instances of DHT implementations include at least the following:

- The address space is a parameter of DHT. Several real world DHTs use 128-bit or 160-bit key space
- Some real-world DHTs use hash functions other than SHA-1.
- In the real world the key  $k$  could be a hash of a file's *content* rather than a hash of a file's *name* to provide **content-addressable storage**, so that renaming of the file does not prevent users from finding it.
- Some DHTs may also publish objects of different types. For example, key  $k$  could be the node *ID* and associated data could describe how to contact this node. This allows publication-of-presence information and often used in IM applications, etc. In the simplest case, *ID* is just a random number that is directly used as key  $k$  (so in a 160-bit DHT *ID* will be a 160-bit number, usually randomly chosen). In some DHTs, publishing of nodes IDs is also used to optimize DHT operations.
- Redundancy can be added to improve reliability. The  $(k, data)$  key pair can be stored in more than one node corresponding to the key. Usually, rather than selecting just one node, real world DHT algorithms select  $i$  suitable nodes, with  $i$  being an implementation-specific parameter of the DHT. In

some DHT designs, nodes agree to handle a certain keyspace range, the size of which may be chosen dynamically, rather than hard-coded.

- Some advanced DHTs like **Kademlia** perform iterative lookups through the DHT first in order to select a set of suitable nodes and send  $put(k, data)$  messages only to those nodes, thus drastically reducing useless traffic, since published messages are only sent to nodes that seem suitable for storing the key  $k$ ; and iterative lookups cover just a small set of nodes rather than the entire DHT, reducing useless forwarding. In such DHTs, forwarding of  $put(k, data)$  messages may only occur as part of a self-healing algorithm: if a target node receives a  $put(k, data)$  message, but believes that  $k$  is out of its handled range and a closer node (in terms of DHT keyspace) is known, the message is forwarded to that node. Otherwise, data are indexed locally. This leads to a somewhat self-balancing DHT behavior. Of course, such an algorithm requires nodes to publish their presence data in the DHT so the iterative lookups can be performed.

#### 11.27.5 Examples

#### 11.27.6 See also

- **Couchbase Server**: a persistent, replicated, clustered distributed object storage system compatible with memcached protocol.
- **Memcached**: a high-performance, distributed memory object caching system.
- **Prefix hash tree**: sophisticated querying over DHTs.
- **Merkle tree**: tree having every non-leaf node labelled with the hash of the labels of its children nodes.
- Most **distributed data stores** employ some form of DHT for lookup.

#### 11.27.7 References

- [1] *Searching in a Small World Chapters 1 & 2* (PDF), retrieved 2012-01-10
- [2] “Section 5.2.2”, *A Distributed Decentralized Information Storage and Retrieval System* (PDF), retrieved 2012-01-10
- [3] Ratnasamy et al. (2001). “A Scalable Content-Addressable Network” (PDF). In Proceedings of ACM SIGCOMM 2001. Retrieved 2013-05-20.
- [4] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. In Communications of the ACM, February 2003.

- [5] David Cohen (October 1, 2002). “New P2P network funded by US government”. *New Scientist*. Retrieved November 10, 2013.
- [6] “MIT, Berkeley, ICSI, NYU, and Rice Launch the IRIS Project”. *Press release* (MIT). September 25, 2002. Retrieved November 10, 2013.
- [7] Guido Urdaneta, Guillaume Pierre and Maarten van Steen. A Survey of DHT Security Techniques. *ACM Computing Surveys* 43(2), January 2011.
- [8] Moni Naor and Udi Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. *Proc. SPAA*, 2003.
- [9] Gurmeet Singh Manku. *Dipsea: A Modular Distributed Hash Table*. Ph. D. Thesis (Stanford University), August 2004.
- [10] Agostino Forestiero, Emilio Leonardi, Carlo Mastroianni and Michela Meo. Self-Chord: a Bio-Inspired P2P Framework for Self-Organizing Distributed Systems. *IEEE/ACM Transactions on Networking*, 2010.
- [11] *The (Degree,Diameter) Problem for Graphs*, Maite71.upc.es, retrieved 2012-01-10
- [12] Gurmeet Singh Manku, Moni Naor, and Udi Wieder. Know thy Neighbor’s Neighbor: the Power of Lookahead in Randomized P2P Networks. *Proc. STOC*, 2004.
- [13] Ali Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables. KTH-Royal Institute of Technology, 2006.
- [14] Miguel Castro, Manuel Costa, and Antony Rowstron. Should we build Gnutella on a structured overlay?. *Computer Communication Review*, 2004.
- [15] Domenico Talia and Paolo Trunfio. Enabling Dynamic Querying over Distributed Hash Tables. *Journal of Parallel and Distributed Computing*, 2010.
- [16] Tribler wiki retrieved January 2010.
- [17] Retroshare FAQ retrieved December 2011

### 11.27.8 External links

- Distributed Hash Tables, Part 1 by Brandon Wiley.
- Distributed Hash Tables links Carles Pairot’s Page on DHT and P2P research
- [kademlia.scs.cs.nyu.edu](http://kademlia.scs.cs.nyu.edu) Archive.org snapshots of [kademlia.scs.cs.nyu.edu](http://kademlia.scs.cs.nyu.edu)
- Hazelcast: open source DHT implementation
- Eng-Keong Lua; Crowcroft, Jon; Pias, Marcelo; Sharma, Ravi; Lim, Steve. “IEEE Survey on overlay network schemes”. *CiteSeerX: 10.1.1.111.4197*: covering unstructured and structured decentralized overlay networks including DHTs (Chord, Pastry, Tapestry and others).
- Mainline DHT Measurement at Department of Computer Science, University of Helsinki, Finland.

## 11.28 Consistent hashing

**Consistent hashing** is a special kind of **hashing** such that when a hash table is resized and consistent hashing is used, only  $K/n$  keys need to be remapped on average, where  $K$  is the number of keys, and  $n$  is the number of slots. In contrast, in most traditional **hash tables**, a change in the number of array slots causes nearly all keys to be remapped.

Consistent hashing achieves some of the same goals as **Rendezvous hashing** (also called HRW Hashing). The two techniques use different algorithms, and were devised independently and contemporaneously.

### 11.28.1 History

Originally devised by Karger *et al.* at MIT for use in distributed caching, the idea has now been expanded to other areas also. An academic paper from 1997 introduced the term “consistent hashing” as a way of distributing requests among a changing population of Web servers. Each slot is then represented by a node in a distributed system. The addition (joins) and removal (leaves/failures) of nodes only requires  $K/n$  items to be re-shuffled when the number of slots/nodes change.<sup>[1]</sup>

Consistent hashing has also been used to reduce the impact of partial system failures in large Web applications as to allow for robust caches without incurring the system wide fallout of a failure.<sup>[2]</sup>

The consistent hashing concept also applies to the design of **distributed hash tables** (DHTs). DHTs use consistent hashing to partition a keyspace among a distributed set of nodes, and additionally provide an overlay network that connects nodes such that the node responsible for any key can be efficiently located.

Rendezvous hashing, designed at the same time as consistent hashing, achieves the same goals using the very different Highest Random Weight (HRW) algorithm.

### 11.28.2 Need for consistent hashing

While running collections of caching machines some limitations are experienced. A common way of load balancing  $n$  cache machines is to put object  $o$  in cache machine number  $\text{hash}(o) \bmod n$ . But this will not work if a cache machine is added or removed because  $n$  changes and every object is hashed to a new location. This can be disastrous since the originating content servers are flooded with requests from the cache machines. Hence consistent hashing is needed to avoid swamping of servers.

Consistent hashing maps objects to the same cache machine, as far as possible. It means when a cache machine is added, it takes its share of objects from all the other

cache machines and when it is removed, its objects are shared between the remaining machines.

The main idea behind the consistent hashing algorithm is to associate each cache with one or more hash value intervals where the interval boundaries are determined by calculating the hash of each cache identifier. (The hash function used to define the intervals does not have to be the same function used to hash the cached values. Only the range of the two functions need match.) If the cache is removed its interval is taken over by a cache with an adjacent interval. All the remaining caches are unchanged.

### 11.28.3 Technique

Consistent hashing is based on mapping each object to a point on the edge of a circle (or equivalently, mapping each object to a real angle). The system maps each available machine (or other storage bucket) to many pseudo-randomly distributed points on the edge of the same circle.

To find where an object should be placed, the system finds the location of that object's key on the edge of the circle; then walks around the circle until falling into the first bucket it encounters (or equivalently, the first available bucket with a higher angle). The result is that each bucket contains all the resources located between its point and the previous bucket point.

If a bucket becomes unavailable (for example because the computer it resides on is not reachable), then the angles it maps to will be removed. Requests for resources that would have mapped to each of those points now map to the next highest point. Since each bucket is associated with many pseudo-randomly distributed points, the resources that were held by that bucket will now map to many different buckets. The items that mapped to the lost bucket must be redistributed among the remaining ones, but values mapping to other buckets will still do so and do not need to be moved.

A similar process occurs when a bucket is added. By adding a bucket point, we make any resources between that and the next smaller angle map to the new bucket. These resources will no longer be associated with the previous bucket, and any value previously stored there will not be found by the selection method described above.

The portion of the keys associated with each bucket can be altered by altering the number of angles that bucket maps to.

### 11.28.4 Monotonic keys

If it is known that key values will always increase monotonically, an alternative approach to consistent hashing is possible.

### 11.28.5 Properties

David Karger et al. list several properties of consistent hashing that make it useful for distributed caching protocols on the Internet.<sup>[1]</sup>

- “spread”
- “load”
- “smoothness”
- “balance”
- “monotonicity”

### 11.28.6 Examples of use

Some known instances where consistent hashing is used are:

- Openstack's Object Storage Service Swift<sup>[3]</sup>
- Partitioning component of Amazon's storage system Dynamo<sup>[4][5]</sup>
- Data partitioning in Apache Cassandra<sup>[6]</sup>
- Data Partitioning in Voldemort<sup>[7]</sup>
- Akka's consistent hashing router<sup>[8]</sup>
- Riak, a distributed key-value database<sup>[9]</sup>
- GlusterFS, a network-attached storage file system<sup>[10]</sup>
- Skylable, an open-source distributed object-storage system<sup>[11]</sup>

### 11.28.7 References

- [1] Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D. (1997). *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*. Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing. ACM Press New York, NY, USA. pp. 654–663. doi:10.1145/258533.258660.
- [2] Karger, D.; Sherman, A.; Berkheimer, A.; Bogstad, B.; Dhanidina, R.; Iwamoto, K.; Kim, B.; Matkins, L.; Yerushalmi, Y. (1999). “Web Caching with Consistent Hashing”. *Computer Networks* **31** (11): 1203–1213. doi:10.1016/S1389-1286(99)00055-9.
- [3] <http://docs.openstack.org/developer/swift/ring.html>
- [4] DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. (2007). “Dynamo: Amazon's Highly Available Key-Value Store”. *Proceedings of the 21st ACM Symposium on Operating Systems Principles*.

- [5] <http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>
- [6] Lakshman, Avinash; Malik, Prashant (2010). "Cassandra: a decentralized structured storage system". *ACM SIGOPS Operating Systems Review*.
- [7] "Design -- Voldemort". <http://www.project-voldemort.com/>. Archived from the original on 9 February 2015. Retrieved 9 February 2015. Consistent hashing is a technique that avoids these problems, and we use it to compute the location of each key on the cluster.
- [8] Akka Routing
- [9] "Riak Concepts".
- [10] <http://www.gluster.org/2012/03/glusterfs-algorithms-distribution/>
- [11] "Skylable architecture".

### 11.28.8 External links

- Understanding Consistent hashing
- Implementations in various languages:
  - C++
  - C#
  - Erlang
  - Go
  - Java
  - PHP
  - Python
  - Ruby
  - Perl

## 11.29 Stable hashing

**Stable hashing** is a tool used to implement randomized load balancing and distributed lookup in peer-to-peer computer systems.

### 11.29.1 See also

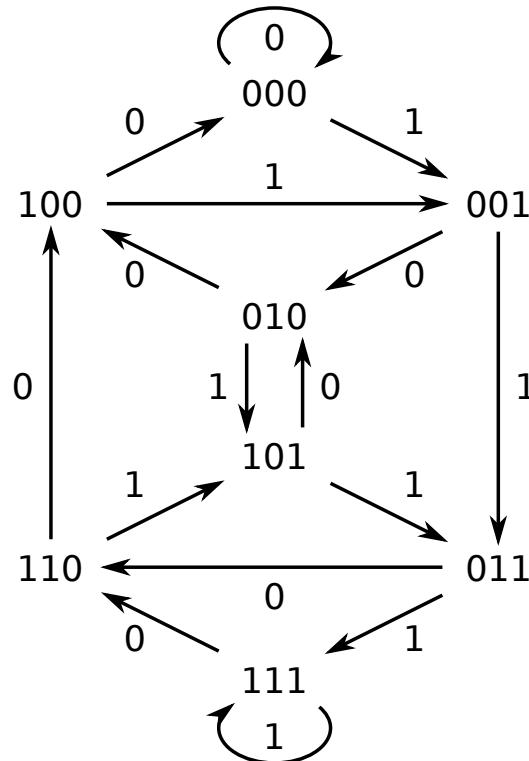
- Hash function
- Consistent hashing

## 11.30 Koorde

In peer-to-peer networks, **Koorde** is a Distributed hash table (DHT) system based on the Chord DHT and the De Bruijn graph (De Bruijn sequence). Inheriting the simplicity of Chord, Koorde meets  $O(\log n)$  hops per node (where  $n$  is the number of nodes in the DHT), and  $O(\log n / \log \log n)$  hops per lookup request with  $O(\log n)$  neighbors per node.

The Chord concept is based on a wide range of identifiers (e.g.  $2^{160}$ ) in a structure of a ring where an identifier can stand for both node and data. Node-successor is responsible for the whole range of IDs between itself and its predecessor.

### 11.30.1 De Bruijn's graphs



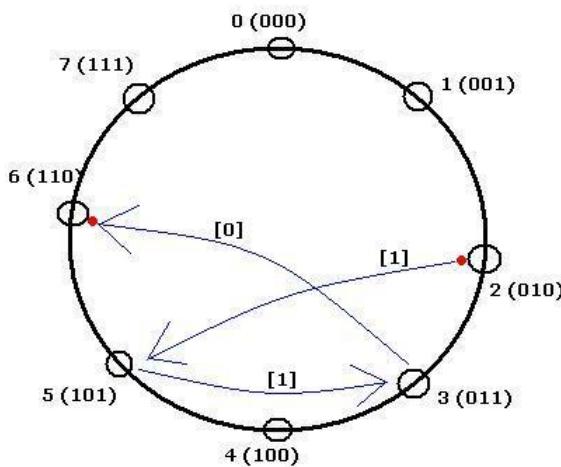
A de Bruijn's 3-dimensional graph

Koorde is based on Chord but also on De Bruijn graph (De Bruijn sequence). In a  $d$ -dimensional de Bruijn graph, there are  $2^d$  nodes, each of which has a unique  $d$ -bit ID. The node with ID  $i$  is connected to nodes  $2i$  modulo  $2^d$  and  $2i+1$  modulo  $2^d$ . Thanks to this property, the routing algorithm can route to any destination in  $d$  hops by successively "shifting in" the bits of the destination ID but only if the dimensions of the distance between modulo  $1d$  and  $3d$  are equal.

Routing a message from node  $m$  to node  $k$  is accomplished by taking the number  $m$  and shifting in the bits

of  $k$  one at a time until the number has been replaced by  $k$ . Each shift corresponds to a routing hop to the next intermediate address; the hop is valid because each node's neighbors are the two possible outcomes of shifting a 0 or 1 onto its own address. Because of the structure of de Bruijn graphs, when the last bit of  $k$  has been shifted, the query will be at node  $k$ . Node  $k$  responds whether key  $k$  exists.

### 11.30.2 Routing example



Example of the way Koorde routes from Node2 to Node6 using a 3-dimensional, binary graph.

For example, when a message needs to be routed from node “2” (which is “010”) to “6” (which is “110”), the steps are following:

Step 1) Node #2 routes the message to Node #5 (using its connection to  $2i+1 \bmod 8$ ), shifts the bits left and puts “1” as the youngest bit (right side).

Step 2) Node #5 routes the message to Node #3 (using its connection to  $2i+1 \bmod 8$ ), shifts the bits left and puts “1” as the youngest bit (right side).

Step 3) Node #3 routes the message to Node #6 (using its connection to  $2i \bmod 8$ ), shifts the bits left and puts “0” as the youngest bit (right side).

### 11.30.3 Non-constant degree Koorde

The  $d$ -dimensional de Bruijn can be generalized to base  $k$ , in which case node  $i$  is connected to nodes  $k * i + j$  modulo  $kd$ ,  $0 \leq j < k$ . The diameter is reduced to  $\Theta(\log n)$ . Koorde node  $i$  maintains pointers to  $k$  consecutive nodes beginning at the predecessor of  $k * i$  modulo  $kd$ . Each de Bruijn routing step can be emulated with an expected constant number of messages, so routing uses  $O(\log k n)$

```
function n.lookup(k, shift, i)
{
 if k ∈ {n, s} return (s);
 else if i ∈ {n, s} return
 (p.lookup(k, shift <= 1, i ∘ topBit(shift)));
 else return (s.lookup(k, shift, i));
}
```

The Koorde lookup algorithm at node  $n$ .  $k$  is the key.  $i$  is the imaginary De Bruijn node.  $p$  is the reference to the predecessor of  $2n$ .  $s$  is the reference to the successor of  $n$ .

*Koorde lookup algorithm.*

expected hops- For  $k = \Theta(\log n)$ , we get  $\Theta(\log n)$  degree and  $\Theta(\log n / \log \log n)$  diameter.

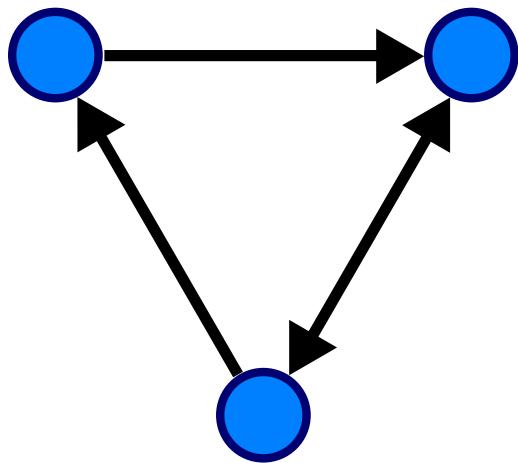
### 11.30.4 References

- “Internet Algorithms” by Greg Plaxton, Fall 2003:
- “Koorde: A simple degree-optimal distributed hash table” by M. Frans Kaashoek and David R. Karger:
- Chord and Koorde descriptions:

# Chapter 12

## Graphs

### 12.1 Graph



A graph with 3 nodes and 3 edges.

In computer science, a **graph** is an abstract data type that is meant to implement the **graph** and **directed graph** concepts from **mathematics**.

A graph data structure consists of a finite (and possibly mutable) set of **nodes** or **vertices**, together with a set of ordered pairs of these nodes (or, in some cases, a set of unordered pairs). These pairs are known as **edges** or **arcs**. As in mathematics, an edge  $(x,y)$  is said to **point** or **go from  $x$  to  $y$** . The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

#### 12.1.1 Algorithms

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like **depth-first search** and **breadth-first search** and finding the shortest path from one node to another, like **Dijkstra's**

algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the **Floyd–Warshall** algorithm.

#### 12.1.2 Operations

The basic operations provided by a graph data structure  $G$  usually include:

- $\text{adjacent}(G, x, y)$ : tests whether there is an edge from node  $x$  to node  $y$ .
- $\text{neighbors}(G, x)$ : lists all nodes  $y$  such that there is an edge from  $x$  to  $y$ .
- $\text{add}(G, x, y)$ : adds to  $G$  the edge from  $x$  to  $y$ , if it is not there.
- $\text{delete}(G, x, y)$ : removes the edge from  $x$  to  $y$ , if it is there.
- $\text{get\_node\_value}(G, x)$ : returns the value associated with the node  $x$ .
- $\text{set\_node\_value}(G, x, a)$ : sets the value associated with the node  $x$  to  $a$ .

Structures that associate values to the edges usually also provide:

- $\text{get\_edge\_value}(G, x, y)$ : returns the value associated to the edge  $(x,y)$ .
- $\text{set\_edge\_value}(G, x, y, v)$ : sets the value associated to the edge  $(x,y)$  to  $v$ .

#### 12.1.3 Representations

Different data structures for the representation of graphs are used in practice:

**Adjacency list** Vertices are stored as records or objects, and every vertex stores a **list** of adjacent vertices. This data structure allows the storage of additional data on the vertices. Additional data can be stored if

edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.

**Adjacency matrix** A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.

**Incidence matrix** A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

The following table gives the time complexity cost of performing various operations on graphs, for each of these representations. In the matrix representations, the entries encode the cost of following an edge. The cost of edges that are not present are assumed to be  $\infty$ .

Adjacency lists are generally preferred because they efficiently represent **sparse graphs**. An adjacency matrix is preferred if the graph is dense, that is the number of edges  $|E|$  is close to the number of vertices squared,  $|V|^2$ , or if one must be able to quickly look up if there is an edge connecting two vertices.<sup>[1]</sup>

#### 12.1.4 See also

- Graph traversal for graph walking strategies
- Graph database for graph (data structure) persistency
- Graph rewriting for rule based transformations of graphs (graph data structures)
- GraphStream
- Graphviz
- yEd Graph Editor – Java-based diagram editor for creating and editing graphs

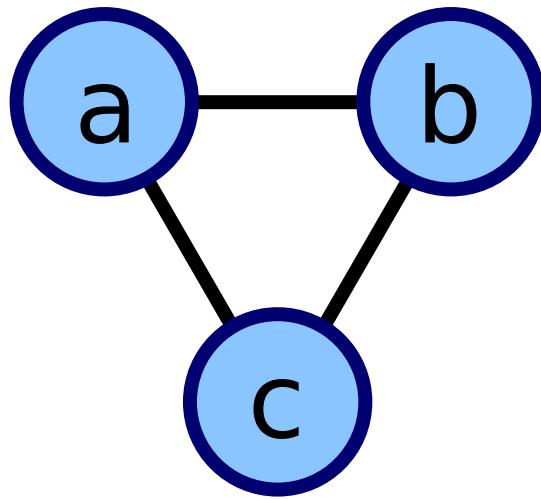
#### 12.1.5 References

- [1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). Introduction to Algorithms (2nd ed.). MIT Press and McGraw–Hill. ISBN 0-262-53196-8.

#### 12.1.6 External links

- Boost Graph Library: a powerful C++ graph library
- Networkx: a Python graph library
- Graphs Tutorial by Jebril FILALI

## 12.2 Adjacency list



*This undirected cyclic graph can be described by the list {a,b}, {a,c}, {b,c}.*

In graph theory and computer science, an **adjacency list** representation of a graph is a collection of unordered lists, one for each **vertex** in the graph. Each list describes the set of neighbors of its vertex. See "Storing a sparse matrix" for an alternative approach.

#### 12.2.1 Implementation details

An adjacency list representation for a graph associates each vertex in the graph with the collection of its neighboring vertices or edges. There are many variations of this basic idea, differing in the details of how they implement the association between vertices and collections, in how they implement the collections, in whether they include both vertices and edges or only vertices as first class objects, and in what kinds of objects are used to represent the vertices and edges.

- An implementation suggested by Guido van Rossum uses a **hash table** to associate each vertex in a graph with an **array** of adjacent vertices. In this representation, a vertex may be represented by any hashable object. There is no explicit representation of edges as objects.<sup>[1]</sup>
- Cormen et al. suggest an implementation in which the vertices are represented by index numbers.<sup>[2]</sup> Their representation uses an array indexed by vertex number, in which the array cell for each vertex points to a **singly linked list** of the neighboring vertices of that vertex. In this representation, the nodes of the singly linked list may be interpreted as edge objects; however, they do not store the full information about each edge (they only store one of the two endpoints of the edge) and in undirected graphs

there will be two different linked list nodes for each edge (one within the lists for each of the two endpoints of the edge).

- The object oriented **incidence list** structure suggested by Goodrich and Tamassia has special classes of vertex objects and edge objects. Each vertex object has an instance variable pointing to a collection object that lists the neighboring edge objects. In turn, each edge object points to the two vertex objects at its endpoints.<sup>[3]</sup> This version of the adjacency list uses more memory than the version in which adjacent vertices are listed directly, but the existence of explicit edge objects allows it extra flexibility in storing additional information about edges.

## 12.2.2 Operations

The main operation performed by the adjacency list data structure is to report a list of the neighbors of a given vertex. Using any of the implementations detailed above, this can be performed in constant time per neighbor. In other words, the total time to report all of the neighbors of a vertex  $v$  is proportional to the **degree** of  $v$ .

It is also possible, but not as efficient, to use adjacency lists to test whether an edge exists or does not exist between two specified vertices. In an adjacency list in which the neighbors of each vertex are unsorted, testing for the existence of an edge may be performed in time proportional to the degree of one of the two given vertices, by using a **sequential search** through the neighbors of this vertex. If the neighbors are represented as a sorted array, **binary search** may be used instead, taking time proportional to the logarithm of the degree.

## 12.2.3 Trade-offs

The main alternative to the adjacency list is the **adjacency matrix**, a **matrix** whose rows and columns are indexed by vertices and whose cells contain a Boolean value that indicates whether an edge is present between the vertices corresponding to the row and column of the cell. For a **sparse graph** (one in which most pairs of vertices are not connected by edges) an adjacency list is significantly more space-efficient than an adjacency matrix (stored as an array): the space usage of the adjacency list is proportional to the number of edges and vertices in the graph, while for an adjacency matrix stored in this way the space is proportional to the square of the number of vertices. However, it is possible to store adjacency matrices more space-efficiently, matching the linear space usage of an adjacency list, by using a hash table indexed by pairs of vertices rather than an array.

The other significant difference between adjacency lists and adjacency matrices is in the efficiency of the operations they perform. In an adjacency list, the neighbors

of each vertex may be listed efficiently, in time proportional to the degree of the vertex. In an adjacency matrix, this operation takes time proportional to the number of vertices in the graph, which may be significantly higher than the degree. On the other hand, the adjacency matrix allows testing whether two vertices are adjacent to each other in constant time; the adjacency list is slower to support this operation.

## 12.2.4 Data structures

For use as a data structure, the main alternative to the adjacency list is the adjacency matrix. Because each entry in the adjacency matrix requires only one bit, it can be represented in a very compact way, occupying only  $|V|^2/8$  bytes of contiguous space. Besides avoiding wasted space, this compactness encourages locality of reference.

However, for a sparse graph, adjacency lists require less storage space, because they do not waste any space to represent edges that are not present. Using a naïve array implementation on a 32-bit computer, an adjacency list for an undirected graph requires about  $8|E|$  bytes of storage.

Noting that a simple graph can have at most  $|V|^2$  edges, allowing loops, we can let  $d = |E|/|V|^2$  denote the density of the graph. Then,  $8|E| > |V|^2/8$ , or the adjacency list representation occupies more space precisely when  $d > 1/64$ . Thus a graph must be sparse indeed to justify an adjacency list representation.

Besides the space tradeoff, the different data structures also facilitate different operations. Finding all vertices adjacent to a given vertex in an adjacency list is as simple as reading the list. With an adjacency matrix, an entire row must instead be scanned, which takes  $O(|V|)$  time. Whether there is an edge between two given vertices can be determined at once with an adjacency matrix, while requiring time proportional to the minimum degree of the two vertices with the adjacency list.

## 12.2.5 References

- [1] Guido van Rossum (1998). “Python Patterns — Implementing Graphs”.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2001). *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill. pp. 527–529 of section 22.1: Representations of graphs. ISBN 0-262-03293-7.
- [3] Michael T. Goodrich and Roberto Tamassia (2002). *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons. ISBN 0-471-38365-1.

## 12.2.6 Further reading

- David Eppstein (1996). “ICS 161 Lecture Notes”.

Graph Algorithms".

### 12.2.7 External links

- The Boost Graph Library implements an efficient adjacency list
- Open Data Structures - Section 12.2 - AdjacencyList: A Graph as a Collection of Lists

## 12.3 Adjacency matrix

In mathematics, computer science and application areas such as **sociology**, an **adjacency matrix** is a means of representing which **vertices** (or **nodes**) of a **graph** are adjacent to which other vertices. Another matrix representation for a graph is the **incidence matrix**.

Specifically, the adjacency matrix of a **finite graph**  $G$  on  $n$  vertices is the  $n \times n$  matrix where the non-diagonal entry  $a_{ij}$  is the number of edges from vertex  $i$  to vertex  $j$ , and the diagonal entry  $a_{ii}$ , depending on the convention, is either once or twice the number of edges (loops) from vertex  $i$  to itself. Undirected graphs often use the latter convention of counting loops twice, whereas directed graphs typically use the former convention. There exists a unique adjacency matrix for each isomorphism class of graphs (up to permuting rows and columns), and it is not the adjacency matrix of any other isomorphism class of graphs. In the special case of a finite **simple graph**, the adjacency matrix is a **(0,1)-matrix** with zeros on its diagonal. If the graph is undirected, the adjacency matrix is **symmetric**.

The relationship between a graph and the **eigenvalues** and **eigenvectors** of its adjacency matrix is studied in **spectral graph theory**.

### 12.3.1 Examples

The convention followed here is that an adjacent edge counts 1 in the matrix for an undirected graph.

- The adjacency matrix of a **complete graph** contains all ones except along the diagonal where there are only zeros.
- The adjacency matrix of an **empty graph** is a zero matrix.

### 12.3.2 Adjacency matrix of a bipartite graph

The adjacency matrix  $A$  of a **bipartite graph** whose parts have  $r$  and  $s$  vertices has the form

$$A = \begin{pmatrix} 0_{r,r} & B \\ B^T & 0_{s,s} \end{pmatrix},$$

where  $B$  is an  $r \times s$  matrix, and  $0$  represents the zero matrix. Clearly, the matrix  $B$  uniquely represents the bipartite graphs. It is sometimes called the **biadjacency matrix**. Formally, let  $G = (U, V, E)$  be a bipartite graph with parts  $U = u_1, \dots, u_r$  and  $V = v_1, \dots, v_s$ . The **biadjacency matrix** is the  $r \times s$  0-1 matrix  $B$  in which  $b_{i,j} = 1$  iff  $(u_i, v_j) \in E$ .

If  $G$  is a bipartite **multigraph** or **weighted graph** then the elements  $b_{i,j}$  are taken to be the number of edges between the vertices or the weight of the edge  $(u_i, v_j)$ , respectively.

### 12.3.3 Properties

The adjacency matrix of an undirected simple graph is **symmetric**, and therefore has a complete set of **real eigenvalues** and an orthogonal eigenvector basis. The set of eigenvalues of a graph is the **spectrum** of the graph.

Suppose two directed or undirected graphs  $G_1$  and  $G_2$  with adjacency matrices  $A_1$  and  $A_2$  are given.  $G_1$  and  $G_2$  are **isomorphic** if and only if there exists a **permutation matrix**  $P$  such that

$$PA_1P^{-1} = A_2.$$

In particular,  $A_1$  and  $A_2$  are **similar** and therefore have the same **minimal polynomial**, **characteristic polynomial**, **eigenvalues**, **determinant** and **trace**. These can therefore serve as isomorphism invariants of graphs. However, two graphs may possess the same set of eigenvalues but not be isomorphic.<sup>[1]</sup> Such **linear operators** are said to be **isospectral**.

If  $A$  is the adjacency matrix of the directed or undirected graph  $G$ , then the matrix  $A^n$  (i.e., the **matrix product** of  $n$  copies of  $A$ ) has an interesting interpretation: the entry in row  $i$  and column  $j$  gives the number of (directed or undirected) walks of length  $n$  from vertex  $i$  to vertex  $j$ . If  $n$  is the smallest nonnegative integer, such that for all  $i, j$ , the  $(i,j)$ -entry of  $A^n > 0$ , then  $n$  is the **distance** between vertex  $i$  and vertex  $j$ . This implies, for example, that the number of triangles in an undirected graph  $G$  is exactly the trace of  $A^3$  divided by 6. Note that, the adjacent matrix can determine whether or not the graph is **connected**.

The main diagonal of every adjacency matrix corresponding to a graph without loops has all zero entries. Note that here 'loops' means, for example  $A \rightarrow A$ , not 'cycles' such as  $A \rightarrow B \rightarrow A$ .

For  $(d)$ -regular graphs,  $d$  is also an eigenvalue of  $A$  for the vector  $v = (1, \dots, 1)$ , and  $G$  is connected if and only if the multiplicity of the eigenvalue  $d$  is 1. It can

be shown that  $-d$  is also an eigenvalue of  $A$  if  $G$  is a connected bipartite graph. The above are results of the Perron–Frobenius theorem.

### 12.3.4 Variations

An  $(a, b, c)$ -adjacency matrix  $A$  of a simple graph has  $A_{ij} = a$  if  $ij$  is an edge,  $b$  if it is not, and  $c$  on the diagonal. The Seidel adjacency matrix is a **(-1,1,0)-adjacency matrix**. This matrix is used in studying strongly regular graphs and two-graphs.<sup>[2]</sup>

The **distance matrix** has in position  $(i,j)$  the distance between vertices  $v_i$  and  $v_j$ . The distance is the length of a shortest path connecting the vertices. Unless lengths of edges are explicitly provided, the length of a path is the number of edges in it. The distance matrix resembles a high power of the adjacency matrix, but instead of telling only whether or not two vertices are connected (i.e., the connection matrix, which contains boolean values), it gives the exact distance between them.

### 12.3.5 Data structures

For use as a data structure, the main alternative to the adjacency matrix is the **adjacency list**. Because each entry in the adjacency matrix requires only one bit, it can be represented in a very compact way, occupying only  $n^2/8$  bytes of contiguous space, where  $n$  is the number of vertices. Besides avoiding wasted space, this compactness encourages **locality of reference**.

However, for a **sparse graph**, adjacency lists require less storage space, because they do not waste any space to represent edges that are *not* present. Using a naïve **array** implementation on a 32-bit computer, an adjacency list for an undirected graph requires about  $8e$  bytes of storage, where  $e$  is the number of edges.

Noting that a simple graph can have at most  $n^2$  edges, allowing loops, we can let  $d = e/n^2$  denote the **density** of the graph. Then,  $8e > n^2/8$ , or the adjacency list representation occupies more space precisely when  $d > 1/64$ . Thus a graph must be sparse indeed to justify an adjacency list representation.

Besides the space tradeoff, the different data structures also facilitate different operations. Finding all vertices adjacent to a given vertex in an adjacency list is as simple as reading the list. With an adjacency matrix, an entire row must instead be scanned, which takes  $O(n)$  time. Whether there is an edge between two given vertices can be determined at once with an adjacency matrix, while requiring time proportional to the minimum degree of the two vertices with the adjacency list.

### 12.3.6 References

[1] Godsil, Chris; Royle, Gordon *Algebraic Graph Theory*,

Springer (2001), ISBN 0-387-95241-1, p.164

[2] Seidel, J. J. (1968). “Strongly Regular Graphs with  $(-1,1,0)$  Adjacency Matrix Having Eigenvalue 3”. *Lin. Alg. Appl.* 1 (2): 281–298. doi:10.1016/0024-3795(68)90008-6.

### 12.3.7 Further reading

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). “Section 22.1: Representations of graphs”. *Introduction to Algorithms* (Second ed.). MIT Press and McGraw-Hill. pp. 527–531. ISBN 0-262-03293-7.
- Godsil, Chris; Royle, Gordon (2001). *Algebraic Graph Theory*. New York: Springer. ISBN 0-387-95241-1.

### 12.3.8 External links

- Fluffschack — an educational Java web start game demonstrating the relationship between adjacency matrices and graphs.
- Open Data Structures - Section 12.1 - Adjacency-Matrix: Representing a Graph by a Matrix
- McKay, Brendan. “Description of graph6 and sparse6 encodings”.
- Café math : Adjacency Matrices of Graphs : Application of the adjacency matrices to the computation generating series of walks.

## 12.4 And-inverter graph

An **and-inverter graph (AIG)** is a directed, acyclic graph that represents a structural implementation of the logical functionality of a **circuit or network**. An AIG consists of two-input nodes representing logical conjunction, terminal nodes labeled with variable names, and edges optionally containing markers indicating logical negation. This representation of a logic function is rarely structurally efficient for large circuits, but is an efficient representation for manipulation of boolean functions. Typically, the abstract graph is represented as a **data structure** in software.

Conversion from the network of logic gates to AIGs is fast and scalable. It only requires that every gate be expressed in terms of **AND gates** and inverters. This conversion does not lead to unpredictable increase in memory use and runtime. This makes the AIG an efficient representation in comparison with either the **binary decision diagram** (BDD) or the “sum-of-product” ( $\Sigma\pi$ ) form, that is, the canonical form in Boolean algebra known as the **disjunctive normal form** (DNF). The BDD and DNF may

also be viewed as circuits, but they involve formal constraints that deprive them of scalability. For example,  $\Sigma\Omega\Pi$ s are circuits with at most two levels while BDDs are canonical, that is, they require that input variables be evaluated in the same order on all paths.

Circuits composed of simple gates, including AIGs, are an “ancient” research topic. The interest in AIGs started in the late 1950s<sup>[1]</sup> and continued in the 1970s when various local transformations have been developed. These transformations were implemented in several logic synthesis and verification systems, such as Darringer et al.<sup>[2]</sup> and Smith et al.,<sup>[3]</sup> which reduce circuits to improve area and delay during synthesis, or to speed up **formal equivalence checking**. Several important techniques were discovered early at IBM, such as combining and reusing multi-input logic expressions and subexpressions, now known as **structural hashing**.

Recently there has been a renewed interest in AIGs as a **functional representation** for a variety of tasks in synthesis and verification. That is because representations popular in the 1990s (such as BDDs) have reached their limits of scalability in many of their applications. Another important development was the recent emergence of much more efficient **boolean satisfiability** (SAT) solvers. When coupled with **AIGs** as the circuit representation, they lead to remarkable speedups in solving a wide variety of **boolean problems**.

AIGs found successful use in diverse **EDA** applications. A well-tuned combination of **AIGs** and **boolean satisfiability** made an impact on **formal verification**, including both **model checking** and **equivalence checking**.<sup>[4]</sup> Another recent work shows that efficient circuit compression techniques can be developed using AIGs.<sup>[5]</sup> There is a growing understanding that logic and **physical synthesis** problems can be solved using AIGs simulation and **boolean satisfiability** compute functional properties (such as **symmetries**<sup>[6]</sup>) and node flexibilities (such as **don't-cares**, **resubstitutions**, and **SPFDs**<sup>[7]</sup>). This work shows that AIGs are a promising *unifying* representation, which can bridge **logic synthesis**, **technology mapping**, **physical synthesis**, and **formal verification**. This is, to a large extent, due to the simple and uniform structure of AIGs, which allow rewriting, simulation, mapping, placement, and verification to share the same data structure.

In addition to combinational logic, AIGs have also been applied to **sequential logic** and **sequential transformations**. Specifically, the method of structural hashing was extended to work for AIGs with memory elements (such as **D-type flip-flops** with an initial state, which, in general, can be unknown) resulting in a data structure that is specifically tailored for applications related to **retiming**.<sup>[8]</sup>

Ongoing research includes implementing a modern logic synthesis system completely based on AIGs. The prototype called **ABC** features an AIG package, several AIG-based synthesis and equivalence-checking techniques,

as well as an experimental implementation of sequential synthesis. One such technique combines technology mapping and retiming in a single optimization step. These optimizations can be implemented using networks composed of arbitrary gates, but the use of AIGs makes them more scalable and easier to implement.

### 12.4.1 Implementations

- Logic Synthesis and Verification System **ABC**
- A set of utilities for AIGs **AIGER**
- **OpenAccess Gear**

### 12.4.2 References

- [1] L. Hellerman (June 1963). “A catalog of three-variable Or-Inverter and And-Inverter logical circuits”. *IEEE Trans. Electron. Comput.* EC-12 (3): 198–223. doi:10.1109/PGEC.1963.263531.
- [2] A. Darringer, W. H. Joyner, Jr., C. L. Berman, L. Trevillyan (1981). “Logic synthesis through local transformations”. *IBM J. of Research and Development* 25 (4): 272–280. doi:10.1147/rd.254.0272.
- [3] G. L. Smith, R. J. Bahnsen, H. Halliwell (1982). “Boolean comparison of hardware and flowcharts”. *IBM J. of Research and Development* 26 (1): 106–116. doi:10.1147/rd.261.0106.
- [4] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai (2002). “Robust boolean reasoning for equivalence checking and functional property verification”. *IEEE Trans. CAD* 21 (12): 1377–1394.
- [5] P. Bjesse and A. Boralv. “DAG-aware circuit compression for formal verification”. *Proc. ICCAD '04*. pp. 42–49.
- [6] K.-H. Chang, I. L. Markov, V. Bertacco. “Post-placement rewiring and rebuffering by exhaustive search for functional symmetries”. *Proc. ICCAD '05* pages=56–63.
- [7] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske (May 2006). “Using simulation and satisfiability to compute flexibilities in Boolean networks”. *IEEE Trans. CAD* 25 (5): 743–755.
- [8] J. Baumgartner and A. Kuehlmann. “Min-area retiming on flexible circuit structures”. *Proc. ICCAD'01*. pp. 176–182.

### 12.4.3 See also

- **Binary decision diagram**
- **Logical conjunction**

*This article is adapted from a column in the ACM SIGDA e-newsletter by Alan Mishchenko*  
Original text is available [here](#).

## 12.5 Binary decision diagram

In computer science, a **binary decision diagram (BDD)** or **branching program**, like a negation normal form (NNF) or a propositional directed acyclic graph (PDAG), is a **data structure** that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of **sets or relations**. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression.

### 12.5.1 Definition

A Boolean function can be represented as a rooted, directed, acyclic **graph**, which consists of several decision nodes and terminal nodes. There are two types of terminal nodes called 0-terminal and 1-terminal. Each decision node  $N$  is labeled by Boolean variable  $V_N$  and has two **child nodes** called low child and high child. The edge from node  $V_N$  to a low (or high) child represents an assignment of  $V_N$  to 0 (resp. 1). Such a **BDD** is called 'ordered' if different variables appear in the same order on all paths from the root. A BDD is said to be 'reduced' if the following two rules have been applied to its graph:

- Merge any isomorphic subgraphs.
- Eliminate any node whose two children are isomorphic.

In popular usage, the term **BDD** almost always refers to **Reduced Ordered Binary Decision Diagram (ROBDD)** in the literature, used when the ordering and reduction aspects need to be emphasized). The advantage of an ROBDD is that it is canonical (unique) for a particular function and variable order.<sup>[1]</sup> This property makes it useful in **functional equivalence checking** and other operations like **functional technology mapping**.

A path from the root node to the 1-terminal represents a (possibly partial) variable assignment for which the represented Boolean function is true. As the path descends to a low (or high) child from a node, then that node's variable is assigned to 0 (resp. 1).

#### Example

The left figure below shows a binary decision *tree* (the reduction rules are not applied), and a **truth table**, each representing the function  $f(x_1, x_2, x_3)$ . In the tree on the left, the value of the function can be determined for a given variable assignment by following a path down the graph to a terminal. In the figures below, dotted lines represent edges to a low child, while solid lines represent edges to a high child. Therefore, to find  $(x_1=0, x_2=1, x_3=1)$ , begin at  $x_1$ , traverse down the dotted line to  $x_2$

(since  $x_1$  has an assignment to 0), then down two solid lines (since  $x_2$  and  $x_3$  each have an assignment to one). This leads to the terminal 1, which is the value of  $f(x_1=0, x_2=1, x_3=1)$ .

The binary decision *tree* of the left figure can be transformed into a binary decision *diagram* by maximally reducing it according to the two reduction rules. The resulting **BDD** is shown in the right figure.

### 12.5.2 History

The basic idea from which the data structure was created is the **Shannon expansion**. A switching function is split into two sub-functions (cofactors) by assigning one variable (cf. *if-then-else normal form*). If such a sub-function is considered as a sub-tree, it can be represented by a **binary decision tree**. Binary decision diagrams (BDD) were introduced by Lee,<sup>[2]</sup> and further studied and made known by Akers<sup>[3]</sup> and Boute.<sup>[4]</sup>

The full potential for efficient algorithms based on the data structure was investigated by **Randal Bryant** at **Carnegie Mellon University**: his key extensions were to use a fixed variable ordering (for canonical representation) and shared sub-graphs (for compression). Applying these two concepts results in an efficient data structure and algorithms for the representation of sets and relations.<sup>[5][6]</sup> By extending the sharing to several BDDs, i.e. one sub-graph is used by several BDDs, the data structure **Shared Reduced Ordered Binary Decision Diagram** is defined.<sup>[7]</sup> The notion of a BDD is now generally used to refer to that particular data structure.

In his video lecture *Fun With Binary Decision Diagrams (BDDs)*,<sup>[8]</sup> Donald Knuth calls BDDs "one of the only really fundamental data structures that came out in the last twenty-five years" and mentions that Bryant's 1986 paper was for some time one of the most-cited papers in computer science.

Adnan Darwiche and his collaborators have shown that BDDs are one of several normal forms for Boolean functions, each induced by a different combination of requirements. Another important normal form identified by Darwiche is **Decomposable Negation Normal Form** or DNNF.

### 12.5.3 Applications

BDDs are extensively used in **CAD** software to synthesize circuits (**logic synthesis**) and in **formal verification**. There are several lesser known applications of BDD, including **fault tree analysis**, **Bayesian reasoning**, **product configuration**, and **private information retrieval**<sup>[9][10]</sup>.

Every arbitrary BDD (even if it is not reduced or ordered) can be directly implemented in hardware by replacing each node with a 2 to 1 **multiplexer**; each multiplexer can be directly implemented by a 4-LUT in a **FPGA**. It is not

so simple to convert from an arbitrary network of logic gates to a BDD (unlike the and-inverter graph).

### 12.5.4 Variable ordering

The size of the BDD is determined both by the function being represented and the chosen ordering of the variables. There exist Boolean functions  $f(x_1, \dots, x_n)$  for which depending upon the ordering of the variables we would end up getting a graph whose number of nodes would be linear (in  $n$ ) at the best and exponential at the worst case (e.g., a ripple carry adder). Let us consider the Boolean function  $f(x_1, \dots, x_{2n}) = x_1x_2 + x_3x_4 + \dots + x_{2n-1}x_{2n}$ . Using the variable ordering  $x_1 < x_3 < \dots < x_{2n-1} < x_2 < x_4 < \dots < x_{2n}$ , the BDD needs  $2^{n+1}$  nodes to represent the function. Using the ordering  $x_1 < x_2 < x_3 < x_4 < \dots < x_{2n-1} < x_{2n}$ , the BDD consists of  $2n + 2$  nodes.

It is of crucial importance to care about variable ordering when applying this data structure in practice. The problem of finding the best variable ordering is NP-hard.<sup>[11]</sup> For any constant  $c > 1$  it is even NP-hard to compute a variable ordering resulting in an OBDD with a size that is at most  $c$  times larger than an optimal one.<sup>[12]</sup> However there exist efficient heuristics to tackle the problem.<sup>[13]</sup>

There are functions for which the graph size is always exponential — independent of variable ordering. This holds e. g. for the multiplication function (an indication as to the apparent complexity of factorization).

Researchers have of late suggested refinements on the BDD data structure giving way to a number of related graphs, such as BMD (binary moment diagrams), ZDD (zero-suppressed decision diagram), FDD (free binary decision diagrams), PDD (parity decision diagrams), and MTBDDs (multiple terminal BDDs).

### 12.5.5 Logical operations on BDDs

Many logical operations on BDDs can be implemented by polynomial-time graph manipulation algorithms.

- conjunction
- disjunction
- negation
- existential abstraction
- universal abstraction

However, repeating these operations several times, for example forming the conjunction or disjunction of a set of BDDs, may in the worst case result in an exponentially big BDD. This is because any of the preceding operations for two BDDs may result in a BDD with a size proportional to the product of the BDDs' sizes, and consequently for several BDDs the size may be exponential.

### 12.5.6 See also

- Boolean satisfiability problem
- L/poly, a complexity class that captures the complexity of problems with polynomially sized BDDs
- Model checking
- Radix tree
- Binary key – a method of species identification in biology using binary trees
- Barrington's theorem

### 12.5.7 References

- [1] Graph-Based Algorithms for Boolean Function Manipulation, Randal E. Bryant, 1986
- [2] C. Y. Lee. “Representation of Switching Circuits by Binary-Decision Programs”. Bell Systems Technical Journal, 38:985–999, 1959.
- [3] Sheldon B. Akers. Binary Decision Diagrams, IEEE Transactions on Computers, C-27(6):509–516, June 1978.
- [4] Raymond T. Boute, “The Binary Decision Machine as a programmable controller”. EUROMICRO Newsletter, Vol. 1(2):16–22, January 1976.
- [5] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". IEEE Transactions on Computers, C-35(8):677–691, 1986.
- [6] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", ACM Computing Surveys, Vol. 24, No. 3 (September, 1992), pp. 293–318.
- [7] Karl S. Brace, Richard L. Rudell and Randal E. Bryant. "Efficient Implementation of a BDD Package". In Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC 1990), pages 40–45. IEEE Computer Society Press, 1990.
- [8] <http://scpd.stanford.edu/knuth/index.jsp>
- [9] R.M. Jensen. “CLab: A C++ library for fast backtrack-free interactive product configuration”. Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming, 2004.
- [10] H.L. Lipmaa. “First CPIR Protocol with Data-Dependent Computation”. ICISC 2009.
- [11] Beate Bollig, Ingo Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete, IEEE Transactions on Computers, 45(9):993–1002, September 1996.
- [12] Detlef Sieling. “The nonapproximability of OBDD minimization.” Information and Computation 172, 103–138. 2002.
- [13] Rice, Michael. “A Survey of Static Variable Ordering Heuristics for Efficient BDD/MDD Construction” (PDF).

- R. Ubar, "Test Generation for Digital Circuits Using Alternative Graphs (in Russian)", in Proc. Tallinn Technical University, 1976, No.409, Tallinn Technical University, Tallinn, Estonia, pp. 75–81.

### 12.5.8 Further reading

- D. E. Knuth, "The Art of Computer Programming Volume 4, Fascicle 1: Bitwise tricks & techniques; Binary Decision Diagrams" (Addison–Wesley Professional, March 27, 2009) viii+260pp, ISBN 0-321-58050-8. Draft of Fascicle 1b available for download.
- H. R. Andersen "An Introduction to Binary Decision Diagrams," Lecture Notes, 1999, IT University of Copenhagen.
- Ch. Meinel, T. Theobald, "Algorithms and Data Structures in VLSI-Design: OBDD – Foundations and Applications", Springer-Verlag, Berlin, Heidelberg, New York, 1998. Complete textbook available for download.
- Rüdiger Ebendt; Görschwin Fey; Rolf Drechsler (2005). *Advanced BDD optimization*. Springer. ISBN 978-0-387-25453-1.
- Bernd Becker; Rolf Drechsler (1998). *Binary Decision Diagrams: Theory and Implementation*. Springer. ISBN 978-1-4419-5047-5.

### 12.5.9 External links

- Fun With Binary Decision Diagrams (BDDs), lecture by Donald Knuth
- List of BDD software libraries for several programming languages.

## 12.6 Binary moment diagram

A **binary moment diagram (BMD)** is a generalization of the **binary decision diagram (BDD)** to linear functions over domains such as booleans (like BDDs), but also to integers or to real numbers.

They can deal with boolean functions with complexity comparable to BDDs, but also some functions that are dealt with very inefficiently in a BDD are handled easily by BMD, most notably **multiplication**.

The most important properties of BMD is that, like with BDDs, each function has exactly one canonical representation, and many operations can be efficiently performed on these representations.

The main features that differentiate BMDs from BDDs are using linear instead of pointwise diagrams, and having weighted edges.

The rules that ensure the canonicity of the representation are:

- Decision over variables higher in the ordering may only point to decisions over variables lower in the ordering.
- No two nodes may be identical (in normalization such nodes all references to one of these nodes should be replaced by references to another)
- No node may have all decision parts equivalent to 0 (links to such nodes should be replaced by links to their always part)
- No edge may have weight zero (all such edges should be replaced by direct links to 0)
- Weights of the edges should be **coprime**. Without this rule or some equivalent of it, it would be possible for a function to have many representations, for example  $2x + 2$  could be represented as  $2 \cdot (1 + x)$  or  $1 \cdot (2 + 2x)$ .

### 12.6.1 Pointwise and linear decomposition

In pointwise decomposition, like in BDDs, on each branch point we store result of all branches separately. An example of such decomposition for an integer function  $(2x + y)$  is:

$$\begin{cases} \text{if } x \begin{cases} \text{if } y, 3 \\ \text{if } \neg y, 2 \end{cases} \\ \text{if } \neg x \begin{cases} \text{if } y, 1 \\ \text{if } \neg y, 0 \end{cases} \end{cases}$$

In linear decomposition we provide instead a default value and a difference:

$$\begin{cases} \text{always} \begin{cases} \text{always0} \\ \text{if } y, +1 \end{cases} \\ \text{if } x, +2 \end{cases}$$

It can easily be seen that the latter (linear) representation is much more efficient in case of additive functions, as when we add many elements the latter representation will have only  $O(n)$  elements, while the former (pointwise), even with sharing, exponentially many.

### 12.6.2 Edge weights

Another extension is using weights for edges. A value of function at given node is a sum of the true nodes below

it (the node under always, and possibly the decided node) times the edges' weights.

For example  $(4x_2 + 2x_1 + x_0)(4y_2 + 2y_1 + y_0)$  can be represented as:

1. Result node, always  $1 \times$  value of node 2, if  $x_2$  add  $4 \times$  value of node 4
2. Always  $1 \times$  value of node 3, if  $x_1$  add  $2 \times$  value of node 4
3. Always 0, if  $x_0$  add  $1 \times$  value of node 4
4. Always  $1 \times$  value of node 5, if  $y_2$  add +4
5. Always  $1 \times$  value of node 6, if  $y_1$  add +2
6. Always 0, if  $y_0$  add +1

Without weighted nodes a much more complex representation would be required:

1. Result node, always value of node 2, if  $x_2$  value of node 4
2. Always value of node 3, if  $x_1$  value of node 7
3. Always 0, if  $x_0$  value of node 10
4. Always value of node 5, if  $y_2$  add +16
5. Always value of node 6, if  $y_1$  add +8
6. Always 0, if  $y_0$  add +4
7. Always value of node 8, if  $y_2$  add +8
8. Always value of node 9, if  $y_1$  add +4
9. Always 0, if  $y_0$  add +2
10. Always value of node 11, if  $y_2$  add +4
11. Always value of node 12, if  $y_1$  add +2
12. Always 0, if  $y_0$  add +1

### 12.6.3 References

- 

## 12.7 Zero-suppressed decision diagram

A **zero-suppressed decision diagram** (ZSDD or ZDD) is a type of **binary decision diagram** (BDD) where instead of nodes being introduced when the positive and the negative part are different, they are introduced when positive part is different from constant 0. A **zero-suppressed**

decision diagram is also commonly referred to as a **zero-suppressed binary decision diagram** (ZBDD).

They are useful when dealing with functions that are almost everywhere 0.

In a 2011 talk “All Questions Answered”,<sup>[1]</sup> Donald Knuth referred to ZDD as the most beautiful construct in computer science.

In *The Art of Computer Programming*, volume 4, Knuth introduces his **Simpath algorithm** for constructing a ZDD representing all simple paths between two vertices in a graph.

### 12.7.1 Available packages

- **CUDD**: A BDD package written in C that implements BDDs and ZBDDs, University of Colorado, Boulder
- **JDD**, A java library that implements common BDD and ZBDD operations
- **Graphillion**, A ZDD software implementation based on Python
- , A CWEB ZDD implementation by Donald Knuth.

### 12.7.2 References

[1] “All Questions Answered” by Donald Knuth. *YouTube.com*. Retrieved 12 June 2013.

- Shin-ichi Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems", DAC '93: Proceedings of the 30th international conference on Design automation, 1993
- Ch. Meinel, T. Theobald, "Algorithms and Data Structures in VLSI-Design: OBDD – Foundations and Applications", Springer-Verlag, Berlin, Heidelberg, New York, 1998.

### 12.7.3 External links

- Alan Mishchenko, **An Introduction to Zero-Suppressed Binary Decision Diagrams**
- Donald Knuth, **Fun With Zero-Suppressed Binary Decision Diagrams (ZDDs)** (video lecture, 2008)
- Minato Shin-ichi, **Counting paths in graphs (fundamentals of ZDD)** (video illustration produced on Miraikan)

## 12.8 Propositional directed acyclic graph

A **propositional directed acyclic graph (PDAG)** is a data structure that is used to represent a **Boolean function**. A Boolean function can be represented as a rooted, directed acyclic graph of the following form:

- Leaves are labeled with  $\top$  (true),  $\perp$  (false), or a Boolean variable.
- Non-leaves are  $\triangle$  (logical and),  $\triangledown$  (logical or) and  $\diamond$  (logical not).
- $\triangle$  - and  $\triangledown$  -nodes have at least one child.
- $\diamond$  -nodes have exactly one child.

Leaves labeled with  $\top$  ( $\perp$ ) represent the constant Boolean function which always evaluates to 1 (0). A leaf labeled with a Boolean variable  $x$  is interpreted as the assignment  $x = 1$ , i.e. it represents the Boolean function which evaluates to 1 if and only if  $x = 1$ . The Boolean function represented by a  $\triangle$  -node is the one that evaluates to 1, if and only if the Boolean function of all its children evaluate to 1. Similarly, a  $\triangledown$  -node represents the Boolean function that evaluates to 1, if and only if the Boolean function of at least one child evaluates to 1. Finally, a  $\diamond$  -node represents the complementary Boolean function its child, i.e. the one that evaluates to 1, if and only if the Boolean function of its child evaluates to 0.

### 12.8.1 PDAG, BDD, and NNF

Every **binary decision diagram (BDD)** and every **negation normal form (NNF)** are also a PDAG with some particular properties. The following pictures represent the Boolean function  $f(x_1, x_2, x_3) = -x_1 * -x_2 * -x_3 + x_1 * x_2 + x_2 * x_3$ :

### 12.8.2 See also

- Data structure
- Boolean satisfiability problem
- Proposition

### 12.8.3 References

- M. Wachter & R. Haenni, “Propositional DAGs: a New Graph-Based Language for Representing Boolean Functions”, KR’06, 10th International Conference on Principles of Knowledge Representation and Reasoning, Lake District, UK, 2006.

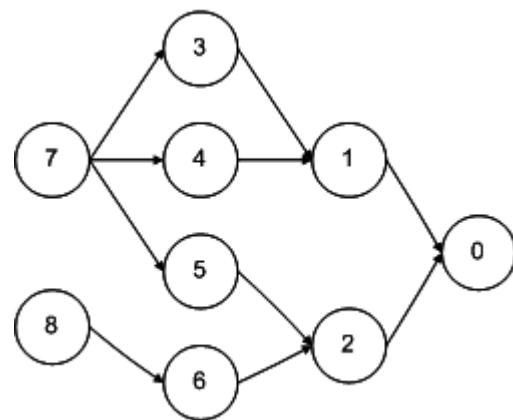
- M. Wachter & R. Haenni, “Probabilistic Equivalence Checking with Propositional DAGs”, Technical Report iam-2006-001, Institute of Computer Science and Applied Mathematics, University of Bern, Switzerland, 2006.

- M. Wachter, R. Haenni & J. Jonczy, “Reliability and Diagnostics of Modular Systems: a New Probabilistic Approach”, DX’06, 18th International Workshop on Principles of Diagnosis, Peñaranda de Duero, Burgos, Spain, 2006.

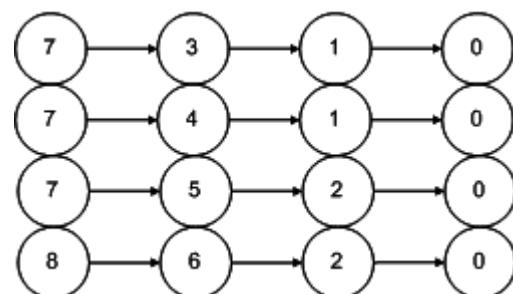
## 12.9 Graph-structured stack

In computer science, a **graph-structured stack** is a directed acyclic graph where each directed path represents a **stack**. The graph-structured stack is an essential part of Tomita’s algorithm, where it replaces the usual stack of a pushdown automaton. This allows the algorithm to encode the nondeterministic choices in parsing an **ambiguous grammar**, sometimes with greater efficiency.

In the following diagram, there are four stacks: {7,3,1,0}, {7,4,1,0}, {7,5,2,0}, and {8,6,2,0}.



Another way to simulate nondeterminism would be to duplicate the stack as needed. The duplication would be less efficient since vertices would not be shared. For this example, 16 vertices would be needed instead of 9.



### 12.9.1 References

- Masaru Tomita. *Graph-Structured Stack And Natural Language Parsing*. Annual Meeting of the Association of Computational Linguistics, 1988.

## 12.10 Scene graph

This article is about general data structure for vector-based graphics. For the software component used in user interfaces, see [Canvas \(GUI\)](#).

A **scene graph** is a general data structure commonly used by [vector-based graphics editing](#) applications and modern computer games, which arranges the logical and often (but not necessarily) spatial representation of a graphical scene. Examples of such programs include [Acrobat 3D](#), [Adobe Illustrator](#), [AutoCAD](#), [CorelDRAW](#), [OpenSceneGraph](#), [OpenSG](#), [VRML97](#), [X3D](#), [Hoops](#) and [Open Inventor](#).

A scene graph is a collection of nodes in a [graph](#) or [tree](#) structure. A tree node (in the overall tree structure of the scene graph) may have many children but often only a single parent, with the effect of a parent applied to all its child nodes; an operation performed on a group automatically propagates its effect to all of its members. In many programs, associating a geometrical [transformation matrix](#) (see also [transformation](#) and [matrix](#)) at each group level and concatenating such matrices together is an efficient and natural way to process such operations. A common feature, for instance, is the ability to group related shapes/objects into a compound object that can then be moved, transformed, selected, etc. as easily as a single object.

It also happens that in some scene graphs, a node can have a relation to any node including itself, or at least an extension that refers to another node (for instance [Pixar's PhotoRealistic RenderMan](#) because of its usage of [Reyes](#) rendering algorithm, or [Adobe Systems's Acrobat 3D](#) for advanced interactive manipulation).

### 12.10.1 Scene graphs in graphics editing tools

In vector-based graphics editing, each [leaf](#) node in a scene graph represents some atomic unit of the document, usually a shape such as an [ellipse](#) or [Bezier path](#). Although shapes themselves (particularly paths) can be decomposed further into nodes such as [spline nodes](#), it is practical to think of the scene graph as composed of shapes rather than going to a lower level of representation.

Another useful and user-driven node concept is the [layer](#). A layer acts like a transparent sheet upon which any num-

ber of shapes and shape groups can be placed. The document then becomes a set of layers, any of which can be conveniently made invisible, dimmed, or locked (made read-only). Some applications place all layers in a linear list, while others support sublayers (i.e., layers within layers to any desired depth).

Internally, there may be no real structural difference between layers and groups at all, since they are both just nodes of a scene graph. If differences are needed, a common type declaration in [C++](#) would be to make a generic node class, and then derive layers and groups as subclasses. A visibility member, for example, would be a feature of a layer, but not necessarily of a group.

### 12.10.2 Scene graphs in games and 3D applications

Scene graphs are useful for modern games using [3D](#) graphics and increasingly large worlds or levels. In such applications, nodes in a scene graph (generally) represent entities or objects in the scene.

For instance, a game might define a logical relationship between a knight and a horse so that the knight is considered an extension to the horse. The scene graph would have a 'horse' node with a 'knight' node attached to it.

As well as describing the logical relationship, the scene graph may also describe the spatial relationship of the various entities: the knight moves through 3D space as the horse moves.

In these large applications, memory requirements are major considerations when designing a scene graph. For this reason, many large scene graph systems use instancing to reduce memory costs and increase speed. In our example above, each knight is a separate scene node, but the graphical representation of the knight (made up of a 3D mesh, textures, materials and shaders) is instanced. This means that only a single copy of the data is kept, which is then referenced by any 'knight' nodes in the scene graph. This allows a reduced memory budget and increased speed, since when a new knight node is created, the appearance data does not need to be duplicated.

### 12.10.3 Scene graph implementation

The simplest form of scene graph uses an array or [linked list](#) [data structure](#), and displaying its shapes is simply a matter of linearly iterating the nodes one by one. Other common operations, such as checking to see which shape intersects the mouse pointer (e.g., in a [GUI](#)-based applications) are also done via linear searches. For small scene graphs, this tends to suffice.

Larger scene graphs cause linear operations to become noticeably slow and thus more complex underlying data structures are used, the most popular and common form

being a **tree**. In these scene graphs, the **composite design pattern** is often employed to create the hierarchical representation of group nodes and leaf nodes.

**Group nodes** — Can have any number of child nodes attached to it. Group nodes include transformations and switch nodes.

**Leaf nodes** — Are nodes that are actually **rendered** or see the effect of an operation. These include objects, sprites, sounds, lights and anything that could be considered 'rendered' in some abstract sense.

### Scene graph operations and dispatch

Applying an operation on a scene graph requires some way of dispatching an operation based on a node's type. For example, in a render operation, a transformation group node would accumulate its transformation by matrix multiplication, vector displacement, **quaternions** or **Euler angles**. After which a leaf node sends the object off for rendering to the renderer. Some implementations might render the object directly, which invokes the underlying rendering **API**, such as **DirectX** or **OpenGL**. But since the underlying implementation of the rendering API usually lacks portability, one might separate the scene graph and rendering systems instead. In order to accomplish this type of dispatching, several different approaches can be taken.

In object-oriented languages such as **C++**, this can easily be achieved by **virtual functions**, where each represents an operation that can be performed on a node. Virtual functions are simple to write, but it is usually impossible to add new operations to nodes without access to the source code. Alternatively, the **visitor pattern** can be used. This has a similar disadvantage in that it is similarly difficult to add new node types.

Other techniques involve the use of **RTTI** (Run-Time Type Information). The operation can be realised as a class that is passed to the current node; it then queries the node's type using RTTI and looks up the correct operation in an array of **callbacks** or **functors**. This requires that the map of types to callbacks or functors be initialized at run-time, but offers more flexibility, speed and extensibility.

Variations on these techniques exist, and new methods can offer added benefits. One alternative is scene graph rebuilding, where the scene graph is rebuilt for each of the operations performed. This, however, can be very slow, but produces a highly optimised scene graph. It demonstrates that a good scene graph implementation depends heavily on the application in which it is used.

**Traversals** Traversals are the key to the power of applying operations to scene graphs. A traversal generally consists of starting at some arbitrary node (often the root of the scene graph), applying the operation(s) (often the updating and rendering operations are applied one after

the other), and recursively moving down the scene graph (tree) to the child nodes, until a leaf node is reached. At this point, many scene graph engines then traverse back up the tree, applying a similar operation. For example, consider a render operation that takes transformations into account: while recursively traversing down the scene graph hierarchy, a pre-render operation is called. If the node is a transformation node, it adds its own transformation to the current transformation matrix. Once the operation finishes traversing all the children of a node, it calls the node's post-render operation so that the transformation node can undo the transformation. This approach drastically reduces the necessary amount of matrix multiplication.

Some scene graph operations are actually more efficient when nodes are traversed in a different order — this is where some systems implement scene graph rebuilding to reorder the scene graph into an easier-to-parse format or tree.

For example, in 2D cases, scene graphs typically render themselves by starting at the tree's root node and then recursively draw the child nodes. The tree's leaves represent the most foreground objects. Since drawing proceeds from back to front with closer objects simply overwriting farther ones, the process is known as employing the **Painter's algorithm**. In 3D systems, which often employ **depth buffers**, it is more efficient to draw the closest objects first, since farther objects often need only be depth-tested instead of actually rendered, because they are occluded by nearer objects.

### 12.10.4 Scene graphs and bounding volume hierarchies (BVHs)

**Bounding Volume Hierarchies** (BVHs) are useful for numerous tasks — including efficient culling and speeding up collision detection between objects. A BVH is a spatial structure, but doesn't have to partition the geometry (see **spatial partitioning** below).

A BVH is a tree of bounding volumes (often spheres, axis-aligned **bounding boxes** or oriented bounding boxes). At the bottom of the hierarchy, the size of the volume is just large enough to encompass a single object tightly (or possibly even some smaller fraction of an object in high resolution BVHs). As one ascends the hierarchy, each node has its own volume that tightly encompasses all the volumes beneath it. At the root of the tree is a volume that encompasses all the volumes in the tree (the whole scene).

BVHs are useful for speeding up collision detection between objects. If an object's bounding volume does not intersect a volume higher in the tree, it cannot intersect any object below that node (so they are all rejected very quickly).

There are some similarities between BVHs and scene

graphs. A scene graph can easily be adapted to include/become a BVH — if each node has a volume associated or there is a purpose-built “bound node” added in at convenient location in the hierarchy. This may not be the typical view of a scene graph, but there are benefits to including a BVH in a scene graph.

### 12.10.5 Scene graphs and spatial partitioning

An effective way of combining spatial partitioning and scene graphs is by creating a scene leaf node that contains the spatial partitioning data. This can increase computational efficiency of rendering.

Spatial data is usually static and generally contains non-moving level data in some partitioned form. Some systems may have the systems and their rendering separately. This is fine and there are no real advantages to either method. In particular, it is bad to have the scene graph contained within the spatial partitioning system, as the scene graph is better thought of as the grander system to the spatial partitioning.

Very large drawings, or scene graphs that are generated solely at runtime (as happens in ray tracing rendering programs), require defining of group nodes in a more automated fashion. A raytracer, for example, will take a scene description of a 3D model and build an internal representation that breaks up its individual parts into bounding boxes (also called bounding slabs). These boxes are grouped hierarchically so that ray intersection tests (as part of visibility determination) can be efficiently computed. A group box that does not intersect an eye ray, for example, can entirely skip testing any of its members.

A similar efficiency holds in 2D applications as well. If the user has magnified a document so that only part of it is visible on his computer screen, and then scrolls in it, it is useful to use a bounding box (or in this case, a bounding rectangle scheme) to quickly determine which scene graph elements are visible and thus actually need to be drawn.

Depending on the particulars of the application’s drawing performance, a large part of the scene graph’s design can be impacted by rendering efficiency considerations. In 3D video games such as *Quake*, for example, binary space partitioning (BSP) trees are heavily favored to minimize visibility tests. BSP trees, however, take a very long time to compute from design scene graphs, and must be recomputed if the design scene graph changes, so the levels tend to remain static, and dynamic characters aren’t generally considered in the spatial partitioning scheme.

Scene graphs for dense regular objects such as heightfields and polygon meshes tend to employ quadtrees and octrees, which are specialized variants of a 3D bounding box hierarchy. Since a heightfield occupies a box volume itself, recursively subdividing this box into

eight subboxes (hence the ‘oct’ in octree) until individual heightfield elements are reached is efficient and natural. A quadtree is simply a 2D octree.

### 12.10.6 Standards

#### PHIGS

PHIGS was the first commercial scene graph specification, and became an ANSI standard in 1988. Disparate implementations were provided by Unix hardware vendors. The HOOPS 3D Graphics System appears to have been the first commercial scene graph library provided by a single software vendor. It was designed to run on disparate lower-level 2D and 3D interfaces, with the first major production version (v3.0) completed in 1991. Shortly thereafter, Silicon Graphics released IRIS Inventor 1.0 (1992), which was a scene graph built on top of the IRIS GL 3D API. It was followed up with Open Inventor in 1994, a portable scene graph built on top of OpenGL. More 3D scene graph libraries can be found in Category:3D scenegraph APIs.

#### X3D

X3D is a royalty-free open-standards file format and run-time architecture to represent and communicate 3D scenes and objects using XML. It is an ISO-ratified standard that provides a system for the storage, retrieval and playback of real-time graphics content embedded in applications, all within an open architecture to support a wide array of domains and user scenarios.

### 12.10.7 See also

- Graph (data structure)
- Graph theory
- Jreality
- Space partitioning
- Tree (data structure)

### 12.10.8 References

#### Books

- Leler, Wm and Merry, Jim (1996) *3D with HOOPS*, Addison-Wesley
- Wernecke, Josie (1994) *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, ISBN 0-201-62495-8 (Release 2)

## Articles

- Bar-Zeev, Avi. “Scenegraphs: Past, Present, and Future”
- Carey, Rikk and Bell, Gavin (1997). “The Annotated VRML 97 Reference Manual”
- James H. Clark (1976). “Hierarchical Geometric Models for Visible Surface Algorithms”. *Communications of the ACM*. 19(10): 547–554.
- Helman, Jim; Rohlff, John (1994). “IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics”
- PEXTimes
- Strauss, Paul (1993). “IRIS Inventor, a 3D Graphics Toolkit”

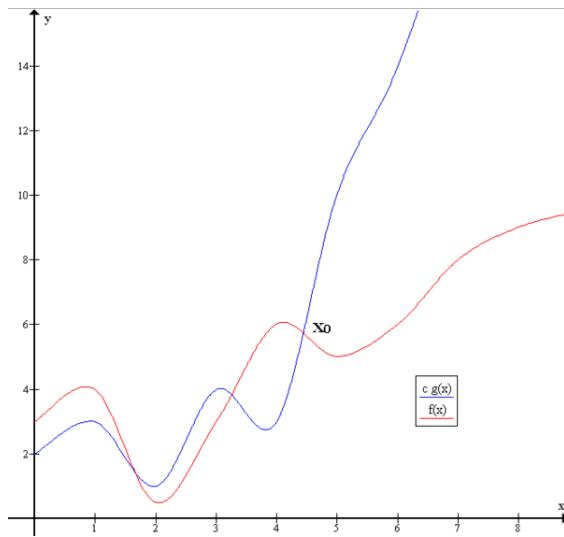
### 12.10.9 External links

- Java3D
  - Aviatrix3D
  - LG3D
- OpenSG
- OpenSceneGraph
- OSG.JS Javascript Implementation of OpenSceneGraph
- Visualization Library

# Chapter 13

## Appendix

### 13.1 Big O notation



Example of Big O notation:  $f(x) \in O(g(x))$  as there exists  $c > 0$  (e.g.,  $c = 1$ ) and  $x_0$  (e.g.,  $x_0 = 5$ ) such that  $f(x) < cg(x)$  whenever  $x > x_0$ .

In mathematics, **big O notation** describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. It is a member of a larger family of notations that is called **Landau notation**, **Bachmann-Landau notation** (after Edmund Landau and Paul Bachmann),<sup>[1][2]</sup> or **asymptotic notation**. In computer science, big O notation is used to classify algorithms<sup>[3][4]</sup> by how they respond (e.g., in their processing time or working space requirements) to changes in input size. In analytic number theory, it is used to estimate the “error committed” while replacing the asymptotic size, or asymptotic mean size, of an arithmetical function, by the value, or mean value, it takes at a large finite argument. A famous example is the problem of estimating the remainder term in the prime number theorem.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. The letter O is used because the growth rate of a function is also referred to as order of the function. A description

of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function. Associated with big O notation are several related notations, using the symbols  $o$ ,  $\Omega$ ,  $\omega$ , and  $\Theta$ , to describe other kinds of bounds on asymptotic growth rates.

Big O notation is also used in many other fields to provide similar estimates.

#### 13.1.1 Formal definition

Let  $f$  and  $g$  be two functions defined on some subset of the real numbers. One writes

$$f(x) = O(g(x)) \text{ as } x \rightarrow \infty$$

if and only if there is a positive constant  $M$  such that for all sufficiently large values of  $x$ , the absolute value of  $f(x)$  is at most  $M$  multiplied by the absolute value of  $g(x)$ . That is,  $f(x) = O(g(x))$  if and only if there exists a positive real number  $M$  and a real number  $x_0$  such that

$$|f(x)| \leq M|g(x)| \text{ all for } x \geq x_0.$$

In many contexts, the assumption that we are interested in the growth rate as the variable  $x$  goes to infinity is left unstated, and one writes more simply that  $f(x) = O(g(x))$ . The notation can also be used to describe the behavior of  $f$  near some real number  $a$  (often,  $a = 0$ ): we say

$$f(x) = O(g(x)) \text{ as } x \rightarrow a$$

if and only if there exist positive numbers  $\delta$  and  $M$  such that

$$|f(x)| \leq M|g(x)| \text{ for } |x - a| < \delta.$$

If  $g(x)$  is non-zero for values of  $x$  sufficiently close to  $a$ , both of these definitions can be unified using the limit superior:

### 13.1.3 Usage

$f(x) = O(g(x))$  as  $x \rightarrow a$

if and only if

$$\limsup_{x \rightarrow a} \left| \frac{f(x)}{g(x)} \right| < \infty.$$

#### 13.1.2 Example

In typical usage, the formal definition of  $O$  notation is not used directly; rather, the  $O$  notation for a function  $f$  is derived by the following simplification rules:

- If  $f(x)$  is a sum of several terms, the one with the largest growth rate is kept, and all others omitted.
- If  $f(x)$  is a product of several factors, any constants (terms in the product that do not depend on  $x$ ) are omitted.

For example, let  $f(x) = 6x^4 - 2x^3 + 5$ , and suppose we wish to simplify this function, using  $O$  notation, to describe its growth rate as  $x$  approaches infinity. This function is the sum of three terms:  $6x^4$ ,  $-2x^3$ , and 5. Of these three terms, the one with the highest growth rate is the one with the largest exponent as a function of  $x$ , namely  $6x^4$ . Now one may apply the second rule:  $6x^4$  is a product of 6 and  $x^4$  in which the first factor does not depend on  $x$ . Omitting this factor results in the simplified form  $x^4$ . Thus, we say that  $f(x)$  is a “big-oh” of  $(x^4)$ . Mathematically, we can write  $f(x) = O(x^4)$ . One may confirm this calculation using the formal definition: let  $f(x) = 6x^4 - 2x^3 + 5$  and  $g(x) = x^4$ . Applying the formal definition from above, the statement that  $f(x) = O(x^4)$  is equivalent to its expansion,

$$|f(x)| \leq M|g(x)|$$

for some suitable choice of  $x_0$  and  $M$  and for all  $x > x_0$ . To prove this, let  $x_0 = 1$  and  $M = 13$ . Then, for all  $x > x_0$ :

$$\begin{aligned} |6x^4 - 2x^3 + 5| &\leq 6x^4 + |2x^3| + 5 \\ &\leq 6x^4 + 2x^4 + 5x^4 \\ &= 13x^4 \end{aligned}$$

so

$$|6x^4 - 2x^3 + 5| \leq 13x^4.$$

Big O notation has two main areas of application. In mathematics, it is commonly used to describe how closely a finite series approximates a given function, especially in the case of a truncated Taylor series or asymptotic expansion. In computer science, it is useful in the analysis of algorithms. In both applications, the function  $g(x)$  appearing within the  $O(\dots)$  is typically chosen to be as simple as possible, omitting constant factors and lower order terms. There are two formally close, but noticeably different, usages of this notation: infinite asymptotics and infinitesimal asymptotics. This distinction is only in application and not in principle, however—the formal definition for the “big O” is the same for both cases, only with different limits for the function argument.

#### Infinite asymptotics

Big O notation is useful when analyzing algorithms for efficiency. For example, the time (or the number of steps) it takes to complete a problem of size  $n$  might be found to be  $T(n) = 4n^2 - 2n + 2$ . As  $n$  grows large, the  $n^2$  term will come to dominate, so that all other terms can be neglected—for instance when  $n = 500$ , the term  $4n^2$  is 1000 times as large as the  $2n$  term. Ignoring the latter would have negligible effect on the expression’s value for most purposes. Further, the coefficients become irrelevant if we compare to any other order of expression, such as an expression containing a term  $n^3$  or  $n^4$ . Even if  $T(n) = 1,000,000n^2$ , if  $U(n) = n^3$ , the latter will always exceed the former once  $n$  grows larger than 1,000,000 ( $T(1,000,000) = 1,000,000^3 = U(1,000,000)$ ). Additionally, the number of steps depends on the details of the machine model on which the algorithm runs, but different types of machines typically vary by only a constant factor in the number of steps needed to execute an algorithm. So the big O notation captures what remains: we write either

$$T(n) = O(n^2)$$

or

$$T(n) \in O(n^2)$$

and say that the algorithm has *order of  $n^2$*  time complexity. Note that “=” is not meant to express “is equal to” in its normal mathematical sense, but rather a more colloquial “is”, so the second expression is technically accurate (see the “Equals sign” discussion below) while the first is a common abuse of notation.<sup>[5]</sup>

### Infinitesimal asymptotics

Big O can also be used to describe the **error term** in an approximation to a mathematical function. The most significant terms are written explicitly, and then the least-significant terms are summarized in a single big O term. Consider, for example, the **exponential series** and two expressions of it that are valid when  $x$  is small:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots && \text{all for } x \\ &= 1 + x + \frac{x^2}{2} + O(x^3) && \text{as } x \rightarrow 0, \\ &= 1 + x + O(x^2) && \text{as } x \rightarrow 0. \end{aligned}$$

The second expression (the one with  $O(x^3)$ ) means the absolute-value of the error  $e^x - (1 + x + x^2/2)$  is smaller than some constant times  $|x^3|$  when  $x$  is close enough to 0.

#### 13.1.4 Properties

If the function  $f$  can be written as a finite sum of other functions, then the fastest growing one determines the order of  $f(n)$ . For example

$$f(n) = 9 \log n + 5(\log n)^3 + 3n^2 + 2n^3 = O(n^3), \quad \text{asn} \rightarrow \infty.$$

In particular, if a function may be bounded by a polynomial in  $n$ , then as  $n$  tends to *infinity*, one may disregard *lower-order* terms of the polynomial.  $O(n^c)$  and  $O(c^n)$  are very different. If  $c$  is greater than one, then the latter grows much faster. A function that grows faster than  $n^c$  for any  $c$  is called *superpolynomial*. One that grows more slowly than any exponential function of the form  $c^n$  is called *subexponential*. An algorithm can require time that is both superpolynomial and subexponential; examples of this include the fastest known algorithms for **integer factorization**.  $O(\log n)$  is exactly the same as  $O(\log(n^c))$ . The logarithms differ only by a constant factor (since  $\log(n^c) = c \log n$ ) and thus the big O notation ignores that. Similarly, logs with different constant bases are equivalent.

Exponentials with different bases, on the other hand, are not of the same order. For example,  $2^n$  and  $3^n$  are not of the same order. Changing units may or may not affect the order of the resulting algorithm. Changing units is equivalent to multiplying the appropriate variable by a constant wherever it appears. For example, if an algorithm runs in the order of  $n^2$ , replacing  $n$  by  $cn$  means the algorithm runs in the order of  $c^2n^2$ , and the big O notation ignores the constant  $c^2$ . This can be written as  $c^2n^2 \in O(n^2)$ . If, however, an algorithm runs in the order of  $2^n$ , replacing  $n$  with  $cn$  gives  $2^{cn} = (2^c)^n$ . This is not equivalent to  $2^n$  in general. Changing of variable may

affect the order of the resulting algorithm. For example, if an algorithm's running time is  $O(n)$  when measured in terms of the number  $n$  of *digits* of an input number  $x$ , then its running time is  $O(\log x)$  when measured as a function of the input number  $x$  itself, because  $n = \Theta(\log x)$ .

### Product

$$\begin{aligned} f_1 \in O(g_1) \text{ and } f_2 \in O(g_2) &\Rightarrow f_1 f_2 \in O(g_1 g_2) \\ f \cdot O(g) \subset O(fg) \end{aligned}$$

### Sum

$$f_1 \in O(g_1) \text{ and } f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(|g_1| + |g_2|)$$

This implies  $f_1 \in O(g)$  and  $f_2 \in O(g) \Rightarrow f_1 + f_2 \in O(g)$ , which means that  $O(g)$  is a **convex cone**.

If  $f$  and  $g$  are positive functions,  $f + O(g) \in O(f + g)$

### Multiplication by a constant

Let  $k$  be a constant. Then:

$$O(kg) = O(g) \text{ if } k \text{ is nonzero.}$$

$$f \in O(g) \Rightarrow kf \in O(g).$$

#### 13.1.5 Multiple variables

Big O (and little o, and  $\Omega$ ...) can also be used with multiple variables. To define Big O formally for multiple variables, suppose  $f(\vec{x})$  and  $g(\vec{x})$  are two functions defined on some subset of  $\mathbb{R}^n$ . We say

$f(\vec{x})$  is  $O(g(\vec{x}))$  as  $\vec{x} \rightarrow \infty$

if and only if<sup>[6]</sup>

$\exists M \exists C > 0$  all for that such  $\vec{x}$  with  $x_i \geq M$  some for  $i$ ,  $|f(\vec{x})| \leq C|g(\vec{x})|$

Equivalently, the condition that  $x_i \geq M$  for some  $i$  can be replaced with the condition that  $\|\vec{x}\| \geq M$ , where  $\|\vec{x}\|$  denotes the **Chebyshev distance**. For example, the statement

$$f(n, m) = n^2 + m^3 + O(n + m) \text{ as } n, m \rightarrow \infty$$

asserts that there exist constants  $C$  and  $M$  such that

$$\forall \|\vec{(n, m)}\| \geq M : |g(n, m)| \leq C(n + m),$$

where  $g(n,m)$  is defined by

$$f(n, m) = n^2 + m^3 + g(n, m).$$

Note that this definition allows all of the coordinates of  $\vec{x}$  to increase to infinity. In particular, the statement

$$f(n, m) = O(n^m) \text{ as } n, m \rightarrow \infty$$

(i.e.,  $\exists C \exists M \forall n \forall m \dots$ ) is quite different from

$$\forall m: f(n, m) = O(n^m) \text{ as } n \rightarrow \infty$$

(i.e.,  $\forall m \exists C \exists M \forall n \dots$ ).

This is not the only generalization of big O to multivariate functions, and in practice, there is some inconsistency in the choice of definition.<sup>[7]</sup>

### 13.1.6 Matters of notation

#### Equals sign

The statement " $f(x)$  is  $O(g(x))$ " as defined above is usually written as  $f(x) = O(g(x))$ . Some consider this to be an **abuse of notation**, since the use of the equals sign could be misleading as it suggests a symmetry that this statement does not have. As de Bruijn says,  $O(x) = O(x^2)$  is true but  $O(x^2) = O(x)$  is not.<sup>[8]</sup> Knuth describes such statements as "one-way equalities", since if the sides could be reversed, "we could deduce ridiculous things like  $n = n^2$  from the identities  $n = O(n^2)$  and  $n^2 = O(n^2)$ ".<sup>[9]</sup> For these reasons, it would be more precise to use **set notation** and write  $f(x) \in O(g(x))$ , thinking of  $O(g(x))$  as the class of all functions  $h(x)$  such that  $|h(x)| \leq Cg(x)$  for some constant  $C$ .<sup>[9]</sup> However, the use of the equals sign is customary. Knuth pointed out that "mathematicians customarily use the = sign as they use the word 'is' in English: Aristotle is a man, but a man isn't necessarily Aristotle."<sup>[10]</sup>

#### Other arithmetic operators

Big O notation can also be used in conjunction with other arithmetic operators in more complicated equations. For example,  $h(x) + O(f(x))$  denotes the collection of functions having the growth of  $h(x)$  plus a part whose growth is limited to that of  $f(x)$ . Thus,

$$g(x) = h(x) + O(f(x))$$

expresses the same as

$$g(x) - h(x) \in O(f(x)).$$

**Example** Suppose an **algorithm** is being developed to operate on a set of  $n$  elements. Its developers are interested in finding a function  $T(n)$  that will express how long the algorithm will take to run (in some arbitrary measurement of time) in terms of the number of elements in the input set. The algorithm works by first calling a subroutine to sort the elements in the set and then perform its own operations. The sort has a known time complexity of  $O(n^2)$ , and after the subroutine runs the algorithm must take an additional  $55n^3 + 2n + 10$  time before it terminates. Thus the overall time complexity of the algorithm can be expressed as

$$T(n) = 55n^3 + O(n^2).$$

Here the terms  $2n+10$  are subsumed within the faster-growing  $O(n^2)$ . Again, this usage disregards some of the formal meaning of the "=" symbol, but it does allow one to use the big O notation as a kind of convenient placeholder.

#### Declaration of variables

Another feature of the notation, although less exceptional, is that function arguments may need to be inferred from the context when several variables are involved. The following two right-hand side big O notations have dramatically different meanings:

$$f(m) = O(m^n),$$

$$g(n) = O(m^n).$$

The first case states that  $f(m)$  exhibits polynomial growth, while the second, assuming  $m > 1$ , states that  $g(n)$  exhibits exponential growth. To avoid confusion, some authors use the notation

$$g(x) \in O(f(x)).$$

rather than the less explicit

$$g \in O(f),$$

#### Multiple usages

In more complicated usage,  $O(\dots)$  can appear in different places in an equation, even several times on each side. For example, the following are true for  $n \rightarrow \infty$

$$(n+1)^2 = n^2 + O(n)$$

$$(n + O(n^{1/2}))(n + O(\log n))^2 = n^3 + O(n^{5/2})$$

$$n^{O(1)} = O(e^n).$$

The meaning of such statements is as follows: for *any* functions which satisfy each  $O(\dots)$  on the left side, there are *some* functions satisfying each  $O(\dots)$  on the right side, such that substituting all these functions into the equation makes the two sides equal. For example, the third equation above means: "For any function  $f(n) = O(1)$  , there is some function  $g(n) = O(e^n)$  such that  $n^{f(n)} = g(n)$  ." In terms of the "set notation" above, the meaning is that the class of functions represented by the left side is a subset of the class of functions represented by the right side. In this use the "=" is a formal symbol that unlike the usual use of "=" is not a **symmetric relation**. Thus for example  $n^{O(1)} = O(e^n)$  does not imply the false statement  $O(e^n) = n^{O(1)}$  .

### 13.1.7 Orders of common functions

Further information: Time complexity § Table of common time complexities

Here is a list of classes of functions that are commonly encountered when analyzing the running time of an algorithm. In each case,  $c$  is a constant and  $n$  increases without bound. The slower-growing functions are generally listed first.

The statement  $f(n) = O(n!)$  is sometimes weakened to  $f(n) = O(n^n)$  to derive simpler formulas for asymptotic complexity. For any  $k > 0$  and  $c > 0$  ,  $O(n^c(\log n)^k)$  is a subset of  $O(n^{c+\varepsilon})$  for any  $\varepsilon > 0$  , so may be considered as a polynomial with some bigger order.

### 13.1.8 Related asymptotic notations

Big  $O$  is the most commonly used asymptotic notation for comparing functions, although in many cases Big  $O$  may be replaced with Big Theta  $\Theta$  for asymptotically tighter bounds. Here, we define some related notations in terms of Big  $O$ , progressing up to the family of Bachmann-Landau notations to which Big  $O$  notation belongs.

#### Little-o notation

"Little o" redirects here. For the baseball player, see [Omar Vizquel](#).

The relation  $f(x) \in o(g(x))$  is read as "  $f(x)$  is little-o of  $g(x)$  ". Intuitively, it means that  $g(x)$  grows much faster than  $f(x)$  , or similarly, the growth of  $f(x)$  is nothing compared to that of  $g(x)$  . It assumes that  $f$  and  $g$  are both functions of one variable. Formally,  $f(n) \in o(g(n))$  as  $n \rightarrow \infty$  means that for every positive constant  $\epsilon$  there exists a constant  $N$  such that

$$|f(n)| \leq \epsilon |g(n)| \quad \text{all for } n \geq N . \quad [9]$$

Note the difference between the earlier formal definition for the big-O notation, and the present definition of little-o: while the former has to be true for *at least one* constant  $M$  the latter must hold for *every* positive constant  $\epsilon$  , however small.<sup>[5]</sup> In this way little-o notation makes a stronger statement than the corresponding big-O notation: every function that is little-o of  $g$  is also big-O of  $g$ , but not every function that is big-O of  $g$  is also little-o of  $g$  (for instance  $g$  itself is not, unless it is identically zero near  $\infty$ ).

If  $g(x)$  is nonzero, or at least becomes nonzero beyond a certain point, the relation  $f(x) = o(g(x))$  is equivalent to

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0.$$

For example,

- $2x \in o(x^2)$
- $2x^2 \notin o(x^2)$
- $1/x \in o(1)$

Little-o notation is common in mathematics but rarer in computer science. In computer science the variable (and function value) is most often a natural number. In mathematics, the variable and function values are often real numbers. The following properties can be useful:

- $o(f) + o(f) \subseteq o(f)$
- $o(f)o(g) \subseteq o(fg)$
- $o(o(f)) \subseteq o(f)$
- $o(f) \subset O(f)$  (and thus the above properties apply with most combinations of o and O).

As with big O notation, the statement "  $f(x)$  is  $o(g(x))$  " is usually written as  $f(x) = o(g(x))$  , which is a slight abuse of notation.

#### Big Omega notation

There are two very widespread and incompatible definitions of the statement

$$f(x) = \Omega(g(x)) \quad (x \rightarrow a),$$

where  $a$  is some real number,  $\infty$  , or  $-\infty$  , where  $f$  and  $g$  are real functions defined in a neighbourhood of  $a$  , and where  $g$  is positive in this neighbourhood.

The first one (chronologically) is used in **analytic number theory**, and the other one in **computational complexity theory**. When the two subjects meet, this situation is bound to generate confusion.

**The Hardy–Littlewood definition** In 1914 G.H. Hardy and J.E. Littlewood introduced the new symbol  $\Omega$ <sup>[11]</sup> which is defined as follows:

$$f(x) = \Omega(g(x)) \ (x \rightarrow \infty) \Leftrightarrow \limsup_{x \rightarrow \infty} \left| \frac{f(x)}{g(x)} \right| > 0$$

Thus  $f(x) = \Omega(g(x))$  is the negation of  $f(x) = o(g(x))$ .

In 1918 the same authors introduced the two new symbols  $\Omega_R$  and  $\Omega_L$ <sup>[12]</sup> thus defined:

$$f(x) = \Omega_R(g(x)) \ (x \rightarrow \infty) \Leftrightarrow \limsup_{x \rightarrow \infty} \frac{f(x)}{g(x)} > 0$$

$$f(x) = \Omega_L(g(x)) \ (x \rightarrow \infty) \Leftrightarrow \liminf_{x \rightarrow \infty} \frac{f(x)}{g(x)} < 0$$

Hence  $f(x) = \Omega_R(g(x))$  is the negation of  $f(x) < o(g(x))$ , and  $f(x) = \Omega_L(g(x))$  the negation of  $f(x) > o(g(x))$ .

Contrary to a later assertion of D.E. Knuth,<sup>[13]</sup> Edmund Landau did use these three symbols, with the same meanings, in 1924.<sup>[14]</sup>

These Hardy–Littlewood symbols are prototypes, which after Landau were never used again exactly thus.

$\Omega_R$  became  $\Omega_+$ , and  $\Omega_L$  became  $\Omega_-$ .

These three symbols  $\Omega, \Omega_+, \Omega_-$ , as well as  $f(x) = \Omega_{\pm}(g(x))$  (meaning that  $f(x) = \Omega_+(g(x))$  and  $f(x) = \Omega_-(g(x))$  are both satisfied), are now currently used in analytic number theory.

**Simple examples** We have

$$\sin x = \Omega(1) \ (x \rightarrow \infty),$$

and more precisely

$$\sin x = \Omega_{\pm}(1) \ (x \rightarrow \infty).$$

We have

$$\sin x + 1 = \Omega(1) \ (x \rightarrow \infty),$$

and more precisely

$$\sin x + 1 = \Omega_+(1) \ (x \rightarrow \infty);$$

however

$$\sin x + 1 \neq \Omega_-(1) \ (x \rightarrow \infty).$$

**The Knuth definition** In 1976 D.E. Knuth published a paper<sup>[13]</sup> to justify his use of the  $\Omega$ -symbol to describe a stronger property. Knuth wrote: “For all the applications I have seen so far in computer science, a stronger requirement [...] is much more appropriate”. He defined

$$f(x) = \Omega(g(x)) \Leftrightarrow g(x) = O(f(x))$$

with the comment: “Although I have changed Hardy and Littlewood’s definition of  $\Omega$ , I feel justified in doing so because their definition is by no means in wide use, and because there are other ways to say what they want to say in the comparatively rare cases when their definition applies”. However, the Hardy–Littlewood definition had been used for at least 25 years.<sup>[15]</sup>

### Family of Bachmann–Landau notations

Aside from the Big  $O$  notation, the Big Theta  $\Theta$  and Big Omega  $\Omega$  notations are the two most often used in computer science; the small omega  $\omega$  notation is occasionally used in computer science.

Aside from the Big  $O$  notation, the small  $o$ , Big Omega  $\Omega$  and  $\sim$  notations are the three most often used in number theory; the small omega  $\omega$  notation is never used in number theory.

### Use in computer science

For more details on this topic, see [Analysis of algorithms](#).

Informally, especially in computer science, the Big  $O$  notation often is permitted to be somewhat abused to describe an asymptotic tight bound where using Big Theta  $\Theta$  notation might be more factually appropriate in a given context. For example, when considering a function  $T(n) = 73n^3 + 22n^2 + 58$ , all of the following are generally acceptable, but tightnesses of bound (i.e., numbers 2 and 3 below) are usually strongly preferred over laxness of bound (i.e., number 1 below).

1.  $T(n) = O(n^{100})$ , which is identical to  $T(n) \in O(n^{100})$
2.  $T(n) = O(n^3)$ , which is identical to  $T(n) \in O(n^3)$
3.  $T(n) = \Theta(n^3)$ , which is identical to  $T(n) \in \Theta(n^3)$ .

The equivalent English statements are respectively:

1.  $T(n)$  grows asymptotically no faster than  $n^{100}$
2.  $T(n)$  grows asymptotically no faster than  $n^3$
3.  $T(n)$  grows asymptotically as fast as  $n^3$ .

So while all three statements are true, progressively more information is contained in each. In some fields, however, the Big O notation (number 2 in the lists above) would be used more commonly than the Big Theta notation (bullets number 3 in the lists above) because functions that grow more slowly are more desirable. For example, if  $T(n)$  represents the running time of a newly developed algorithm for input size  $n$ , the inventors and users of the algorithm might be more inclined to put an upper asymptotic bound on how long it will take to run without making an explicit statement about the lower asymptotic bound.

### Extensions to the Bachmann–Landau notations

Another notation sometimes used in computer science is  $\tilde{O}$  (read *soft-O*):  $f(n) = \tilde{O}(g(n))$  is shorthand for  $f(n) = O(g(n) \log^k g(n))$  for some  $k$ . Essentially, it is Big O notation, ignoring logarithmic factors because the growth-rate effects of some other super-logarithmic function indicate a growth-rate explosion for large-sized input parameters that is more important to predicting bad run-time performance than the finer-point effects contributed by the logarithmic-growth factor(s). This notation is often used to obviate the “nitpicking” within growth-rates that are stated as too tightly bounded for the matters at hand (since  $\log^k n$  is always  $o(n^\varepsilon)$  for any constant  $k$  and any  $\varepsilon > 0$ ).

Also the **L** notation, defined as

$$L_n[\alpha, c] = O\left(e^{(c+o(1))(\ln n)^\alpha(\ln \ln n)^{1-\alpha}}\right),$$

is convenient for functions that are between polynomial and exponential.

### 13.1.9 Generalizations and related usages

The generalization to functions taking values in any normed vector space is straightforward (replacing absolute values by norms), where  $f$  and  $g$  need not take their values in the same space. A generalization to functions  $g$  taking values in any topological group is also possible. The “limiting process”  $x \rightarrow x_0$  can also be generalized by introducing an arbitrary filter base, i.e. to directed nets  $f$  and  $g$ . The  $o$  notation can be used to define derivatives and differentiability in quite general spaces, and also (asymptotical) equivalence of functions,

$$f \sim g \iff (f - g) \in o(g)$$

which is an equivalence relation and a more restrictive notion than the relationship “ $f$  is  $\Theta(g)$ ” from above. (It reduces to  $\lim f/g = 1$  if  $f$  and  $g$  are positive real valued functions.) For example,  $2x$  is  $\Theta(x)$ , but  $2x - x$  is not  $o(x)$ .

### 13.1.10 History (Bachmann–Landau, Hardy, and Vinogradov notations)

The symbol  $O$  was first introduced by number theorist Paul Bachmann in 1894, in the second volume of his book *Analytische Zahlentheorie* (“analytic number theory”), the first volume of which (not yet containing big O notation) was published in 1892.<sup>[16]</sup> The number theorist Edmund Landau adopted it, and was thus inspired to introduce in 1909 the notation  $o$ ;<sup>[17]</sup> hence both are now called Landau symbols. These notations were used in applied mathematics during the 1950s for asymptotic analysis.<sup>[18]</sup> The big O was popularized in computer science by Donald Knuth, who re-introduced the related Omega and Theta notations.<sup>[13]</sup> Knuth also noted that the Omega notation had been introduced by Hardy and Littlewood<sup>[11]</sup> under a different meaning “ $\neq o$ ” (i.e. “is not an  $o$  of”), and proposed the above definition. Hardy and Littlewood’s original definition (which was also used in one paper by Landau<sup>[14]</sup>) is still used in number theory (where Knuth’s definition is never used). In fact, Landau also used in 1924, in the paper just mentioned, the symbols  $\Omega_R$  (“right”) and  $\Omega_L$  (“left”), which were introduced in 1918 by Hardy and Littlewood,<sup>[12]</sup> and which were precursors for the modern symbols  $\Omega_+$  (“is not smaller than a small  $o$  of”) and  $\Omega_-$  (“is not larger than a small  $o$  of”). Thus the Omega symbols (with their original meanings) are sometimes also referred to as “Landau symbols”.

Also, Landau never used the Big Theta and small omega symbols.

Hardy’s symbols were (in terms of the modern  $O$  notation)

$$f \preceq g \iff f \in O(g) \text{ and } f \prec g \iff f \in o(g);$$

(Hardy however never defined or used the notation  $\preccurlyeq$ , nor  $\ll$ , as it has been sometimes reported). It should also be noted that Hardy introduces the symbols  $\preceq$  and  $\prec$  (as well as some other symbols) in his 1910 tract “Orders of Infinity”, and makes use of it only in three papers (1910–1913). In his nearly 400 remaining papers and books he consistently uses the Landau symbols  $O$  and  $o$ .

Hardy’s notation is not used anymore. On the other hand, in the 1930s,<sup>[19]</sup> the Russian number theorist Ivan Matveyevich Vinogradov introduced his notation  $\ll$ , which has been increasingly used in number theory instead of the  $O$  notation. We have

$$f \ll g \iff f \in O(g),$$

and frequently both notations are used in the same paper.

The big-O originally stands for “order of” (“Ordnung”, Bachmann 1894), and is thus a roman letter. Neither Bachmann nor Landau ever call it “Omicron”. The symbol was much later on (1976) viewed by Knuth as a capital

omicron,<sup>[13]</sup> probably in reference to his definition of the symbol Omega. The digit zero should not be used.

### 13.1.11 See also

- **Asymptotic expansion:** Approximation of functions generalizing Taylor's formula
- **Asymptotically optimal:** A phrase frequently used to describe an algorithm that has an upper bound asymptotically within a constant of a lower bound for the problem
- **Big O in probability notation:**  $O_p, o_p$
- **Limit superior and limit inferior:** An explanation of some of the limit notation used in this article
- **Nachbin's theorem:** A precise method of bounding complex analytic functions so that the domain of convergence of integral transforms can be stated

### 13.1.12 References and Notes

- [1] Homayoon Beigi (9 December 2011). *Fundamentals of Speaker Recognition*. Springer. p. 777. ISBN 978-0-387-77592-0.
- [2] Mark H. Holmes (5 December 2012). *Introduction to Perturbation Methods*. Springer. pp. 4–. ISBN 978-1-4614-5477-9.
- [3] “Big O Notation (MIT Lecture)” (LECTURE). Retrieved 7 June 2014. Big O notation (with a capital letter O, not a zero), also called Landau's symbol, is a symbolism used in complexity theory
- [4] Mohr, Austin. “Quantum Computing in Complexity Theory and Theory of Computation” (PDF). p. 2. Retrieved 7 June 2014.
- [5] Thomas H. Cormen et al., 2001, *Introduction to Algorithms*, Second Edition
- [6] Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2009). *Introduction to Algorithms* (Third ed.). MIT. p. 53.
- [7] Howell, Rodney. “On Asymptotic Notation with Multiple Variables” (PDF). Retrieved 2015-04-23.
- [8] N. G. de Bruijn (1958). *Asymptotic Methods in Analysis*. Amsterdam: North-Holland. pp. 5–7. ISBN 978-0-486-64221-5.
- [9] Ronald Graham, Donald Knuth, and Oren Patashnik (1994). 0-201-55802-5 *Concrete Mathematics* (2 ed.). Reading, Massachusetts: Addison-Wesley. p. 446. ISBN 978-0-201-55802-9.
- [10] Donald Knuth (June–July 1998). “Teach Calculus with Big O” (PDF). *Notices of the American Mathematical Society* 45 (6): 687. (Unabridged version)
- [11] G. H. Hardy and J. E. Littlewood, “Some problems of Diophantine approximation”, *Acta Mathematica* 37 (1914), p. 225
- [12] G. H. Hardy and J. E. Littlewood, « Contribution to the theory of the Riemann zeta-function and the theory of the distribution of primes », *Acta Mathematica*, vol. 41, 1918.
- [13] Donald Knuth. “Big Omicron and big Omega and big Theta”, *SIGACT News*, Apr.-June 1976, 18-24.
- [14] E. Landau, “Über die Anzahl der Gitterpunkte in gewissen Bereichen. IV.” *Nachr. Gesell. Wiss. Gött. Math-phys. Kl.* 1924, 137–150.
- [15] E. C. Titchmarsh, *The Theory of the Riemann Zeta-Function* (Oxford; Clarendon Press, 1951)
- [16] Nicholas J. Higham, *Handbook of writing for the mathematical sciences*, SIAM. ISBN 0-89871-420-6, p. 25
- [17] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*, Teubner, Leipzig 1909, p.883.
- [18] Erdelyi, A. (1956). *Asymptotic Expansions*. ISBN 978-0486603186.
- [19] See for instance “A new estimate for  $G(n)$  in Waring's problem” (Russian). *Doklady Akademii Nauk SSSR* 5, No 5-6 (1934), 249-253. Translated in English in: Selected works / Ivan Matveevič Vinogradov ; prepared by the Steklov Mathematical Institute of the Academy of Sciences of the USSR on the occasion of his 90th birthday. Springer-Verlag, 1985.

### 13.1.13 Further reading

- Paul Bachmann. *Die Analytische Zahlentheorie. Zahlentheorie*. pt. 2 Leipzig: B. G. Teubner, 1894.
- Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. 2 vols. Leipzig: B. G. Teubner, 1909.
- G. H. Hardy. *Orders of Infinity: The 'Infinitäräcalcül' of Paul du Bois-Reymond*, 1910.
- Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 1.2.11: Asymptotic Representations, pp. 107–123.
- Paul Vitanyi, Lambert Meertens, Big Omega versus the wild functions, *Bull. European Association Theoret. Comput. Sci. (EATCS)* 22(1984), 14-19. Also: *ACM-SIGACT News*, 16:4(1984) 56-59.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 3.1: Asymptotic notation, pp. 41–50.

- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X. Pages 226–228 of section 7.1: Measuring complexity.
- Jeremy Avigad, Kevin Donnelly. *Formalizing O notation in Isabelle/HOL*
- Paul E. Black, “big-O notation”, in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 11 March 2005. Retrieved December 16, 2006.
- Paul E. Black, “little-o notation”, in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 December 2004. Retrieved December 16, 2006.
- Paul E. Black, “ $\Omega$ ”, in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 December 2004. Retrieved December 16, 2006.
- Paul E. Black, “ $\omega$ ”, in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 29 November 2004. Retrieved December 16, 2006.
- Paul E. Black, “ $\Theta$ ”, in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 17 December 2004. Retrieved December 16, 2006.

### 13.1.14 External links

- Growth of sequences — OEIS (Online Encyclopedia of Integer Sequences) Wiki
- Introduction to Asymptotic Notations
- Landau Symbols
- Big-O Notation – What is it good for

## 13.2 Amortized analysis

“Amortized” redirects here. For other uses, see Amortization.

In computer science, **amortized analysis** is a method for analyzing a given algorithm’s **time complexity**, or how much of a resource, especially time or memory in the context of computer programs, it takes to **execute**. Unlike other analyses that focus on an algorithm’s run time in a **worst case scenario**, amortized analysis examines how an algorithm will perform in practice or on average.<sup>[1]</sup>

While certain operations for a given algorithm may have a significant cost in resources, other operations may not be as costly. Amortized analysis considers both the costly and less costly operations together over the whole series of operations of the algorithm. This may include accounting for different types of input, length of the input, and other factors that affect its performance.<sup>[2]</sup>

### 13.2.1 History

Amortized analysis initially emerged from a method called aggregate analysis, which is now subsumed by amortized analysis. However, the technique was first formally introduced by Robert Tarjan in his 1985 paper *Amortized Computational Complexity*, which addressed the need for more useful form of analysis than the common probabilistic methods used. Amortization was initially used for very specific types of algorithms, particularly those involving **binary trees** and **union** operations. However, it is now ubiquitous and comes into play when analyzing many other algorithms as well.<sup>[2]</sup>

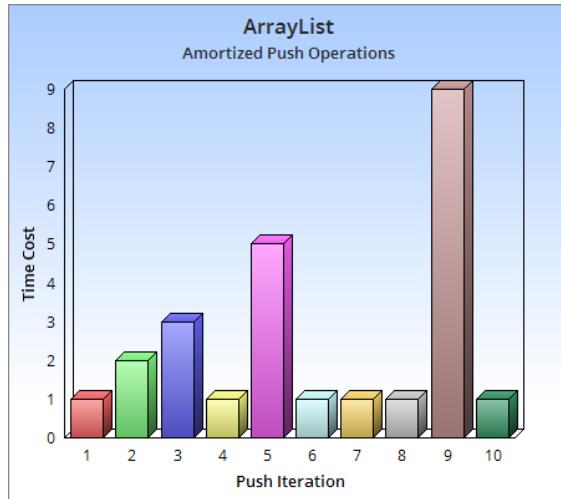
### 13.2.2 Method

The method requires knowledge of which series of operations are possible. This is most commonly the case with **data structures**, which have **state** that persists between operations. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus “amortizing” its cost.

There are generally three methods for performing amortized analysis: the aggregate method, the accounting method, and the potential method. All of these give the same answers, and their usage difference is primarily circumstantial and due to individual preference.<sup>[3]</sup>

- Aggregate analysis determines the upper bound  $T(n)$  on the total cost of a sequence of  $n$  operations, then calculates the amortized cost to be  $T(n) / n$ .<sup>[3]</sup>
- The accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a “debt” of unfavorable state in small increments, while rare long-running operations decrease it drastically.<sup>[3]</sup>
- The potential method is like the accounting method, but overcharges operations early to compensate for undercharges later.<sup>[3]</sup>

### 13.2.3 Examples



Amortized Analysis of the Push operation for a Dynamic Array

## Dynamic Array

Consider a **dynamic array** that grows in size as more elements are added to it such as an `ArrayList` in Java. If we started out with a dynamic array of size 4, it would take **constant time** to push four elements onto it. Yet pushing a fifth element onto that array would take longer as the array would have to create a new array of double the current size (8), copy the old elements onto the new array, and then add the new element. The next three push operations would similarly take constant time, and then the subsequent addition would require another slow doubling of the array size.

In general if we consider an arbitrary number of pushes  $n$  to an array of size  $n$ , we notice that push operations take constant time except for the last one which takes  $O(n)$  time to perform the size doubling operation. Since there were  $n$  operations total we can take the average of this and find that for pushing elements onto the dynamic array takes :  $O(\frac{n}{n}) = O(1)$  , constant time.<sup>[3]</sup>

## Queue

Let's look at a Ruby implementation of a Queue, a **FIFO** data structure:

```
class Queue
 def initialize
 @input = []
 @output = []
 end
 def enqueue(element)
 @input << element
 end
 def dequeue
 if @output.empty?
 while @input.any?
 @output << @input.pop
 end
 end
 @output.pop
 end
end
```

The amortized enqueue operation is always constant time because it just pushes an element onto the input array. This operation does not depend on the lengths of either input or output and so takes just constant time.

However the dequeue operation is more complicated. If the output array already has some elements in it, then it takes constant time to perform dequeue. Otherwise if

output is empty, it will take  $O(n)$  time to add all the elements onto the output array from the input array, with  $n$  being the length of the input array. Now if we have just copied  $n$  elements from input, then we can perform  $n$  dequeue operations each taking constant time before we have to perform another costly operation of copying elements from input again. Therefore in practice the amortized run time of dequeue is  $O(1)$ .<sup>[4]</sup>

### 13.2.4 Common use

- In common usage, an “amortized algorithm” is one that an amortized analysis has shown to perform well.
- **Online algorithms** commonly use amortized analysis.

### 13.2.5 References

- Allan Borodin and Ran El-Yaniv (1998). *Online Computation and Competitive Analysis*. Cambridge University Press. pp. 20,141.
- [1] “Lecture 7: Amortized Analysis” (PDF). <https://www.cs.cmu.edu/>. Retrieved 14 March 2015.
- [2] Rebecca Fiebrink (2007), *Amortized Analysis Explained* (PDF), retrieved 2011-05-03
- [3] “Lecture 20: Amortized Analysis”. <http://www.cs.cornell.edu/>. Cornell University. Retrieved 14 March 2015.
- [4] Grossman, Dan. “CSE332: Data Abstractions” (PDF). [cs.washington.edu](http://cs.washington.edu/). Retrieved 14 March 2015.

## 13.3 Locality of reference

In **computer science**, **locality of reference**, also known as the **principle of locality**, is a phenomenon describing the same value, or related **storage** locations, being frequently accessed. There are two basic types of reference locality – temporal and spatial locality. Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as, traversing the elements in a one-dimensional array.

Locality is merely one type of **predictable** behavior that occurs in computer systems. Systems that exhibit strong **locality of reference** are great candidates for performance optimization through the use of techniques such as the cache, instruction prefetch technology for memory, or the advanced branch predictor at the **pipelining** of processors.

### 13.3.1 Types of locality

There are several different types of locality of reference:

**Temporal locality** If at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future. There is a temporal proximity between the adjacent references to the same memory location. In this case it is common to make efforts to store a copy of the referenced data in special memory storage, which can be accessed faster. Temporal locality is a special case of the spatial locality, namely when the prospective location is identical to the present location.

**Spatial locality** If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In this case it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access.

**Branch locality** If there are only few amount of possible alternatives for the prospective part of the path in the *spatial-temporal coordinate space*. This is the case when an instruction loop has a simple structure, or the possible outcome of a small system of conditional branching instructions is restricted to a small set of possibilities. Branch locality is typically not a spatial locality since the few possibilities can be located far away from each other.

**Equidistant locality** It is halfway between the spatial locality and the branch locality. Consider a loop accessing locations in an equidistant pattern, i.e. the path in the *spatial-temporal coordinate space* is a dotted line. In this case, a simple linear function can predict which location will be accessed in the near future.

In order to benefit from the very frequently occurring *temporal* and *spatial* kind of locality, most of the information storage systems are *hierarchical*; see below. The *equidistant* locality is usually supported by the diverse nontrivial increment instructions of the processors. For the case of *branch* locality, the contemporary processors have sophisticated branch predictors, and on the base of this prediction the memory manager of the processor tries to collect and preprocess the data of the plausible alternatives.

### 13.3.2 Reasons for locality

There are several reasons for locality. These reasons are either goals to achieve or circumstances to accept, depending on the aspect. The reasons below are not disjoint; in fact, the list below goes from the most general case to special cases:

**Predictability** In fact, locality is merely one type of predictable behavior in computer systems. Luckily, many of the practical problems are *decidable* and hence the corresponding program can behave predictably, if it is well written.

**Structure of the program** Locality occurs often because of the way in which *computer programs* are created, for handling decidable problems. Generally, related data is stored in nearby locations in storage. One common pattern in *computing* involves the processing of several items, one at a time. This means that if a lot of processing is done, the single item will be accessed more than once, thus leading to temporal locality of reference. Furthermore, moving to the next item implies that the next item will be read, hence spatial locality of reference, since memory locations are typically read in batches.

**Linear data structures** Locality often occurs because code contains loops that tend to reference arrays or other data structures by indices. Sequential locality, a special case of spatial locality, occurs when relevant data elements are arranged and accessed linearly. For example, the simple traversal of elements in a one-dimensional array, from the base address to the highest element would exploit the sequential locality of the array in memory.<sup>[1]</sup> The more general *equidistant locality* occurs when the linear traversal is over a longer area of adjacent *data structures* with identical structure and size, accessing mutually corresponding elements of each structure rather than each entire structure. This is the case when a matrix is represented as a sequential matrix of rows and the requirement is to access a single column of the matrix.

### 13.3.3 General locality usage

If most of the time the substantial portion of the references aggregate into clusters, and if the shape of this system of clusters can be well predicted, then it can be used for speed optimization. There are several ways to benefit from locality using *optimization* techniques. Common techniques are:

**Increasing the locality of references** This is achieved usually on the software side.

**Exploiting the locality of references** This is achieved usually on the hardware side. The *temporal* and *spatial* locality can be capitalized by hierarchical storage hardwares. The *equidistant* locality can be used by the appropriately specialized instructions of the processors, this possibility is not only the responsibility of hardware, but the software as well, whether its structure is suitable for compiling a binary program that calls the specialized instructions in ques-

tion. The *branch* locality is a more elaborate possibility, hence more developing effort is needed, but there is much larger reserve for future exploration in this kind of locality than in all the remaining ones.

### 13.3.4 Spatial and temporal locality usage

#### Hierarchical memory

Main article: [Memory hierarchy](#)

Hierarchical memory is a hardware optimization that takes the benefits of spatial and temporal locality and can be used on several levels of the memory hierarchy. *Paging* obviously benefits from *temporal and spatial locality*. A cache is a simple example of exploiting temporal locality, because it is a specially designed faster but smaller memory area, generally used to keep recently referenced data and data near recently referenced data, which can lead to potential performance increases.

Data in a cache does not necessarily correspond to data that is spatially close in the main memory; however, data elements are brought into cache one *cache line* at a time. This means that spatial locality is again important: if one element is referenced, a few neighboring elements will also be brought into cache. Finally, temporal locality plays a role on the lowest level, since results that are referenced very closely together can be kept in the *machine registers*. Programming languages such as C allow the programmer to suggest that certain variables be kept in registers.

Data locality is a typical memory reference feature of regular programs (though many irregular memory access patterns exist). It makes the hierarchical memory layout profitable. In computers, memory is divided up into a hierarchy in order to speed up data accesses. The lower levels of the memory hierarchy tend to be slower, but larger. Thus, a program will achieve greater performance if it uses memory while it is cached in the upper levels of the memory hierarchy and avoids bringing other data into the upper levels of the hierarchy that will displace data that will be used shortly in the future. This is an ideal, and sometimes cannot be achieved.

Typical memory hierarchy (access times and cache sizes are approximations of typical values used as of 2013 for the purpose of discussion; actual values and actual numbers of levels in the hierarchy vary):

- **CPU registers** (8-256 registers) – immediate access, with the speed of the inner most core of the processor
- **L1 CPU caches** (32 KiB to 512 KiB) – fast access, with the speed of the inner most memory bus owned exclusively by each core

- **L2 CPU caches** (128 KiB to 24 MiB) – slightly slower access, with the speed of the memory bus shared between twins of cores
- **L3 CPU caches** (2 MiB to 32 MiB) – even slower access, with the speed of the memory bus shared between even more cores of the same processor
- **Main physical memory (RAM)** (256 MiB to 64 GiB) – slow access, the speed of which is limited by the spatial distances and general hardware interfaces between the processor and the memory modules on the motherboard
- **Disk (virtual memory, file system)** (1 GiB to 256 TiB) – very slow, due to the narrower (in bit width), physically much longer data channel between the main board of the computer and the disk devices, and due to the extraneous software protocol needed on the top of the slow hardware interface
- **Remote Memory** (such as other computers or the Internet) (Practically unlimited) – speed varies from very slow to extremely slow

Modern machines tend to read blocks of lower memory into the next level of the memory hierarchy. If this displaces used memory, the operating system tries to predict which data will be accessed least (or latest) and move it down the memory hierarchy. Prediction algorithms tend to be simple to reduce hardware complexity, though they are becoming somewhat more complicated.

#### Matrix multiplication

A common example is matrix multiplication:

```
for i in 0..n for j in 0..m for k in 0..p C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

By switching the looping order for j and k, the speedup in large matrix multiplications becomes dramatic, at least for languages that put contiguous array elements in the last dimension. This will not change the mathematical result, but it improves efficiency. In this case, “large” means, approximately, more than 100,000 elements in each matrix, or enough addressable memory such that the matrices will not fit in L1 and L2 caches.

```
for i in 0..n for k in 0..p for j in 0..m C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

The reason for this speed up is that in the first case, the reads of A[i][k] are in cache (since the k index is the contiguous, last dimension), but B[k][j] is not, so there is a cache miss penalty on B[k][j]. C[i][j] is irrelevant, because it can be factored out of the inner loop. In the second case, the reads and writes of C[i][j] are both in cache, the reads of B[k][j] are in cache, and the read of A[i][k]

can be factored out of the inner loop. Thus, the second example has no cache miss penalty in the inner loop while the first example has a cache penalty.

On a year 2014 processor, the second case is approximately five times faster than the first case, when written in C and compiled with `gcc -O3`. (A careful examination of the disassembled code shows that in the first case, `gcc` uses SIMD instructions and in the second case it does not, but the cache penalty is much worse than the SIMD gain).

Temporal locality can also be improved in the above example by using a technique called *blocking*. The larger matrix can be divided into evenly-sized sub-matrices, so that the smaller blocks can be referenced (multiplied) several times while in memory.

```
for (ii = 0; ii < SIZE; ii += BLOCK_SIZE) for (kk = 0; kk < SIZE; kk += BLOCK_SIZE) for (jj = 0; jj < SIZE; jj += BLOCK_SIZE) for (i = ii; i < ii + BLOCK_SIZE && i < SIZE; i++) for (k = kk; k < kk + BLOCK_SIZE && k < SIZE; k++) for (j = jj; j < jj + BLOCK_SIZE && j < SIZE; j++) C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

The temporal locality of the above solution is provided because a block can be used several times before moving on, so that it is moved in and out of memory less often. Spatial locality is improved because elements with consecutive memory addresses tend to be pulled up the memory hierarchy together.

### 13.3.5 See also

- Burst mode (computing)
- Cache-oblivious algorithm
- File system fragmentation
- Partitioned global address space
- Row-major order
- Scalable locality

### 13.3.6 Bibliography

- P.J. Denning, The Locality Principle, Communications of the ACM, Volume 48, Issue 7, (2005), Pages 19–24
- P.J. Denning, S.C. Schwartz, Properties of the working-set model, Communications of the ACM, Volume 15, Issue 3 (March 1972), Pages 191–198

### 13.3.7 References

- [1] Aho, Lam, Sethi, and Ullman. “Compilers: Principles, Techniques & Tools” 2nd ed. Pearson Education, Inc. 2007

## 13.4 Standard Template Library

The **Standard Template Library (STL)** is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called *algorithms*, *containers*, *functional*, and *iterators*.<sup>[1]</sup>

The STL provides a ready-made set of common classes for C++, such as containers and *associative arrays*, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.

The STL achieves its results through the use of templates. This approach provides *compile-time polymorphism* that is often more efficient than traditional *run-time polymorphism*. Modern C++ compilers are tuned to minimize any abstraction penalty arising from heavy use of the STL.

The STL was created as the first library of generic algorithms and data structures for C++, with four ideas in mind: *generic programming*, *abstractness without loss of efficiency*, the *Von Neumann computation model*,<sup>[2]</sup> and *value semantics*.

### 13.4.1 Composition

#### Containers

The STL contains sequence containers and associative containers. The standard *sequence containers* include *vector*, *deque*, and *list*. The standard *associative containers* are *set*, *multiset*, *map*, *multimap*, *hash\_set*, *hash\_map*, *hash\_multiset* and *hash\_multimap*. There are also *container adaptors* *queue*, *priority\_queue*, and *stack*, that are containers with specific interface, using other containers as implementation.

#### Iterators

The STL implements five different types of *iterators*. These are *input iterators* (that can only be used to read a sequence of values), *output iterators* (that can only be used to write a sequence of values), *forward iterators* (that can be read, written to, and move forward), *bidirectional iterators* (that are like forward iterators, but can also move backwards) and *random access iterators* (that can move freely any number of steps in one operation).

It is possible to have bidirectional iterators act like random access iterators, as moving forward ten steps could be done by simply moving forward a step at a time a total of ten times. However, having distinct random access iterators offers efficiency advantages. For example, a *vector* would have a random access iterator, but a *list* only a

bidirectional iterator.

Iterators are the major feature that allow the generality of the STL. For example, an algorithm to reverse a sequence can be implemented using bidirectional iterators, and then the same implementation can be used on lists, vectors and **deques**. User-created containers only have to provide an iterator that implements one of the five standard iterator interfaces, and all the algorithms provided in the STL can be used on the container.

This generality also comes at a price at times. For example, performing a search on an associative container such as a map or set can be much slower using iterators than by calling member functions offered by the container itself. This is because an associative container's methods can take advantage of knowledge of the internal structure, which is opaque to algorithms using iterators.

## Algorithms

A large number of algorithms to perform activities such as searching and sorting are provided in the STL, each implemented to require a certain level of iterator (and therefore will work on any container that provides an interface by iterators). Searching algorithms like `binary_search` and `lower_bound` use **binary search** and like sorting algorithms require that the type of data must implement comparison operator `<` or custom comparator function must be specified; such comparison operator or comparator function must guarantee **strict weak ordering**. Apart from these, algorithms are provided for making heap from a range of elements, generating lexicographically ordered permutations of a range of elements, merge sorted ranges and perform union, intersection, difference of sorted ranges.

## Functors

The STL includes classes that **overload** the function call operator (`operator()`). Instances of such classes are called **functors** or **function objects**. Functors allow the behavior of the associated function to be parameterized (e.g. through arguments passed to the functor's `constructor`) and can be used to keep associated per-functor state information along with the function. Since both functors and function pointers can be invoked using the syntax of a function call, they are interchangeable as arguments to templates when the corresponding parameter only appears in function call contexts.

A particularly common type of functor is the **predicate**. For example, algorithms like `find_if` take a **unary predicate** that operates on the elements of a sequence. Algorithms like `sort`, `partial_sort`, `nth_element` and all sorted containers use a **binary predicate** that must provide a **strict weak ordering**, that is, it must behave like a membership test on a transitive, non reflexive and asymmetric binary relation. If none is supplied, these algorithms and con-

ainers use `less` by default, which in turn calls the less-than-operator `<`.

### 13.4.2 History

The architecture of STL is largely the creation of **Alexander Stepanov**. In 1979 he began working out his initial ideas of **generic programming** and exploring their potential for revolutionizing software development. Although **David Musser** had developed and advocated some aspects of generic programming already by year 1971, it was limited to a rather specialized area of software development (**computer algebra**).

Stepanov recognized the full potential for generic programming and persuaded his then-colleagues at **General Electric Research and Development** (including, primarily, **David Musser** and **Deepak Kapur**) that generic programming should be pursued as a comprehensive basis for software development. At the time there was no real support in any programming language for generic programming.

The first major language to provide such support was **Ada** (**ANSI** standard 1983), with its generic units feature. In 1985, the **Eiffel** programming language became the first object-oriented language to include intrinsic support for generic classes, combined with the object-oriented notion of inheritance.<sup>[3]</sup> By 1987 Stepanov and Musser had developed and published an Ada library for list processing that embodied the results of much of their research on generic programming. However, Ada had not achieved much acceptance outside the **defense industry** and C++ seemed more likely to become widely used and provide good support for generic programming even though the language was relatively immature. Another reason for turning to C++, which Stepanov recognized early on, was the C/C++ model of computation that allows very flexible access to storage via pointers, which is crucial to achieving generality without losing efficiency.

Much research and experimentation were needed, not just to develop individual components, but to develop an overall architecture for a component library based on generic programming. First at **AT&T Bell Laboratories** and later at **Hewlett-Packard Research Labs** (HP), Stepanov experimented with many architectural and algorithm formulations, first in **C** and later in **C++**. Musser collaborated in this research and in 1992 **Meng Lee** joined Stepanov's project at HP and became a major contributor.

This work undoubtedly would have continued for some time being just a research project or at best would have resulted in an HP proprietary library, if **Andrew Koenig** of Bell Labs had not become aware of the work and asked Stepanov to present the main ideas at a November 1993 meeting of the **ANSI/ISO** committee for C++ standardization. The committee's response was overwhelmingly favorable and led to a request from Koenig for a formal

proposal in time for the March 1994 meeting. Despite the tremendous time pressure, Alex and Meng were able to produce a draft proposal that received preliminary approval at that meeting.

The committee had several requests for changes and extensions (some of them major), and a small group of committee members met with Stepanov and Lee to help work out the details. The requirements for the most significant extension (**associative containers**) had to be shown to be consistent by fully implementing them, a task Stepanov delegated to Musser. Stepanov and Lee produced a proposal that received final approval at the July 1994 ANSI/ISO committee meeting. (Additional details of this history can be found in Stevens.) Subsequently, the Stepanov and Lee document 17 was incorporated into the ANSI/ISO C++ draft standard (1, parts of clauses 17 through 27). It also influenced other parts of the C++ Standard Library, such as the string facilities, and some of the previously adopted standards in those areas were revised accordingly.

In spite of STL's success with the committee, there remained the question of how STL would make its way into actual availability and use. With the STL requirements part of the publicly available draft standard, compiler vendors and independent software library vendors could of course develop their own implementations and market them as separate products or as selling points for their other wares. One of the first edition's authors, Atul Saini, was among the first to recognize the commercial potential and began exploring it as a line of business for his company, Modena Software Incorporated, even before STL had been fully accepted by the committee.

The prospects for early widespread dissemination of STL were considerably improved with Hewlett-Packard's decision to make its implementation freely available on the **Internet** in August 1994. This implementation, developed by Stepanov, Lee, and Musser during the standardization process, became the basis of many implementations offered by compiler and library vendors today.

### 13.4.3 Criticisms

#### Quality of implementation of C++ compilers

The Quality of Implementation (QoI) of the C++ compiler has a large impact on usability of STL (and templated code in general):

- Error messages involving templates tend to be very long and difficult to decipher. This problem has been considered so severe that a number of tools have been written that simplify and **prettyprint** STL-related error messages to make them more comprehensible.
- Careless use of STL templates can lead to code bloat. This has been countered with special tech-

niques within STL implementation (using `void*` containers internally) and by improving optimization techniques used by compilers. This is similar to carelessly just copying a whole set of C library functions to work with a different type.

- Template instantiation tends to increase compilation time and memory usage (even by an order of magnitude). Until the compiler technology improves enough, this problem can be only partially eliminated by very careful coding and avoiding certain idioms.

#### Other issues

- Initialisation of STL containers with constants within the source code is not as easy as data structures inherited from C (addressed in **C++11** with **initializer lists**).
- STL containers are not intended to be used as base classes (their destructors are deliberately non-virtual); deriving from a container is a common mistake.<sup>[2][4]</sup>
- The concept of iterators as implemented by STL can be difficult to understand at first: for example, if a value pointed to by the iterator is deleted, the iterator itself is then no longer valid. This is a common source of errors. Most implementations of the STL provide a debug mode that is slower, but can locate such errors if used. A similar problem exists in other languages, for example **Java**. Ranges have been proposed as a safer, more flexible alternative to iterators.<sup>[5]</sup>
- Certain iteration patterns do not map to the STL iterator model. For example, callback enumeration APIs cannot be made to fit the STL model without the use of **coroutines**,<sup>[6]</sup> which are platform-dependent or unavailable, and are outside the C++ standard.
- Compiler compliance does not guarantee that **Allocator** objects, used for memory management for containers, will work with state-dependent behavior. For example, a portable library can't define an allocator type that will pull memory from different pools using different allocator objects of that type. (Meyers, p. 50) (addressed in **C++11**).
- The set of algorithms is not complete: for example, the `copy_if` algorithm was left out,<sup>[7]</sup> though it has been added in **C++11**.<sup>[8]</sup>
- Hashing containers were left out of the original standard (but added in Technical Report 1 for **C++03**), and have been added in **C++11**.

### 13.4.4 Implementations

- Original STL implementation by Stepanov and Lee. 1994, Hewlett-Packard. No longer maintained.
- SGI STL, based on original implementation by Stepanov & Lee. 1997, Silicon Graphics. No longer maintained.
- libstdc++ by the GNU Project (was part of libg++)<sup>[9]</sup>
- libc++ from LLVM
- STLPort, based on SGI STL
- Rogue Wave Standard Library (HP, SGI, SunSoft, Siemens-Nixdorf)
- Dinkum STL library by P.J. Plauger
- The Microsoft STL which ships with Visual C++ is a licensed derivative of Dinkum's STL.
- Apache C++ Standard Library (The starting point for this library was the 2005 version of the Rogue Wave standard library<sup>[10]</sup>)
- EASTL, developed by Paul Pedriana at Electronic Arts and published as part of EA Open Source.

### 13.4.5 See also

- List of C++ template libraries
- C++11
- Boost C++ Libraries

### 13.4.6 Notes

- [1] Holzner, Steven (2001). *C++ : Black Book*. Scottsdale, Ariz.: Coriolis Group. p. 648. ISBN 1-57610-777-9. The STL is made up of *containers*, *iterators*, *function objects*, and *algorithms*
- [2] Musser, David (2001). *STL tutorial and reference guide: C++ programming with the standard template library*. Addison Wesley. ISBN 0-201-37923-6.
- [3] Meyer, Bertrand. *Genericity versus inheritance*, in ACM Conference on Object-Oriented Programming Languages Systems and Applications (OOPSLA), Portland (Oregon), September 29 - October 2, 1986, pages 391-405.
- [4] Sutter, Herb; Alexandrescu, Andrei (2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley. ISBN 0-321-11358-6.
- [5] Andrei Alexandrescu (6 May 2009). “Iterators Must Go” (PDF). BoostCon 2009. Retrieved 19 March 2011.
- [6] Matthew Wilson (February 2004). “Callback Enumeration APIs & the Input Iterator Concept”. *Dr. Dobb's Journal*.

[7] Bjarne Stroustrup (2000). *The C++ Programming Language* (3rd ed.). Addison-Wesley. ISBN 0-201-70073-5.<sup>[530]</sup>

[8] More STL algorithms (revision 2)

[9] “libstdc++ Homepage”.

[10] Apache C++ Standard Library. Stdcxx.apache.org. Retrieved on 2013-07-29.

### 13.4.7 References

- Alexander Stepanov and Meng Lee, The Standard Template Library. HP Laboratories Technical Report 95-11(R.1), 14 November 1995. (Revised version of A. A. Stepanov and M. Lee: The Standard Template Library, Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.)
- Alexander Stepanov (2007). *Notes on Programming* (PDF). Stepanov reflects about the design of the STL.
- Nicolai M. Josuttis (2000). *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley. ISBN 0-201-37926-0.
- Scott Meyers (2001). *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley. ISBN 0-201-74962-9.
- Al Stevens (March 1995). “Al Stevens Interviews Alex Stepanov”. *Dr. Dobb's Journal*. Retrieved 18 July 2007.
- David Vandevoorde and Nicolai M. Josuttis (2002). *C++ Templates: The Complete Guide*. Addison-Wesley Professional. ISBN 0-201-73484-2.
- Atul Saini and David R. Musser, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Foreword by Alexander Stepanov; [Copyright Modena Software Inc.] Addison-Wesley ISBN 0-201-63398-1

### 13.4.8 External links

- C++ reference
- C/C++ STL reference, includes C++11 features
- STL programmer's guide from SGI
- Apache (formerly Rogue Wave) C++ Standard Library Class Reference
- Apache (formerly Rogue Wave) C++ Standard Library User Guide
- Bjarne Stroustrup on The emergence of the STL (Page 5, Section 3.1)

# Chapter 14

## Text and image sources, contributors, and licenses

### 14.1 Text

- **Data structure** *Source:* <http://en.wikipedia.org/wiki/Data%20structure?oldid=656771742> *Contributors:* LC-enwiki, Ap, -- April, Andre Engels, Karl E. V. Palmen, XJaM, Arvindn, Ghyll-enwiki, Michael Hardy, TakuyaMurata, Minesweeper, Ahoerstemeier, Nanshu, Kingturtle, Glenn, UserGoogol, Jiang, Edaelon, Nikola Smolenski, Dcoetzee, Chris Lundberg, Populus, Traroth, Mrjeff, Bearcat, Robot, Naldoaran, Craig Stuntz, Altenmann, Babbage, Mushroom, Seth Ilys, GreatWhiteNortherner, Tobias Bergemann, Giftlite, DavidCary, Esap, Jorge Stolfi, Siroxo, Pgdn002, Kjetil r, Lancekt, Jacob grace, Pale blue dot, Andreas Kaufmann, Corti, Wrp103, MisterSheik, Lycurgus, Shanes, Viriditas, Vortexrealm, Obradovic Goran, Helix84, Mdd, Jumbuck, Alansohn, Liao, Tablizer, Yaml, PaePae, ReyBrujo, Derbeth, Forderud, Mahanga, Bushytails, Mindmatrix, Carlette, Ruud Koot, Easyas12c, TreveX, Bluemoose, Abd, Palica, Mandarax, Yoric-enwiki, Qwertyus, Koavf, Ligulem, GeorgeBills, Margosbot-enwiki, Fragglet, RexNL, Fresheneesz, Butros, Chobot, Tas50, Banaticus, YurikBot, RobotE, Hairy Dude, Piet Delport, Mipadi, Grafen, Dmoss, Tony1, Googl, Ripper234, Closedmouth, Vicarious, JLaTondre, GrinBot-enwiki, TuukkaH, SmackBot, Reedy, DCDuring, Thunderboltz, BurntSky, Gilliam, Ohnoitsjamie, EncMstr, MalafayaBot, Nick Levine, Frap, Allan McInnes, Khukri, Ryan Roos, Sethwoodworth, Er Komandante, SashatoBot, Demicx, Soumyasch, Antonielly, SpyMagician, Loadmaster, Noah Salzman, Mr Stephen, Alhoori, Sharcho, Caiaffa, Iridescent, CRGreathouse, Ahy1, FinalMinuet, Requestion, Nnp, Peterdjones, GPhilip, Pascal.Tesson, Qwyrxian, MTA-enwiki, Thadius856, AntiVandalBot, Wifedox, Seaphoto, Jirka6, Dougher, Tom 99, Lanov, MER-C, Wikilolo, Wmbolle, Rhawn, Nyq, Wwmbes, David Eppstein, User A1, Cpl Syx, Oicumayberight, Gwern, MasterRadius, Rettetast, Lithui, Sanjay742, Rrwright, Marcin Suwalczan, Jimmytharpe, Santhy, TXiKiBoT, Eve Hall, Vipinhar, Coldfire82, DragonLord, Rozmichelle, Falcon8765, Spinningspark, Spitfire8520, HaiViet-enwiki, SieBot, Caltas, Eurooppa-enwiki, Ham Pastrami, Jerryobject, Strife911, Ramkumaran7, Nskillen, DancingPhilosopher, Digisus, Tanvir Ahmmmed, ClueBot, Spellsinger180, Justin W Smith, The Thing That Should Not Be, Rodhullandemu, Sundar sando, Garyzx, Adrianwn, Abhishek.kumar.ak, Excirial, Alexbot, Erebus Morgaine, Arjayay, Morel, DumZiBoT, XLinkBot, Paushali, Pgallert, Galziger, Alexius08, MystBot, Dsimic, Jncraton, MrOllie, EconoPhysicist, Publichealthguru, Tide rolls, مانی, Teles, Legobot, Luckas-bot, Yobot, Fraggie81, AnomieBOT, SteelPangolin, Jim1138, Kingpin13, Materialscientist, ArthurBot, Xqbot, Pur3r4ngelw, Miym, DAndC, RibotBOT, Shadowjams, Methcub, Prari, FrescoBot, Liridon, Mark Renier, Hypersonic12, Rameshngbot, MertyWiki, Thompsonb24, Profvalente, FoxBot, Laurențiu Dascălu, Lotje, Bharatshettybarkur, Tbhotch, Thinktdub, Kh31311, Vineetzone, Uriah123, DRAGON BOOSTER, EmusaBot, Apoctyliptic, Thecheesykid, ZéroBot, MithrandirAgain, EdEColbert, IGeMiNix, Mentibot, BioPupil, Chandraguptamaurya, Rocketrod1960, Raveendra Lakpriya, Petrb, ClueBot NG, Aks1521, Widr, Danim, Jorgenev, Orzechowskid, Gmharhar, HMSSolent, Wbm1058, Walk&check, Panchobook, Richfaber, SoniyaR, Yashykt, Cncmaster, Sallupandit, Доктор прагматик, Sgord512, Anderson, Vishnu0919, Varma rockzz, Frosty, Hernan mvs, Forgot to put name, I am One of Many, Bereajan, Gauravxpress, Haeynzen, JaconaFrere, Gambhir.jagmeet, Richard Yin, Jrahiele, Guturu Bhuvanamitra, TranquilHope, Iliasm and Anonymous: 350
- **Linked data structure** *Source:* <http://en.wikipedia.org/wiki/Linked%20data%20structure?oldid=646300833> *Contributors:* Jorge Stolfi, Andreas Kaufmann, BD2412, Strait, Katieh5584, Cybercobra, Jeffrey.Rodriguez, Headbomb, R'n'B, Spartlow, Funandtrvl, Flyer22, SchreiberBike, ChrisHodgesUK, Bunnyhop11, Miym, Timtempleton, Gmiller4th, Frze, IshitaMundada, Anuradha tupsundare, Khazar2, Vanamonde93, Ginsuloft and Anonymous: 19
- **Succinct data structure** *Source:* <http://en.wikipedia.org/wiki/Succinct%20data%20structure?oldid=642627327> *Contributors:* Tobias Bergemann, Andreas Kaufmann, Ryk, SmackBot, Nbarth, Sciyoshi-enwiki, Shalom Yechiel, Camilo Sanchez, Smhanov, Magioladitis, David Eppstein, JyBy, Addbot, Citation bot, Citation bot 1, Kmanmike15, Algotime, ZéroBot, Helpful Pixie Bot, Glacialfox, Whym, Monkbot, NovaDenizen, Mondaychen and Anonymous: 9
- **Implicit data structure** *Source:* <http://en.wikipedia.org/wiki/Implicit%20data%20structure?oldid=635422488> *Contributors:* Dcoetzee, Andreas Kaufmann, Runtime, Ruud Koot, Schizobullet, KHenriksson, Nbarth, Ilyathemuromets, Emilkeyder, Alabot, Headbomb, Lf-stevens, RainbowCrane, Roxy the dog, Dekart, Azylber, AnomieBOT, Sinryow, Jesse V., ClueBot NG, This lousy T-shirt, Primergrey, Rangilo Gujarati, BG19bot, Shrutiika girme, Borseashwini, Jochen Burghardt, Jalifej, Federico Cappella, Accentvoiceguy, Yezohtz2, Ramyasrisan, Anupam Anand Diwakar AAD, Bernardo ranchy, Alxm3, Michael.robertson121, Nikhil bhise wiki, SwagDBoss, Robholding, Assasinchan122, Langstonha, KRITSTHECOOLGUY, TwoToes02, Jessica Stemmer, Deviouscrowd1 and Anonymous: 4
- **Compressed data structure** *Source:* <http://en.wikipedia.org/wiki/Compressed%20data%20structure?oldid=570642049> *Contributors:* Andreas Kaufmann, Eekster, Algotime and Anonymous: 1
- **Search data structure** *Source:* <http://en.wikipedia.org/wiki/Search%20data%20structure?oldid=616701114> *Contributors:* GTBacchus,

Jorge Stolfi, Andreas Kaufmann, Scandum, Clayhalliwell, Vicarious, Woodshed, Quibik, Trigger hurt, Headbomb, Kathovo, Hosamaly, Magioladitis, X7q, Devendermishra, Dalton Quinn and Anonymous: 13

- **Persistent data structure** *Source:* <http://en.wikipedia.org/wiki/Persistent%20data%20structure?oldid=655370876> *Contributors:* Edward, Charles Matthews, Dcoetzee, Dfeuer, Tobias Bergemann, Haeleth, Peruvianllama, Ratan203, Spoon!, John Nowak, Hooperbloob, Ken Hirsch, Harej, Ruud Koot, Allen3, Qwertyus, MarSch, Quuxplusone, YurikBot, JiBB, Cdiggins, Cedar101, SmackBot, BiT, Chris the speller, Chris0804, Hiiiiiiiiiiiiiiiiii, Rory O'Kane, Alaibot, DivideByZero14, Headbomb, Sophie means wisdom, Usien6, David Eppstein, Pomte, Billgordon1099, Synthebot, Tomaxer, Svick, ClueBot, Scarverwiki, Addbot, Jarble, Luckas-bot, AnomieBOT, FrescoBot, Belovedeagle, Seunosewa, EmausBot, Wikipelli, Thirdreplicator, ClueBot NG, MelbourneStar, Tehom2000, BG19bot, Conifer, Jwang210, The real hugo and Anonymous: 42
  - **Concurrent data structure** *Source:* <http://en.wikipedia.org/wiki/Concurrent%20data%20structure?oldid=659571887> *Contributors:* Andreas Kaufmann, Guy Harris, Rjwilmsi, Grafen, Hervegirod, Headbomb, David Eppstein, Senthryl, Ronhjones, Citation bot, LiHelpa, Uriah123, John of Reading, Hyang04, Khizmax and Anonymous: 11
  - **Abstract data type** *Source:* <http://en.wikipedia.org/wiki/Abstract%20data%20type?oldid=659335841> *Contributors:* SolKarma, Merphant, Ark~enwiki, B4hand, Michael Hardy, Wapcaplet, Skysmith, Haakon, Silvonen, Populus, Werner, W7cook, Aqualung, BenRG, Noldoaran, Tobias Bergemann, Giftlite, WiseWoman, Jonathan.mark.lingard, Jorge Stolfi, Daniel Brockman, Knutux, Dunks58, Andreas Kaufmann, Corti, Mike Rosoft, Rich Farmbrough, Wrp103, Pink18, RJHall, Leif, Spoon!, R. S. Shaw, Alansohn, Diego Moya, Mr Adequate, Kendrick Hang, Japanese Sea robin, Miaow Miaow, Ruud Koot, Marudubshinki, Graham87, Qwertyus, Kbdank71, MZMcBride, Everton137, Chobot, YurikBot, Wavelength, SAE1962, Cedar101, Fang Aili, Sean Whitton, Petri Krohn, DGaw, Tuukkah, SmackBot, Brick Thrower, Jpivinal, Chris the speller, SchfiftyThree, Nbarts, Cybergobra, Dreadstar, A5b, Breno, Antonielli, MTSbot~enwiki, Phuzion, Only2sea, Blaisorblade, Gfnrf, Thijs!bot, Sagaciousuk, Ideogram, Widefox, JAnDbot, Magioladitis, David Eppstein, Zacciro, Felipe1982, Javawizard, SpallettaAC1041, AntiSpamBot, Cobi, Funandtrvl, Lights, Sector7agent, Anna Lincoln, Don4of4, Kbrose, Arjun024, Yerpo, Svick, Fishnet37222, Denisarona, ClueBot, The Thing That Should Not Be, Unbuttered Parsnip, Garyzx, Adrianwn, Mild Bill Hiccup, PeterV1510, Boing! said Zebedee, Armin Rigo, Cacadril, BOTarate, Thehelpfulone, Aitias, Appicharlask, Baudway, Addbot, Ghettoblaster, Capouch, Daniel.Burckhardt, Chamal N, Debresser, Bluebusy, Jarble, Yobot, Legobot II, Pcap, Vanished user rt41as76l, Materialscientist, ArthurBot, Nhey24, RibotBOT, FrescoBot, Mark Renier, Chevymontecarlo, The Arbiter, RedBot, Tanayseven, Reconsider the static, Babayagagypsies, Dismantle101, Liztanp, Efphf, Dinamik-bot, John of Reading, Thecheesykid, Ebrambot, Demonkyru, ChuispastonBot, Double Dribble, Rocketrod1960, Hoorayforturtles, Frietjes, Widr, BG19bot, Ameyenn, ChrisGualtieri, GoShow, Hower64, JYBot, Epicgenius, Carwile2, Cpt Wise and Anonymous: 170
  - **List (abstract data type)** *Source:* [http://en.wikipedia.org/wiki/List%20\(abstract%20data%20type\)?oldid=646201381](http://en.wikipedia.org/wiki/List%20(abstract%20data%20type)?oldid=646201381) *Contributors:* Mav, Jan Hidders, XJaM, Christian List, Patrick, Michael Hardy, Rp, Wwwwolf, Mic, TakuyaMurata, Delirium, Docu, Samuelsen, Angela, Glenn, Poor Yorick, Wnissen, Ed Cormany, Prumpf, Hyacinth, Stormie, Noldoaran, Fredrik, Altenmann, Peak, Puckly, Paul G, Elf, P0nc, Jorge Stolfi, Daniel Brockman, Wmahan, Neilc, Chowbok, Chevan, Calexico, Jareha, Andreas Kaufmann, EugeneZelenko, ZeroOne, Elwikipedista~enwiki, CanisRufus, Mickeymousechen~enwiki, Spoon!, Palmd, Cmdrjameson, R. S. Shaw, Liao, TShilo12, Drag~enwiki, Mindmatrix, Ruud Koot, Jeff3000, Tokek, Qwertyus, Josh Parris, Ketiltrout, Salix alba, Eubot, Paul foord, BradBeattie, Andrew Eisenberg, Roboto de Ajvol, Wavelength, RussBot, Fabartus, Gaius Cornelius, Mipadi, Dijxtra, Mike.nicholaides, SmackBot, Brick Thrower, Chris the speller, Nbarts, Cybergobra, Paddy3118, Martijn Hoekstra, HQCentral, Eao, Crater Creator, Falk Lieder, Alaibot, ManN, VictorAnyakin, PhilKnight, WODUP, Gwern, Alihaq717, BotKung, Ham Pastrami, Classicaelecon, The Thing That Should Not Be, Adrianwn, Joswig, Addbot, Denispirc, Pcap, Jeffrey Mall, Erik9bot, Redrose64, Dismantle101, WillNess, Alfredo ougaowen, Elaz85, Bomazi, ClueBot NG, Joseghr, MerflwBot, Wbm1058, Jimmisbl, MadScientistX11 and Anonymous: 70
  - **Stack (abstract data type)** *Source:* [http://en.wikipedia.org/wiki/Stack%20\(abstract%20data%20type\)?oldid=660618685](http://en.wikipedia.org/wiki/Stack%20(abstract%20data%20type)?oldid=660618685) *Contributors:* The Anome, Andre Engels, Arvindn, Christian List, Edward, Patrick, RTC, Michael Hardy, Modster, MartinHarper, Ixfd64, Takuya-Murata, Mbessey, Stw, Stan Shebs, Notheruser, Dcoetzee, Jake Nelson, Traroth, JensMueller, Finlay McWalter, Robbot, Noldoaran, Murray Langton, Fredrik, Wlievens, Guy Peters, Tobias Bergemann, Adam78, Giftlite, BenFrantzDale, WiseWoman, Gonzen, Macrakis, VampWillow, Hgfernau, Maximimax, Marc Mongenet, Karl-Henner, Andreas Kaufmann, RevRagnarok, Corti, Poccil, Andrej, CanisRufus, Spoon!, Bobo192, Grue, Shenme, R. S. Shaw, Vystrix Nexo, Physicistjedi, James Foster, Obradovic Goran, Mdd, Musiphil, Liao, Hackwrench, Pion, ReyBrujo, 2mcm, Netkinetic, Postrach, Mindmatrix, MattGiua, Ruud Koot, Mandarax, Slgrandson, Graham87, Qwertyus, Kbdank71, Angusmclellan, Maxim Razin, Vlad Patryshev, FlaBot, Dinoen, Mahlon, Chobot, Gwernol, Whosasking, NoirNoir, Roboto de Ajvol, YurikBot, Borgx, Michael Sloane, Ahluka, Stephenb, ENeville, Mipadi, Reikon, Vanished user 1029384756, Xdenizen, Scs, Epipelagic, Caerwine, Boivie, Rxwxrwx, Fragment~enwiki, Cedar101, Tuukkah, KnightRider~enwiki, SmackBot, Adam majewski, Hftf, Incnis Mrsi, BiT, Edgar181, Fernandopabon, Gilliam, Chris the speller, Agateller, RDBrown, Jprg1966, Thumperward, Oli Filth, Nbarts, DHN-bot~enwiki, Cybergobra, Funky Monkey, PeterJeremy, Mlpkr, Vasilij Faronov, MegaHasher, SashatoBot, Zchenyu, Vanished user 9i39j3, F15x28, Ultranaut, SpyMagician, CoolKoon, Loadmaster, Tasc, Mr Stephen, Iridescent, Nutster, Tsf, Jesse Viviano, IntrigueBlue, Penbat, VTBassMatt, Myasuda, FrontLine~enwiki, SimenH, Jzalae, Pheasantplucker, Bsmntbombdood, Seth Manapio, Thijs!bot, Al Lemos, Headbomb, Davidhorman, Mentifsto, AntiVandalBot, Seaphoto, Stevenbird, CosineKitty, Arch dude, IanOsgood, Jheiv, SiobhanHansa, Wikilolo, Magioladitis, VoABot II, Gamkiller, Individual X, David Eppstein, Gwern, R'n'B, Pomte, Adavidb, Ianboggs, Dillesca, Sanjay742, Bookmaker~enwiki, Cjhoyle, Manassehkat, David.Federman, Funandtrvl, Jeff G., Cheusov, TXiKiBoT, HqB, Klower, JhsBot, Aaron Rotenberg, BotKung, Wikidan829, Ideau4, SieBot, Calliopejen1, BotMultichill, Raviemani, Ham Pastrami, Keilana, Aillema, Ctxppc, OKBot, Hariva, Mr. Stradivarius, ClueBot, Clx321, Melizg, Robert impey, Mahue, Rustamabd, LobStoR, Aitias, Johnnuniq, XLinkBot, Jyotiwaroopr123321, Cericak, Hook43113, MystBot, Dsimic, Gggh, Addbot, Ghettoblaster, CanadianLinuxUser, Numbo3-bot, OIEnglish, Jarble, Aavviof, Luckas-bot, KamikazeBot, Peter Flass, AnomieBOT, 1exec1, Unara, Materialscientist, Citation bot, Xqbot, Quantran202, TechBot, GrouchoBot, RobotBOT, Cmccormick8, In fact, Rpv.imcc, Mark Renier, D'ohBot, Ionutz-movie, Alxeedo, Colin meier, Salocin-yel, Tom.Reding, Xcvista, EINuevoEinstein, Tapkeerrambo007, Trappist the monk, Tbhotch, IIT-Manojit, Yammesicka, Jfmantis, Faysol037, RjwilmsiBot, Ripchip Bot, Mohinib27, EmausBot, WikitanvirBot, Dreamkxd, Luciform, Gecg, Maashatra11, RA0808, ZéroBot, Shuipzv3, Arkahot, Paul Kube, Thine Antique Pen, BlizzmasterPilch, L Kensington, ChuispastonBot, ClueBot NG, StanBally, Dhardik007, Strcat, Ztotothefifth, Zakblade2000, Robin400, Widr, Nakarumaka, KLBot2, Spieren, Vishal G.Dhavale., Nipunbayas, PranavAmbhore, Proxyma, BattyBot, Abhidesai, Nova2358, ChrisGualtieri, Flaqueleto, Chengshuotian, Kushal-biswas777, Mogism, Makecat-bot, Bentonjimmy, Icarot, Tiberius6996, Maeganm, Gauravxpresa, Tranzenic, Rajavenu.itm, Jacektomas, Monkbot, Pre8y, Flayneorange and Anonymous: 320
  - **Queue (abstract data type)** *Source:* [http://en.wikipedia.org/wiki/Queue%20\(abstract%20data%20type\)?oldid=653430150](http://en.wikipedia.org/wiki/Queue%20(abstract%20data%20type)?oldid=653430150) *Contributors:* BlckKnight, Andre Engels, DavidLevinson, LapoLuchini, Edward, Patrick, Michael Hardy, Ixfd64, Ahoerstemeier, Nanshu, Glenn, Emperorbma, Dcoetzee, Furrykef, Traroth, Metasquares, Jusiib, PuzzletChung, Robbot, Noldoaran, Fredrik, JosephBarillari, Rasmus Faber, G.Dhavale, Nipunbayas, PranavAmbhore, Proxyma, BattyBot, Abhidesai, Nova2358, ChrisGualtieri, Flaqueleto, Chengshuotian, Kushal-biswas777, Mogism, Makecat-bot, Bentonjimmy, Icarot, Tiberius6996, Maeganm, Gauravxpresa, Tranzenic, Rajavenu.itm, Jacektomas, Monkbot, Pre8y, Flayneorange and Anonymous: 42

Tobias Bergemann, Giftlite, Massyett, BenFrantzDale, MingMecca, Zoney, Rdsmith4, Tsemii, Andreas Kaufmann, Corti, Discospinster, Wrp103, Mecanismo, Mehrenberg, Indil, Kwamikagami, Chairboy, Spoon!, Robotje, Helix84, Zachlipton, Alansohn, Liao, Conan, Gun-slinger47, Mc6809e, Caesura, Jguk, Kenyon, Woohookitty, Mindmatrix, Peng-enwiki, MattGiua, Ruud Koot, Graham87, Rachel1, Qwertus, DePiep, Olivier Teuliere, Bruce Lee, W3bbo, Margosbot-enwiki, Wouter.oet, Ewlyahoocom, Jrtayloriv, Zotel, Roboto de Ajvol, PhilipR, RussBot, J. M., SpuriousQ, Stephenb, Stassats, Howcheng, JohJak2, Caerwine, Mike1024, Carlosguitar, SmackBot, Honza Záruba, M2MM4M, Dabear-enwiki, Chris the speller, Oli Filth, Wikibarista, Nbarth, DHN-bot-enwiki, OrphanBot, Zvar, Cybercobra, Pissant, Mlpkr, Cdills, Kflorence, Almkglor, PseudoSudo, Ckatz, 16@r, Sharcho, Nutster, Penbat, VTBassMatt, Banditlord, SimenH, Tawkerbot4, Christian75, X96lee15, Uruiamme, Thadius856, Hires an editor, Lperez2029, Egerhart, Defective, SiobhanHansa, Wikilolo, MikeDunlavey, Gwern, GrahamDavies, Sanjay742, Contactbanish, Nwbeeson, Bobo2000, AlnoktaBOT, JhsBot, Broadbot, Atifl, BotKung, Jesin, Calliopejen1, BotMultichill, Ham Pastrami, Keilana, Thesuperslacker, Hariva, Arsenic99, Chelseafan528, WikiBotas, ClueBot, Ggia, Vanmaple, Alexbot, Ksulli10, Jotterbot, TobiasPersson, SensuShinobu1234, DumZiBoT, Kletos, XLinkBot, SilvonenBot, Marry314113, Dsimic, Addbot, Some jerk on the Internet, OliverTwisted, MrOllie, SoSaysChappy, چانپ, Loupeter, Yobot, Vanished user rt41as76lk, KamikazeBot, Materialscientist, LilHelpa, Xqbot, Vegpuff, Joseph.w.s-enwiki, DSisypBot, Ruby.red.roses, FrescoBot, Mark Renier, Miklcct, Gbduende, PrometheeFeu-enwiki, Maxwellterry, John lindgren, Garfieldnate, EmausBot, Akerans, Redhanker, Soran-cio, Donner60, Clehner-enwiki, Grafca, ClueBot NG, Detonadorado, MahlerFive, Ztothefifth, Rahulghose, Iprathik, Zanaferx, Tlefebvre, PhuksyWiki, Fswangke, Dmitrysobolev, Nemo Kartikeyan, Kushalbiswas777, DavidLeighEllis, Tranzenic, ScottDNelson, Ishanalgorithm and Anonymous: 197

- **Double-ended queue** *Source:* <http://en.wikipedia.org/wiki/Double-ended%20queue?oldid=650737871> *Contributors:* The Anome, Frecklefoot, Edward, Axlrosen, CesarB, Dcoetze, Dfeuer, Zoiicon5, Furykef, Fredrik, Merovingian, Rasmus Faber, Tobias Bergemann, Smjg, Sj, BenFrantzDale, Esrogs, Chowbow, Rosen, Andreas Kaufmann, Pt, Spoon!, Mindmatrix, Ruud Koot, Mandarax, Wikibofh, Drrngrvy, Naraht, Ffaarr, YurikBot, Fabartus, Jengelh, SpuriousQ, Fbergo, Schellhammer, Ripper234, Sneftel, Bcbell, SmackBot, Cparker, Psiphior, Chris the speller, Kurykh, TimBentley, Oli Filth, Silly rabbit, Nbarth, Luder, Puetzk, Cybercobra, Offby1, Dicklyon, CmdrObot, Penbat, Funnyfarmofdoom, Mwhitlock, Omicronpersei8, Headbomb, VictorAnyakin, Felix C. Stegerman, David Eppstein, MartinBot, Huzzlet the bot, KILNA, VolkovBot, Anonymous Dissident, BotKung, Ramiromagalhaes, Kbrose, Hawk777, Ham Pastrami, Krishna.91, Rdhettinger, Foxj, Alexbot, Rhododendrites, XLinkBot, Dekart, Wolkykim, Matěj Grabovský, Rrmsgj, Legobot, Yobot, Sae1962, LittleWink, EmausBot, WikitanvirBot, Aamirlang, E Nocress, ClueBot NG, Ztothefifth, Shire Reeve, Helpful Pixie Bot, BG19bot, Gauravi123, Mtnorthpoplar and Anonymous: 88
- **Priority queue** *Source:* <http://en.wikipedia.org/wiki/Priority%20queue?oldid=656511654> *Contributors:* Frecklefoot, Michael Hardy, Nix-dorf, Bdonlan, Dcoetze, Sanxiyn, Robbot, Fredrik, Kowey, Bkell, Tobias Bergemann, Decrypt3, Giftlite, Zigger, Vadmium, Andreas Kaufmann, Byrial, BACbKA, El C, Spoon!, Bobo192, Nyenyec, Dbeardsl, Jeltz, Mbloore, Forderud, RyanGerbil10, Kenyon, Woohookitty, Oliphant, Ruud Koot, Hdante, Pete142, Graham87, Qwertus, Pdelong, Ckelloug, Vegaswikan, StuartBrady, Jeff02, Spl, Anders.Warga, Gareth Jones, Lt-wiki-bot, PaulWright, SmackBot, Emeraldemon, Stux, Gilliam, Riedl, Oli Filth, Silly rabbit, Nbarth, Kostmo, Zvar, Cybercobra, BlackFingolfin, Clicketyclack, Ninjagecko, Robbins, Rory O'Kane, Sabik, John Reed Riley, ShelfSkewed, Chrisahn, Corpz, Omicronpersei8, Thijs!bot, LeeG, Mentifisto, AntiVandalBot, Wayiran, CosineKitty, Ilingod, VoABot II, David Eppstein, Jutiphan, Umpthee, Squids and Chips, TXiKiBoT, Coder Dan, Red Act, RHaden, Rhanekom, SieBot, ThomasTenCate, EnOreg, Volkan YAZICI, ClueBot, Niceguyedc, Thejoshwolfe, SchreiberBike, BOTarate, Krungie factor, DumZiBoT, XLinkBot, Ghettoblaster, Vield, Incraston, Lightbot, Legobot, Yobot, FUZxxl, Bestiasonica, 1exec1, Kimsey0, Xqbot, Redroo, Thore Husfeldt, FrescoBot, Hobsonlane, Itusg15q4user, Orangeroof, ElNuevoEinstein, HenryAyoola, EmausBot, LastKingpin, Moswento, Arkenflame, Meng6, GabKBel, ChuispastonBot, Highway Hitchhiker, Ztothefifth, Widr, FutureTrillionaire, Happyuk, Chmarkine, J.C. Labbrev, Kushalbiswas777, MeekMelange, Lone boatman, Sriharsh1234, Theemathas, Dough34, Mydog333, Luckysud4, Sammydre, Bladesshade2, Mtnorthpoplar and Anonymous: 133
- **Associative array** *Source:* <http://en.wikipedia.org/wiki/Associative%20array?oldid=659894092> *Contributors:* Damian Yerrick, Robert Merkel, Fubar Obfusco, Maury Markowitz, Hirzel, B4hand, Paul Ebermann, Edward, Patrick, Michael Hardy, Shellreef, Graue, Minesweeper, Brianiac, Samuelsen, Bart Massey, Hashar, Dcoetze, Dysprosia, Silvonen, Bevo, Robbot, Naldoaran, Fredrik, Altenmann, Wlievens, Catbar, Wikibot, Ruakh, EvanED, Jleedev, Tobias Bergemann, Ancheta Wis, Jpo, DavidCary, Mintleaf-enwiki, Inter, Wolf-keeper, Jorge Stolfi, Macrakis, Pne, Neilc, Kusunose, Karol Langner, Bosmon, Int19h, Andreas Kaufmann, RevRagnarok, Ericamick, LeeHunter, PP Jewel, Kwamikagami, James b crocker, Spoon!, Bobo192, TommyG, Minghong, Alansohn, Mt-enwiki, Krischik, Sligocki, Kdau, Tony Sidaway, RainbowOfLight, Forderud, TShilo12, Boothy443, Mindmatrix, RzR-enwiki, Apokrif, Kglavin, Bluemoose, ObsidianOrder, Pfunk42, Yurik, Swmed, Scandum, Koavf, Agorf, Jeff02, RexNL, Alvin-cs, Wavelength, Fdb, Maerk, Dggoldst, Cedar101, JLaTondre, Ffangs, TuukkaH, SmackBot, KnowledgeOfSelf, MeiStone, Mirzabah, TheDoctor10, Sam Pointon, Brianski, Hugo-cs, Jdh30, Zven, Cfalin, CheesyPuffs144, Malbrain, Nick Levine, Vegard, Radagast83, Cybercobra, Decltype, Paddy3118, AvramYU, Doug Bell, AmiDaniel, Antonielly, EdC-enwiki, Tobe2199, Hans Bauer, Dreftymac, Pimlottc, George100, JForget, Jokes Free4Me, Pgr94, MrSteve, Countchoc, Ajo Mama, WinBot, Oddity-, Alphachimpbot, Maslin, JonathanCross, Pfast, PhiLho, Wimbolle, Magioladitis, David Eppstein, Gwern, Doc aberdeen, Signalhead, VolkovBot, Chaos5023, Kyle the bot, TXiKiBoT, Anna Lincoln, BotKung, Comet--berkeley, Jesdisciple, PanagosTheOther, Nemo20000, Jerryobject, CultureDrone, Anchor Link Bot, ClueBot, Irishjugg-enwiki, XLinkBot, Orb-nauticus, Frostus, Dsimic, Deineka, Addbot, Debresser, Jarble, Bartledan, Davidwhite544, Margin1522, Legobot, Luckas-bot, Yobot, TaBOT-zerem, Pcap, Peter Flass, AnomieBOT, RobotBOT, January2009, Sae1962, Efadae, Neil Schipper, Floatingdecimal, Tushar858, EmausBot, WikitanvirBot, Marcos canbeiro, AvicBot, ClueBot NG, JannuBl22t, Helpful Pixie Bot, Chr23, Mithrasgregoriae, JYBot, Dc-saba70, Alonsoguillen and Anonymous: 189
- **Bidirectional map** *Source:* <http://en.wikipedia.org/wiki/Bidirectional%20map?oldid=660118958> *Contributors:* Andreas Kaufmann, Xebeth, Cyc-enwiki, Ruud Koot, GregorB, Derek R Bullamore, P199, CmdrObot, Mahdavi110, Dtech, Cobi, Addbot, Ethanpet113, Yobot, Efadae, Skyerise, Sreekarun, Jesse V., Mean as custard, Mattbierner, Helpful Pixie Bot, Cgdecker, Crunchy nibbles, Magixbox, Atepor-marus, Mtnorthpoplar and Anonymous: 4
- **Multimap** *Source:* <http://en.wikipedia.org/wiki/Multimap?oldid=654560497> *Contributors:* GCarty, Topbanana, Enochlau, Aminorex, Andreas Kaufmann, Spoon!, Bantman, Ruud Koot, TheDJ, TuukkaH, Fuhghettaboutit, Cybercobra, JonathanWakely, Macha, Wmbolle, Vainolo, Svick, Excirial, Bluebusy, JakobVoss, BluePyth, Not enough names left, BG19bot, Ceklock, ChrisGaultieri, Pintoch, Iokevins, Rhooke, Mtnorthpoplar and Anonymous: 15
- **Set (abstract data type)** *Source:* [http://en.wikipedia.org/wiki/Set%20\(abstract%20data%20type\)?oldid=650974736](http://en.wikipedia.org/wiki/Set%20(abstract%20data%20type)?oldid=650974736) *Contributors:* Damian Yerrick, William Avery, Mintguy, Patrick, Modster, TakuyaMurata, EdH, Mxn, Dcoetze, Fredrik, Jorge Stolfi, Lvr, Urhixidur, Andreas Kaufmann, CanisRufus, Spoon!, RJFJR, Ruud Koot, Pfunk42, Bgwhite, Roboto de Ajvol, Mlc, Cedar101, QmunkE, Incnis Mrsi, Bluebot, MartinPoulter, Nbarth, Gracenotes, Otus, Cybercobra, Dreadstar, Wizardman, MegaHasher, Hetar, Amniarix, CBM, Polaris408, Peterdjones, Hosamaly, Hut 8.5, Wikilolo, Lt basketball, Gwern, Raise exception, Fylwind, Davecrosby uk, BotKung, Rhanekom, SieBot,

Oxymoron83, Casablanca2000in, Classicaecon, Linforest, Niceguyedc, UKoch, Quinntaylor, Addbot, SoSaysChappy, Loupeter, Legobot, Luckas-bot, Denispirl, Pcap, AnomieBOT, Citation bot, Twri, DSisypBot, GrouchoBot, FrescoBot, Spindocter123, Tyamath, EmausBot, Wikipelli, Elaz85, Mentibot, Nullzero, Helpful Pixie Bot, Poomam7393, Umasoni30, Vimalwatwani, Chmarkine, Irene31, Mark viking, FriendlyCaribou, Brandon.heck, Aristiden7o and Anonymous: 43

- **Tree (data structure)** *Source: [http://en.wikipedia.org/wiki/Tree%20\(data%20structure\)?oldid=660590536](http://en.wikipedia.org/wiki/Tree%20(data%20structure)?oldid=660590536)* *Contributors:* Bryan Derksen, The Anome, Tarquin, BlckKnight, Seb, Boleslav Bobcik, FvdP, Perique des Pallettes, Hotlorp, Mrwojo, Rp, Loisel, Mdebets, Kragen, Julesd, Glenn, Nikai, Evercat, BAxelrod, Dcoetzee, Andrevan, Dysprosia, Jitse Niesen, Robbot, Fredrik, Pingveno, KellyCoinGuy, Ruakh, Tobias Bergemann, Giftlite, Jorge Stolfi, Alvestrand, Mckaysalisbury, Utcursh, Pgdn002, Cbraga, Knutux, Kjetil r, Two Bananas, Creidieki, Dcandeto, Corti, Jiy, Discospinster, Rich Farmbrough, T Long, Paul August, Robertbowerman, Night Gyr, Mattisgoo, RoyBoy, Bobo192, Vdm, R. S. Shaw, Giraffedata, Sara wiki, Nsaa, Mdd, Liao, Menphix, Arthena, Wtmitchell, Wsload, Nuno Tavares, Mindmatrix, Cruccone, RzR-enwiki, Ruud Koot, Btyner, KoRnholio8, Graham87, Qwertyus, Okj579, VKokielov, Mathbot, Margosbot-enwiki, Nivix, DevastatorIIC, Fresheneesz, WhyBeNormal, DVdm, Reetep, Digitalme, Skraz, Wavelength, RobotE, Fabricationary, Stephenb, Iamf-scked, Fabiogramos-enwiki, RazorICE, JulesH, Ripper234, SteveWitham, SmackBot, Mmernex, Jagged 85, Thunderboltz, Mdd4696, Bernard François, Gilliam, Kurykh, Silly rabbit, Nbarth, Cfallin, Hashbrowncipher, Can't sleep, clown will eat me, Afrozenator, Zrulli, Cybercobra, Acdx, Kuru, Tomhubbard, Limaner, Nbhatla, Iridescent, CapitalR, INKubusse, FleetCommand, Jokes Free4Me, Kineticman, Cydebot, Skittleys, N5iln, Michael A. White, Nemti, Mentifisto, Majorly, Gioto, Seaphoto, Alphachimpbot, Myanw, MagiMaster, JAnD-bot, MER-C, SiobhanHansa, Magioladitis, VoABot II, David Eppstein, Lisamh, Jfroelich, Trusilver, SlowJog, 2help, Wxidea, Rponamgi, Anonymous Dissident, Defza, BotKung, Alfredo J. Herrera Lago, NHRHS2010, Dwandelt, Ricardosardenberg, Kpeeters, Iamtheudeus, Yintan, Ham Pastrami, Pi is 3.14159, AlexWaelde, Taemyr, Sbowers3, Dillard421, Svick, Sean.hoyland, Xevior, ClueBot, Justin W Smith, The Thing That Should Not Be, Garyzx, Adrianwn, Happynomad, Lartoven, Ykhwong, Anoshak, Aseld, Kausikghatak, Thingg, SoxBot III, XLinkBot, Ladsgroup, Mike314113, Addbot, Ronjhones, Ollydbg, Thomas Björkan, Eh kia, Tide rolls, Jarble, Solbris, Luckas-bot, Jim1138, Pradeepvaishy, Shamus1331, Twri, ArthurBot, Xqbot, Bdmy, Martnym, Abce2, Grixlkraxl, RibotBOT, Erik9, Thehelpfulbot, Mark Renier, D'ohBot, I dream of horses, Half price, Extra999, Jfmantis, DRAGON BOOSTER, EmausBot, Dixtosa, Tommy2010, Mercury1992, Thibef, Mnogo, Wenttomowameadow, Ovakim, TyA, Coasterlover1994, Chris857, ClueBot NG, Marechal Ney, Rezabot, Ramzysamman, Kaswini15, Floating Boat, HiteshLinus, Adubi Stephen, Lukevanin, Justincheng12345-bot, Pratyaa Ghosh, Electricmuf-fin11, Padenton, Jochen Burghardt, Jamesmcmahon0, Revolution1221, Oonsouomesta, Meteor sandwich yum, Gronk Oz, Biblioworm, Rvraghav93, Miathan and Anonymous: 304
- **Array data structure** *Source: <http://en.wikipedia.org/wiki/Array%20data%20structure?oldid=654863715>* *Contributors:* The Anome, Ed Poor, Andre Engels, Tsja, B4hand, Patrick, RTC, Michael Hardy, Norm, Nixdorf, Graue, TakuyaMurata, Alfio, Ellywa, Julesd, Cgs, Poor Yorick, Rossami, Dcoetzee, Dysprosia, Jogloran, Wernher, Fvw, Sewing, Robbot, Josh Cherry, Fredrik, Lowelian, Wikibot, Jleedev, Giftlite, DavidCary, Massyett, BenFrantzDale, Lardarse, Ssd, Jorge Stolfi, Macrakis, Jonathan Grynspan, Lockeownzj00, Beland, Vanished user 1234567890, Karol Langner, Icairns, Simoneau, Jh51681, Andreas Kaufmann, Matth90, Corti, Jkl, Rich Farmbrough, Guanabot, Qutezuce, ESkog, ZeroOne, Danakil, MisterSheik, G worroll, Spoon!, Army1987, Func, Rbj, Mdd, Jumbuck, Mr Adequate, Atanamir, Krischik, Tawwasser, ReyBrujo, Suruena, Rgrig, Forderud, Beliavsky, Beej71, Mindmatrix, Jimbryho, Ruud Koot, Jeff3000, Grika, Palica, Gerbrant, Graham87, Kbdank71, Zzedar, Ketiltrot, Bill37212, Ligulem, Yamamoto Ichiro, Mike Van Emmerik, Quuxplusone, Intgr, Visor, Sharkface217, Bgwhite, YurikBot, Wavelength, Borgx, RobotE, RussBot, Fabartus, Splash, Piet Delport, Stephenb, Pseudomonas, Kimchi.sg, Dmason, JulesH, Mikeblas, Bota47, JLaTondre, Hide&Reason, Heavyrain2408, SmackBot, Princeatapi, Blue520, Trojo-enwiki, Brick Thrower, Alksub, Apers0n, Betacommand, GwydionM, Anwar saadat, Keegan, Timneu22, Mcaruso, Tscabot, Tamfang, Berland, Cybercobra, Mwtoews, Masterdriverz, Kukini, Smremde, SashatoBot, Derek farn, John, 16@r, Slakr, Beetstra, Dreftymac, Courcelles, George100, Ahy1, Engelec, Wws, Neelix, Simeon, Kaldosh, Travelbird, Mrstonky, Skittleys, Strangely, Christian75, Narayanes, Eprb123, Sagaciousuk, Trevyn, Escarbot, Thadius856, AntiVandalBot, AbstractClass, JAnDbot, JaK81600-enwiki, MER-C, Cameltrader, PhiLho, SiobhanHansa, Magioladitis, VoABot II, Ling.Nut, DAGwyn, David Eppstein, User A1, Squidonius, Gwern, Highegg, Themania, Patar knight, J.delanoy, Slogswipe, Darkspots, Jayden54, Mfb52, Funandtrvl, VolkovBot, TXiKiBoT, Anonymous Dissident, Don4of4, Amog, Redacteur, Nicvaroce, Kbrose, SieBot, Caltas, Garde, Tiptoety, Paolo.dL, Oxymoron83, Svick, Anchor Link Bot, Jlmerrill, ClueBot, LAX, Jackollie, The Thing That Should Not Be, Alksentr, Rilak, Supertouch, R000t, Liempt, Excirial, Immortal Wowbagger, Bargomm, Thingg, Footballfan190, Johnuniq, SoxBot III, Awbell, Chris glenne, Staticshakedown, SilvonenBot, Henry513414, Dsimic, Gaydudes, Btx40, EconoPhysicist, SamatBot, Zorrobot, Legobot, Luckas-bot, Yobot, Pitbotgourou, Fraggle81, Peter Flass, Tempodivalse, Obersachsebot, Xqbot, SPTWriter, FrescoBot, Citation bot 2, I dream of horses, HRoestBot, MastiBot, Jandalhander, Laurențiu Dascălu, Dinamik-bot, TylerWilliamRoss, Merlinsonca, Jfmantis, The Utahraptor, EmausBot, Mfahem007, Donner60, Ipsign, Ieee andy, EdoBot, Muzadded, Mikhail Ryazanov, ClueBot NG, Widr, Helpful Pixie Bot, Roger Wellington-Oguri, Wbm1058, 111008066it, Mark Arsten, Simba2331, Insidiae, ChrisGualtieri, A'bad group, Makecat-bot, Chetan chopade, Ginsuloft and Anonymous: 239
- **Row-major order** *Source: <http://en.wikipedia.org/wiki/Row-major%20order?oldid=657111753>* *Contributors:* Awaterl, Tedernst, Stevenj, BenFrantzDale, Rubik-wuerfel, Pearl, Drf5n, Arthena, Stefan.karpinski, Zawersh, Woohookitty, Tlroche, Strait, Bgwhite, Splintercell-guy, Welsh, Mikeblas, Rwalker, Aaron Will, SmackBot, Mstahl, Mcld, Liberio, Jungrung, Gwern, Nzroller, ZMughal, BarroColorado, Sintaku, Czarkoff, XLinkBot, Gazto, Jonhoo, Yobot, Martnym, Taweetham, Kiefer.Wolfowitz, LittleWink, Jfmantis, JordiGH, ClueBot NG, Rhymeswthorange, Trebb, Misev, Lemnaminor, Theoryno3, Ethically Yours, Monkbot, Lazy8s, Cyque and Anonymous: 60
- **Dope vector** *Source: <http://en.wikipedia.org/wiki/Dope%20vector?oldid=571585342>* *Contributors:* Wik, Sjorford, Xegeo, Finn-Zoltan, Andreas Kaufmann, CanisRufus, Gennaro Prota, Amalas, Dougher, Yobot, Gongshow, Peter Flass, AnomieBOT, LilHelpa, Phresnel, LivinInTheUSA, Stamos20 and Anonymous: 5
- **Iliffe vector** *Source: <http://en.wikipedia.org/wiki/Iliffe%20vector?oldid=634956027>* *Contributors:* Michael Hardy, Doradus, Tobias Bergemann, Pgdn002, Andreas Kaufmann, CanisRufus, Spoon!, Cmdrjameson, Lkinkade, Feydey, Alynnna Kasmira, Ospalh, Wotnarg, Mgreenbe, Cybercobra, Ligulembot, NapoliRoma, RainbowCrane, Gwern, STBot, Staticshakedown, DOI bot, Yobot, Challisc and Anonymous: 12
- **Dynamic array** *Source: <http://en.wikipedia.org/wiki/Dynamic%20array?oldid=659063999>* *Contributors:* Damian Yerrick, Edward, Ixfd64, Phoe6, Dcoetzee, Furrykef, Wdscxsj, Jorge Stolfi, Karol Langner, Andreas Kaufmann, Moxfyre, Dpm64, Wrp103, Forbsey, ZeroOne, MisterSheik, Spoon!, Ryk, Fresheneesz, Wavelength, SmackBot, Rönin, Bluebot, Octahedron80, Nbarth, Cybercobra, Declitype, MegaHasher, Beetstra, Green caterpillar, Tobias382, Icep, Wikilolo, David Eppstein, Gwern, Cobi, Spinningspark, Arbor to SJ, Ctxppc, ClueBot, Simonykill, Garyzx, Alex.vatchenko, Addbot, AndersBot, امان, Aekton, Didz93, Tartarus, Luckas-bot, Yobot, Rubinbot, Citation bot, SPTWriter, FrescoBot, Mutinus, Patmorin, WillNess, EmausBot, Card Zero, François Robere and Anonymous: 39
- **Hashed array tree** *Source: <http://en.wikipedia.org/wiki/Hashed%20array%20tree?oldid=649315574>* *Contributors:* Dcoetzee, Surachit, MegaHasher, Alaibot, RainbowCrane, Cobi, Garyzx, Queenmomcat, WillNess, ChrisGualtieri and Anonymous: 3

- **Gap buffer** *Source:* <http://en.wikipedia.org/wiki/Gap%20buffer?oldid=654236982> *Contributors:* Damian Yerrick, LittleDan, Alf, Charles Matthews, Jogloran, Populus, Tobias Bergemann, Macrakis, Pgdn002, Andreas Kaufmann, Dpm64, Jaberwocky6669, MisterSheik, Chronodm, Hydrostatic, Cybercobra, Hosamaly, Drudru, MER-C, AlleborgoBot, Niceguyedc, Jacobslusser, Alexey Muranov, Addbot, Fyrael, AnomieBOT, J04n, Ansumang, Dennis714, Hitendrashukla, Aniketpate, Mark Arsten, Alexjsthornton and Anonymous: 10
- **Circular buffer** *Source:* <http://en.wikipedia.org/wiki/Circular%20buffer?oldid=656496379> *Contributors:* Damian Yerrick, Julesd, Malcohol, Chocolateboy, Tobias Bergemann, DavidCary, Andreas Kaufmann, Astronouth7303, Foobaz, Shabble, Cburnett, Qwertyus, Bgwhite, Pok148, Cedar101, Mhi, WolfWings, SmackBot, Ohnoitsjamie, Chris the speller, KiloByte, Silly rabbit, Rrelf, Frap, Cybercobra, Zoxc, Mike65535, Anonymi, Joejadams, Mark Giannullo, Headbomb, Llioic, ForrestVoight, Marokwitz, Hosamaly, Parthashome, Magioladitis, Indubitably, Amikake3, Strategist333, Billinghurst, Rhanekom, Calliopejen1, SiegeLord, IsaacAA, 田中, OlivierEM, DrZoomEN, Para15000, Niceguyedc, Lucius Annaeus Seneca, Aparition11, Dekart, Dsimic, Addbot, Shervinemami, MrOllie, OrlinKolev, Matěj Grabovský, Yobot, Ptbotgourou, Tennenrishin, AnomieBOT, BastianVenthur, ChrisCPearson, Serkan Kenar, Shirik, 78.26, Mayukh ittibombay 2008, Hoo man, Sysabod, Ybngalobill, Paulitex, Lipsio, Eight40, ZéroBot, Bloodust, Pokbot, Asimsalam, Shengliangsong, Lemtronix, Exfuent, Tectu, Msoltyspl, Cerabot~enwiki, ScotXW, Jijubin, Hailu143, EUROCALYPTUSTREE and Anonymous: 95
- **Sparse array** *Source:* <http://en.wikipedia.org/wiki/Sparse%20array?oldid=643193770> *Contributors:* Charles Matthews, Timrollpickering, Monedula, Karol Langner, Kenb215, MisterSheik, CanisRufus, Rjwimski, Pako, Hgrangqvist, Cadillac, SmackBot, Incnis Mrsi, 16@r, KurtRaschke, Engelec, Alaibot, RainbowCrane, Funandtrvl, Freependulum, Hawk777, Addbot, Chamal N, WQUlrich, FeatherPluma, Kolrok, Delusion23, Megha92, BPositive, Kalyani Kulkarni, BattyBot, DoctorKubla and Anonymous: 16
- **Bit array** *Source:* <http://en.wikipedia.org/wiki/Bit%20array?oldid=660923003> *Contributors:* Awaterl, Boud, Pnm, Dcoetzee, Furrykef, JesseW, AJim, Bovlb, Vadmiuum, Karol Langner, Sam Hocevar, Andreas Kaufmann, Notinasnaid, Paul August, CanisRufus, Spoon!, R. S. Shaw, Rgrig, Forderud, Jacobolus, Bluemoose, Hack-Man, StuartBrady, Intgr, RussBot, Cedar101, TomJF, JLaTondre, Chris the speller, Bluebot, Doug Bell, Archimerged, DanielLemire, Glen Pepicelli, CRGreathouse, Gyopi, Neelix, Davnor, Kubanczyk, Izyt, Gwern, Themania, R'n'B, TheChrisD, Cobi, Pcordes, Bvds, RomainThibaux, Psychless, Skwa, Onomou, MystBot, Addbot, IOLJeff, Tide rolls, Bluebusy, Peter Flass, Rubinbot, JnRouvignac, ZéroBot, Nomen4Omen, Cocciasik, ClueBot NG, Snotbot, Minakshinajardhane, Chmarkine, Chip123456, BattyBot, Mogism, Thajdog10, User85734, François Robere, Chadha.varun, Francisco Bajumuzi and Anonymous: 51
- **Bitboard** *Source:* <http://en.wikipedia.org/wiki/Bitboard?oldid=643468367> *Contributors:* Damian Yerrick, Dwheeler, Notheruser, Furrykef, Nv8200p, Jleedev, Dissident, Quackor, Andreas Kaufmann, ZeroOne, Kenneth Cochran, RoyBoy, Triona, Tromp, Pearle, Rkalyankumar, RJFJR, M412k, Marudubshinki, Bubba73, Srlleffler, Epolk, Trovatore, DGJM, Avalon, Hirak 99, Zargulon, Psu256, SmackBot, Slamb, QTCaptain, Kaimiddleton, PrometheusX303, Nibuod, Sigma 7, Daniel.Cardenas, Doug Bell, Brainix, Glen Pepicelli, Danielmachin, CmdrObot, Drinibot, Cydebot, AntiVandalBot, Luna Santin, Bigtmepeace, Dylan Lake, MRProgrammer, MER-C, IanOsgood, Gwern, STBot, WiiWillieWiki, Pawnkingthree, VolkovBot, Lokiclock, Miko3k, Pjoeff, Ddxc, CharlesGillingham, Alexbot, Mynameisnotpj, PixelBot, Sun Creator, Luismarques83, Addbot, Ethanpet113, Lightbot, AnomieBOT, Nippashish, Dr.Szlachedzki, Updatehelper, Will Beback Auto, Helpful Pixie Bot, Feg99 and Anonymous: 61
- **Parallel array** *Source:* <http://en.wikipedia.org/wiki/Parallel%20array?oldid=645091491> *Contributors:* TakuyaMurata, Charles Matthews, Dcoetzee, Ruakh, Karol Langner, Peter bertok, Andreas Kaufmann, Thorwald, AlanBarrett, MisterSheik, Dgpop, Apoc2400, GregorB, TheProgrammer, Veledan, Ragzouken, SmackBot, Gwern, Alik Kirillovich, Garyzx, Addbot, Ironholds, RedBot, ClueBot NG, West side 031, Dalton Quinn and Anonymous: 13
- **Lookup table** *Source:* <http://en.wikipedia.org/wiki/Lookup%20table?oldid=654295861> *Contributors:* Patrick, Michael Hardy, Pnm, Angela, Arteitle, Charles Matthews, Dcoetzee, Dysprosia, Omegatron, Davidmaxwaterman, Fredrik, Gazoot, Ralian, Gifflite, Khalid hassani, Girolamo Savonarola, Andreas Kaufmann, Qef, Bender235, CanisRufus, NicM, Woohookitty, Melesse, FlaBot, SteveBaker, Wavelength, Peter S., Jengelh, KSmrq, Welsh, Yahya Abdal-Aziz, Cheese Sandwich, JLaTondre, Ohnoitsjamie, Chris the speller, Sadi~enwiki, Nbarth, Kostmo, Berland, Cybercobra, Genpfault, Gennaro Prota, Fingew, MrDomino, ZAB, Kencf0618, CRGreathouse, David@sickmiller.com, Yflicker, Thij's!bot, Headbomb, Graemec2, Dano312, Dylan Lake, Yellowdesk, Kdakin, Sterrys, .anacondabot, Xoneca, Falcor84, Gwern, Supernoob~enwiki, VolkovBot, Itemirus, AlleborgoBot, SieBot, DaBler, ClueBot, Abhinav, DragonBot, DumZiBoT, Addbot, Mortense, Poco a poco, Jelsova, MrOllie, AnomieBOT, VanishedUser sdu9aya9fasdspa, Citation bot, Petter.kallstrom, Xqbot, Bunny.scientist, RobotBOT, Kyng, Vrenator, EmausBot, Tuankiet65, Deweitech, Cal-linux, ZéroBot, Armandas j, Vkehayas, ClueBot NG, Epok, Sjmoquin, CitationCleanerBot, MadCowpoke, Ysuralkar, ½ and Anonymous: 85
- **Linked list** *Source:* <http://en.wikipedia.org/wiki/Linked%20list?oldid=661290826> *Contributors:* Uriyan, BlckKnght, Fubar Obfusco, Perique des Palottes, BL~enwiki, Paul Ebermann, Stevertigo, Nixdorf, Kku, TakuyaMurata, Karingo, Minesweeper, Stw, Angela, Smack, Dwo, MatrixFrog, Dcoetzee, RickK, Ww, Andrewman327, IceKarma, Silvonen, Thue, Samber~enwiki, Traroth, Kenny Moens, Robbot, Astronautics~enwiki, Fredrik, Yacht, 75th Trombone, Wereon, Pengo, Tobias Bergemann, Enochlau, Gifflite, Achurch, Elf, Haeleth, BenFrantzDale, Kenny sh, Levin, Jason Quinn, Jorge Stolfi, Mboverload, Ferdinand Pienaar, Neile, CryptoDerk, Supadawg, Karl-Henner, Sam Hocevar, Creidieki, Sonett72, Andreas Kaufmann, Jin~enwiki, Corti, Ta bu shi da yu, Poccil, Wrp103, Pluke, Antaeus Feldspar, DcoetzeeBot~enwiki, Jarsyl, MisterSheik, Shanes, Nickj, Spoon!, Bobo192, R. S. Shaw, Adrian~enwiki, Giraffedata, Mdd, JohnyDog, Arthena, Upnishad, Mc6809e, Lord Pistachio, Fawcett5, Theodore Kloba, Wtmitchell, Docboat, RJFJR, Versageek, TheCoffee, Kenyon, Christian \*Yendi\* Severin, Unixxx, Kelmar~enwiki, Mindmatrix, Arneth, Ruud Koot, Tabletop, Terence, Smiler jerg, Rnt20, Graham87, Magister Mathematicae, BD2412, TedPostol, StuartBrady, Arivne, Intgr, Adamking, King of Hearts, Bgwhite, YurikBot, Borgx, Deeptrivia, RussBot, Jengelh, Grafen, Welsh, Daniel Mietchen, Raven4x4x, Quentin mcalmott, ColdFusion650, Cesarsorm~enwiki, Tetracube, Clayhalliwell, LeonardoRob0t, Bluezy, Katieh5584, Tyomitch, Willemo, RichF, 田中, robot, SmackBot, Waltercruz~enwiki, FlashSheridan, Rönin, Sam Pointon, Gilliam, Leafboat, Rmosler2100, NewName, Chris the speller, TimBentley, Stevage, Nbarth, Colonies Chris, Deshraj, JonHarder, Cybercobra, IE, MegaHasher, Lasindi, Atkinson 291, Dreslough, Jan.Smolik, NJZombie, Minna Sora no Shita, 16@r, Hvn0413, Beetstra, ATren, Noah Salzman, Koweja, Hu12, Iridescent, Pavely, Aeons, Tawkerbot2, Ahy1, Penbat, VTBassMatt, Mblumber, JFreeman, Xenochria, HappyInGeneral, Headbomb, Marek69, Neil916, Dark knight, Nick Number, Danarmstrong, Thadius856, AntiVandalBot, Ste4k, Darklilac, Wizmo, JAnDbot, XyBot, MER-C, PhilKnight, SiobhanHansa, Wikilolo, VoABot II, Twsx, Japo, David Eppstein, Philg88, Gwern, Moggie2002, Tgeairn, Trusilver, Javawizard, Dillesca, Daniel5Ko, Cobi, KylieTastic, Ja 62, Brvman, Meiskam, Larryisgood, Vipinhari, Mantipula, Amog, BotKung, BigDunc, Wolfrock, Celticeric, B4upgradeep, Tomaxer, Albertus Aditya, Clowd81, Sprocter, Kbrose, Arjun024, J0hn7r0n, Wjl2, SieBot, Tiddly Tom, Yintan, Ham Pastrami, Pi is 3.14159, Keilana, Flyer22, TechTony, Redmarkviolinist, Beejay, Bughunter2, Mygerardromance, NHSKR, Hariva, Denisaron, Thorncrag, Startswithj, Scarlettwharton, ClueBot, Ravek, Justin W Smith, The Thing That Should Not Be, Raghaven, ImperfectlyInformed, Garyzx, Arakunem, Mild Bill Hiccup, Lindsaygilmour, TobiasPersson, SchreiberBike, Dixie91, Nasty psycho, XLinkBot, Marc van Leeuwen, Avoided, G7mcluvn, Hook43113, Kurniasan, Wollykim, Addbot, Anandvachhani, MrOllie, Freqsh0, Zorrobot, Jarble, Quantumobserver, Yobot, Fraggle81, KamikazeBot, Shadoninja, AnomieBOT, Jim1138, Materialscientist, Mwe 001, Citation bot, Quantran202, SPTWriter, Mtasic, Binaryedit, Miym, Etienne

Lehnart, Sophus Bie, Apollo2991, Constructive editor, Afromayun, Prari, FrescoBot, Meshari alnaim, Ijsf, Mark Renier, Citation bot 1, I dream of horses, Apeculiaz, Patmorin, Carloseow, Vrenator, Zvn, BZRatfink, Arjitzmalviya, Vhcomptech, WillNess, Minimac, Jfmantis, RjwilmsiBot, Agrammenos, EmausBot, KralSS, Simply.ari1201, Eniagrom, MaGa, Donner60, Carmichael, Peter Karlsen, 28bot, Sjoerd-debruin, ClueBot NG, Jack Greenmaven, Millermk, Rezabot, Widr, MerlIwBot, Helpful Pixie Bot, HMMSolent, BG19bot, WikiPhoenix, Tango4567, Dekai Wu, Computersagar, SaurabhKB, Klilidiplomus, Singlaive, IgushevEdward, Electricmuffin11, TalhaIrfanKhan, Frosty, Smortypi, RossMMooney, Gauravxpresa, Noyster, Suzrocksyu, Bryce archer, Melcous, Monkbot, Azx0987, Mahesh Dheravath, Vikas bhatnager, Aswincweety, RationalBlasphemist, Ishanalogrithm and Anonymous: 622

- **XOR linked list** *Source:* <http://en.wikipedia.org/wiki/XOR%20linked%20list?oldid=657715786> *Contributors:* Damian Yerrick, Nixdorf, Jketola, Cyp, Dcoetzee, Zoicon5, Saltine, Itai, McKay, Pakaran, Sander123, Haeleth, Dav4is, Siroxo, Eequor, Vadmiun, Andreas Kaufmann, Cacycle, Paul August, Sleske, Cjoev, Quuxplusone, Spasemunki, Nick, SmackBot, Quotemstr, Chris the speller, Bluebot, Octahedron80, Cybercobra, Jon Awbrey, Amniarix, Thijis!bot, Ais523, Funandtrvl, Tomaxer, VVVBot, XLinkBot, MystBot, Addbot, Lusum, Fraggle81, J04n, Jfmantis, Ebehn, Ysaimanoj, ClueBot NG, Nullzero, Brijesh.2011 and Anonymous: 32
- **Unrolled linked list** *Source:* <http://en.wikipedia.org/wiki/Unrolled%20linked%20list?oldid=647704322> *Contributors:* Damian Yerrick, Dcoetzee, DavidCary, Jason Quinn, Neilc, Andreas Kaufmann, Smyth, Fx2, Kdau, MattGiua, Yuriybrisk, Qwertyus, SmackBot, Ennorehling, Chris the speller, Warren, Ryszard Szopa~enwiki, Headbomb, Johnbod, Potatoswatter, Funandtrvl, Svick, Garyzx, Afrikaner, MystBot, Addbot, Kcsomisetty, DavidHarkness, VanishedUser sdu9aya9fasdsopa, Citation bot, Costaluiz, Patmorin, WillNess, Jfmantis, EmausBot, Shigeru23, Gampuzampe, AxeAdam20 and Anonymous: 8
- **VList** *Source:* <http://en.wikipedia.org/wiki/VList?oldid=650760819> *Contributors:* Mrwojo, Dcoetzee, Fredrik, Andreas Kaufmann, CanisRufus, Irrbloss, Ken Hirsch, Qwertyus, Eubot, WriterHound, Sandstein, Cdiggins, SmackBot, Chris the speller, Bluebot, Cybercobra, MegaHasher, Jbolden1517, Ripounet, Scramjet42, VVVBot, Siskus, Dawdler, Addbot, Luckas-bot, Denispire, GnuCivodul, EmausBot and Anonymous: 16
- **Skip list** *Source:* <http://en.wikipedia.org/wiki/Skip%20list?oldid=650607347> *Contributors:* Mrwojo, Stevenj, Charles Matthews, Dcoetzee, Dysprosia, Doradus, Populus, Noldoaran, Fredrik, Jrockway, Altenmann, Jorge Stolfi, Two Bananas, Andreas Kaufmann, Antaeus Feldspar, R. S. Shaw, Waffleguy4, Nkour, Ruud Koot, MarSch, Drpaule, Intgr, YurikBot, Wavelength, Piet Delport, Bovineone, Gareth Jones, Zr2d2, Cedar101, SmackBot, Chadmcaniel, Silly rabbit, Cybercobra, Viebel, Almkglor, Laurienne Bell, Nsfmc, CRGreathouse, Nczempin, Thijis!bot, Dougher, Bondolo, Sanchom, Magioladitis, A3nm, JaGa, STBotD, Musically ut, Funandtrvl, VolkovBot, Rhanekom, SieBot, Ivan Štambuk, MinorContributor, Menahem.fuchs, Cereblio, OKBot, Svick, Rdhettinger, Denisarona, PuercoPop, Gene91, Kukolar, Xcezbe, Braddunbar, Addbot, DOI bot, Jim10701, Luckas-bot, Yobot, Wojciech mula, AnomieBOT, SvartMan, Citation bot, Carlsotr, Alan Dawrst, RibotBOT, FrescoBot, Jamesooders, MastiBot, Devyni, Patmorin, EmausBot, Pet3ris, Jaspervdg, Overred~enwiki, ClueBot NG, Vishalvishnoi, Rpk512, BG19bot, ChrisGualtieri, Mark viking, Purealtruism, Dmx2010, Monkbot, چوئن, Mtnorthpoplar and Anonymous: 102
- **Self-organizing list** *Source:* <http://en.wikipedia.org/wiki/Self-organizing%20list?oldid=624941647> *Contributors:* Bearcat, Utcursch, Andreas Kaufmann, Foobaz, Rjwilmsi, Aclwon, JonHarder, Khazar, AshLin, VTBassMatt, Alaibot, Headbomb, Lilwik, Sun Creator, Dekart, AnomieBOT, PigFlu Oink, Ansumang, Snotbot, BG19bot, SaurabhKB, Roshmorris, JennyZito and Anonymous: 8
- **Binary tree** *Source:* <http://en.wikipedia.org/wiki/Binary%20tree?oldid=659710900> *Contributors:* LC~enwiki, Tarquin, BlckKnght, XJaM, FvdP, B4hand, Someone else, Mrwojo, Michael Hardy, Pit~enwiki, Dominus, Liftarn, SGBailey, TakuyaMurata, Loisel, Minesweeper, Ahoerstemeier, Julesd, Charles Matthews, Dcoetzee, Dysprosia, David Shay, AndrewKepert, Noldoaran, Altenmann, Bkell, Rege~enwiki, Ruakh, Giftlite, Ferkelparade, Mboverload, Cbraga, Beland, Heirpixel, Spottedowl, Pohl, Allefant, Andreas Kaufmann, Rich Farmbrough, Guanabot, Ilana, Drano, Rspeer, Antaeus Feldspar, Shoujun, Bobo192, ParticleMan, Darangho, Zetawoof, Mdd, Dguido, Liao, Silver hr, ABCD, Ynhockey, RussAbbott, Kgashok, Rzelnik, Kbolino, Linas, Daira Hopwood, Yuubinbako, WadeSimMiser, MONGO, Graham87, Alienus, Mjm1964, Bruce1ee, Vegaswikian, Oblivious, Bhadani, Mathbot, Wavelength, Mahahahaneapneap, Michael Slone, John Quincy Adding Machine, JabberWok, RadioFan2 (usurped), CambridgeBayWeather, Matir, Cdiggins, Loool, Abu adam~enwiki, Ekeb, RG2, Dkasak, Nippoo, SmackBot, Adam majewski, Pgk, Rönin, SmartGuy Old, Mcld, Gilliam, Chris the speller, KaragouniS, Jprg1966, Oli Filth, Jerome Charles Potts, Nbarth, Srvhrs, Cncplyr, Tarrahatikas, Zipdisc, Cybercobra, Kamirao, DMacks, Ck lostsword, Dr. Sunglasses, Kuru, Aroundthewayboy, Loadmaster, Martinp23, Waggers, Nonexistent, LandruBek, Zero sharp, Jafet, Ahy1, Smallpond, Simeon, Mike Christie, Cyhawk, Thrapper, Doug s, Gimmetrov, Malleus Fatuorum, Coelacan, Hazmat2, Itchy Archibald, AntiVandalBot, Luna Santin, Seaphoto, Opelio, Adijk, Eapache, JAnDbot, Ppelleti, Jheiv, VoABot II, Josephskeller, David Eppstein, Otterdam, Dontdoit, S3000, Shentino, Glrx, Maurice Carbonaro, Roreuqnoc, LightningDragon, Michael Angelkovich, Bonadea, Orphic, LokiClock, Gsmodi, Philip Trueman, DuaneBailey, BigDunc, Brianga, Roybristow, Rhanekom, Bfpage, Caltas, Jerryobject, Krishna.91, Jonson22, Reinderien, Taemyr, Djcollom, Jruderman, Anchor Link Bot, Classicalcon, Xevior, ClueBot, Justin W Smith, The Thing That Should Not Be, Kamath.nakul, Garyzx, Mqthihs, Alex.vatchenko, Bender2k14, IanS1967, Jonfore, Behrangsa, Sun Creator, 7, Bojan1989, Doriftu, XLinkBot, Marc van Leeuwen, Stickee, Brentsmith101, Airplaneman, Dawynn, LithiumBreather, Wælgæst wafre, NjardarBot, Sapeur, Nate Wessel, Mdnahas, Numb03-bot, AnotherPerson 2, Bluebusy, Aarslankhalid, Legobot, Yobot, Fraggle81, Mhayes46, AnomieBOT, Shinjixman, Materialscientist, Citation bot, Frozendice, Xqbot, Vegpuff, Wtarreau, Shmomuffin, FrescoBot, Mark Renier, Microbizz, Frankrod44, Ateeq.s, Encognito, RedBot, Wjomlex, SpaceFlight89, Sss41, MathijsM, Neomagic100, Trappist the monk, Shelbymoore3, Belovedeagle, Lotje, Dinamik-bot, Zvn, Kgautam28, Duoduo, Chicodroid, WillNess, Xmn, The tree stump, John of Reading, Werith, 15turnsm, Card Zero, Chris857, 28bot, Petrb, ClueBot NG, JimsMaher, Tdhsmith, Loopwhile1, Cj005257, Ontariolot, Joel B. Lewis, Widr, Helpful Pixie Bot, BG19bot, ISTB351, Metricopolus, Czar.pino, Manohar reddy123123, Happyuk, Crh23, Minsbot, Klilidiplomus, Mbdeir, Pratyya Ghosh, Rohitgoyal100, ChrisGualtieri, Gdevanla, YFdyh-bot, Padenton, Pp007, JYBot, Calvin zcx, Lone boatman, Josell2, Zziccardi, Jean-Baptiste Mestelan, Rahulgatm, Biplavd IIC, Julier, Joery2210, Magicguy11, Ryuunoshounen, AKS.9955, Demonmerlin, Iokevins, JMP EAX, Tcriss, GarlicPasra, Xanderlent and Anonymous: 406
- **Binary search tree** *Source:* <http://en.wikipedia.org/wiki/Binary%20search%20tree?oldid=660218874> *Contributors:* Damian Yerrick, Bryan Derksen, Taw, Mrwojo, Spiff~enwiki, PhilipMW, Michael Hardy, Booyabazooka, Nixdorf, Ixfd64, Minesweeper, Darkwind, LittleDan, Glenn, BAxelrod, Timwi, MatrixFrog, Dcoetzee, Havardk, Dysprosia, Doradus, Maximus Rex, Phil Boswell, Fredrik, Postdlf, Bkell, Hadal, Tobias Bergemann, Enochlau, Awu, Giftlite, P0nc, Ezhiki, Maximax, Qleem, Karl-Henner, Qiq~enwiki, Shen, Andreas Kaufmann, Jin~enwiki, Grunt, Kate, Oskar Sigvardsson, D6, Ilana, Kulp, ZeroOne, Vdm, Func, LeonardorGregianin, Nicolasbock, HasharBot~enwiki, Alansohn, Liao, RoySmith, Rudo.Thomas, Pion, Wtmitchell, Evil Monkey, 4c27f8e656bb34703d936fc59ede9a, Oleg Alexandrov, Mindmatrix, LOL, Oliphant, Ruud Koot, Trevor Andersen, GregorB, Mb1000, MrSomeone, Qwertyus, Nneonneo, Hathawayc, VKokielov, Ecb29, Mathbot, BananaLanguage, DevastatorIIC, Quuxplusone, Sketch-The-Fox, Butros, Banaticus, Roboto de Ajvol, YurikBot, Wavelength, Michael Slone, Hyad, Taejo, Gaius Cornelius, Oni Lukos, TheMandarin, Salrizvy, Moe Epsilon,

BOT-Superzerocool, Googl, Regnaron~enwiki, Abu adam~enwiki, Chery, Cedar101, Jogers, LeonardoRob0t, Richardj311, WikiWizard, SmackBot, Bernard François, Gilliam, Ohnoitsjamie, Theone256, Oli Filth, Neurodivergent, DHN-bot~enwiki, Alexsh, Garoth, Mweber~enwiki, Allan McInnes, Calbaer, NitishP, Cybercobra, Hcethatsme, MegaHasher, Breno, Nux, Beetstra, Dicklyon, Hu12, Vocabo, Konnetikut, JForget, James pic, CRGreathouse, Ahyl, WeggeBot, Mikeputnam, TrainUnderwater, Jdm64, AntiVandalBot, Jirka6, Lanov, Huttarl, Eapache, JAnDbot, Anoopjohnson, Magioladitis, Abednigo, Allstarecho, Tomt22, Gwern, S3000, MartinBot, Anaxial, Leyo, Mike.lifeguard, Phishman3579, Skier Dude, Joshua Issac, Mgjus, Kewlito, Danadocus, Vectorpaladin13, Labalius, BotKung, One half 3544, Spadgos, MclareN212, Nerdgerl, Rdemar, Davekaminski, Rhanekom, SieBot, YonaBot, Casted, VVVBot, Ham Pastrami, Jerryobject, Swapsy, Djcollom, Svick, Anchor Link Bot, GRHooked, Loren.wilton, Xevior, ClueBot, ChandlerMapBot, Madhan virgo, Theta4, Shailen.sobhee, AgentSnoop, Onomou, XLinkBot, WikHead, Metalmax, MrOllie, Jdurham6, Nate Wessel, LinkFA-Bot, میراند, Matekm, Legobot, Luckas-bot, Yobot, Dimchord, AnomieBOT, The Parting Glass, Burakov, Ivan Kuckir, Tbdm, LilHelpa, Capricorn42, SPTWriter, Doctordiehard, Wtarreau, Shmomuffin, Dzikasosna, Smallman12q, Kurapix, Adamuu, FrescoBot, 4get, Citation bot 1, Golle95, Aniskhan001, Frankrod44, Cochito~enwiki, MastiBot, Thesevenseas, Sss41, Vromascanu, Shuri org, Rolpa, Jayaneethatj, Avermapub, MladenWiki, Konstantin Pest, Akim Demaille, Cyc115, WillNess, Nils schmidt hamburg, RjwilmisBot, Ripchip Bot, X1024, ChibbyOne, Your Lord and Master, Nomen4Omen, Meng6, Wmawner, Tolly4bolly, Dan Wang, ClueBot NG, SteveAyre, Jms49, Frietjes, Ontariolot, Solsan88, Nakarumaka, BG19bot, AlanSherlock, Rafikamal, BPositive, RJK1984, Phc1, IgushevEdward, Hdanak, Jingguo Yao, Yaderbh, RachulAdmas, Frosty, Josell2, SchumacherTechnologies, Farazbhinder, Wulfskin, Embanner, Mtahmed, Jihlim, Kaidul, Cybdestroyer, Jabanabba, Gokayhuz, Mathgorges, Jianhui67, Super fish2, Ryuunoshounen, Dk1027, Azx0987, KH-1, Tshubham, HarshalVTripathi, ChaseKR, Filip Euler, Koolnik90, K-evariste, Enzoferber and Anonymous: 340

- **Self-balancing binary search tree** *Source: <http://en.wikipedia.org/wiki/Self-balancing%20binary%20search%20tree?oldid=644414730>* *Contributors:* Michael Hardy, Angela, Dcoetzee, Dysprosia, DJ Clayworth, Noldoaran, Fredrik, Diberri, Enochlau, Wolfkeeper, Jorge Stolfi, Neilc, Pgjan002, Jacob grace, Andreas Kaufmann, Shlomif, Baluba, Mdd, Alansohn, Jeltz, ABCD, Kdau, RJFJR, Japanese Searobin, Jacobolus, Chochopk, Qwertyus, Moskvax, Intgr, YurikBot, Light current, Plyd, MrDrBob, Cybercobra, Jon Awbrey, Ripe, Momet, Jafet, CRGreathouse, Cydebot, Widefox, David Eppstein, Funandtrvl, VolkovBot, Sriganeshs, Lamro, Jruderman, Plastikspork, SteveJothen, Addbot, Bluebusy, Yobot, Larrycz, Xqbot, Drilnoth, Steaphan Greene, FrescoBot, DrilBot, ActuallyRationalThinker, EmausBot, Lark-inzhang1993, Azuris, Andreas4965, Wolfgang42, Josell2, Jochen Burghardt, G PVIB and Anonymous: 43
- **Tree rotation** *Source: <http://en.wikipedia.org/wiki/Tree%20rotation?oldid=656648941>* *Contributors:* Mav, BlckKnght, B4hand, Michael Hardy, Kragen, Dcoetzee, Dysprosia, Altenmann, Michael Devore, Leonard G., Neilc, Andreas Kaufmann, Mr Bound, Chub~enwiki, BRW, Oleg Alexandrov, Joriki, Graham87, Qwertyus, Wizzar, Pako, Mathbot, Abarry, Trainra, Cedar101, SmackBot, DHN-bot~enwiki, Ramasamy, Kjkjava, Hyperionred, Thijs!bot, Headbomb, Waylonflinn, David Eppstein, Gwern, Vegasprof, STBotD, Skaraoke, Mtanti, SCriBu, Castorvx, Salvar, SieBot, Woblosch, Svick, Xevior, Boykobb, LaaknorBot, چاره, Legobot, Mangarah, LilHelpa, GrouchoBot, Adamuu, Citation bot 1, Alexey.kudinkin, ClueBot NG, Knowledgeofthekrell, Josell2, Explorer512, Tar-Elessar, Javier Borrego Fernandez C-512 and Anonymous: 38
- **Weight-balanced tree** *Source: <http://en.wikipedia.org/wiki/Weight-balanced%20tree?oldid=659939980>* *Contributors:* Ciphergoth, Andreas Kaufmann, Etrigan, Qwertyus, Bhadani, Snarius, Reyk, DVD R W, The former 134.250.72.176, Norm mit, Zureks, Pascal.Tesson, KenE, Jointds, JaGa, Addbot, Bobmath and Anonymous: 12
- **Threaded binary tree** *Source: <http://en.wikipedia.org/wiki/Threaded%20binary%20tree?oldid=649079825>* *Contributors:* Michael Hardy, Andreas Kaufmann, R. S. Shaw, Mr2001, Pearle, MoraSique, Qwertyus, Grafen, Ragzouken, SmackBot, Adam murgittroyd, Konstantin Veretennicov, Edlee, Brad101, Headbomb, Ppelleti, MER-C, Hut 8.5, A3nm, Themania, LokiClock, Sdenn, Flyer22, Svick, Denisarona, ImageRemovalBot, Jaccos, Jagat sastry, Moberg, Addbot, Yobot, Amicon, AnomieBOT, Xqbot, Ansumang, Yeskw, Alph Bot, John of Reading, Donner60, ClueBot NG, Sunnygunner, Vaibhavvc1092, Atharvaborkar, Sanju2193p, Ujomin, Kishu Agarwal, Jimten and Anonymous: 35
- **AVL tree** *Source: <http://en.wikipedia.org/wiki/AVL%20tree?oldid=660763779>* *Contributors:* Damian Yerrick, BlckKnght, M~enwiki, FvdP, Infrogmation, Michael Hardy, Nixdorf, Minesweeper, Jll, Poor Yorick, Dcoetzee, Dysprosia, Doradus, Greenrd, Topbanana, Noldoaran, Fredrik, Altenmann, Merovingian, Tobias Bergemann, Andrew Weintraub, Mckaysisbury, Neilc, Pgjan002, Tsemii, Andreas Kaufmann, Safety Cap, Mike Rosoft, Guanabot, Byrial, Pavel Vozenilek, Shlomif, Lankiveil, Rockslave, Smalljim, Geek84, Axe-Lander, Darangho, Kjkolb, Larry V, Obradovic Goran, HasharBot~enwiki, Orimosenzon, Kdau, Docboat, Evil Monkey, Tphyahoo, RJFJR, Kenyon, Oleg Alexandrov, LOL, Ruud Koot, Gruu, Seyer, Graham87, Qwertyus, Toby Douglass, Mikm, Alex Kapranoft, Jeff02, Gurch, Intgr, Chobot, YurikBot, Gaius Cornelius, NawlinWiki, Astral, Dtrebbien, Kain2396, Bkil, Pnorcks, Blackllotus, Bota47, Lt-wiki-bot, Cedar101, KGasso, Gulliveig, Danielx, LeonardoRob0t, Paul D. Anderson, SmackBot, Apanag, InverseHypercube, David.Mestel, KocjoBot~enwiki, Gilliam, Tsof, DHN-bot~enwiki, ChrisMP1, Tamfang, Cybercobra, Flyingspuds, Epachamo, Philvarner, Dcamp314, Kuru, Euchiasmus, Michael miceli, Caviare, Dicklyon, Yksyksyks, Momet, Nysin, Jac16888, Daewoollama, Cyhawk, ST47, Zian, Joeadams, Msanchez1978, Eleuther, AntiVandalBot, Ste4k, Jirka6, Gökhan, JAnDbot, Leuko, Magioladitis, Anant sogani, Avicennasis, Nguyễn Hữu Dũng, MartinBot, J.delanoy, Pedrito, Phishman3579, Jeepday, Michael M Clarke, UnwashedMeme, Binnacle, Adamd1008, DorganBot, Hwbehrens, Funandtrvl, BenBac, VolkovBot, Indubitably, Mtanti, Castorvx, AlexGreat, Uw.Antony, Enviroboy, Srivesh, SieBot, Aent, Vektor330, Flyer22, Svick, Mauritsmaartendejong, Denisarona, Xevior, ClueBot, Nnemo, CounterVandalismBot, Autof06, Kukolar, Ksulli10, Moberg, Mellerho, XLinkBot, Gnowor, Resper~enwiki, DOI bot, Dawynn, Ommiy-Pangaeus~enwiki, Leszek Jańczuk, Mr.Berna, West.andrew.g, Tide rolls, Bluebusy, MattyIX, Legobot, Luckas-bot, Yobot, II MusLiM HyBRID II, Agrawalyogesh, Jim1138, Kingpin13, Materialscientist, Xqbot, Drilnoth, Oliversisson, VladimirReshetnikov, Greg Tyler, Shmomuffin, Adamuu, Mjkoo, FrescoBot, MarkHeily, Mohammad ahad, Ichimonji10, DrilBot, Sebculture, RedBot, Trappist the monk, MladenWiki, EmausBot, Benoit fraikin, Mzruya, Iamnitin, AvicBot, Vlad.c.manea, Nomen4Omen, Geoff55, Mnogo, Compusense, ClueBot NG, Bulldog73, G0gogsc300, Codingrecipes, Helpful Pixie Bot, Titodutta, BG19bot, Northamerica1000, Ravitkhorana, DmitriyVilkov, Zhaofeng Li, ChrisGualtieri, Eta Aquariids, Dexbot, Kushalbiswas777, CostinulAT, Akerbos, Josell2, Jochen Burghardt, G PVIB, Elfbw, Ppkhoa, Yelnatz, Hibbarnt, Jpopesculian, Skr15081997, Devsathish, Aviggiano, Eeb379, Monkbot, HexTree, Badidipedia, Esquivalience, StudentOfStones and Anonymous: 329
- **Red-black tree** *Source: [http://en.wikipedia.org/wiki/Red%20black\\_tree?oldid=661013700](http://en.wikipedia.org/wiki/Red%20black_tree?oldid=661013700)* *Contributors:* Dreamyshade, Jzcool, Ghakko, FvdP, Michael Hardy, Blow~enwiki, Minesweeper, Ahoerstemeier, Cyp, Strebe, Jerome.Abel, Notheruser, Kragen, Ghewgill, Timwi, MatrixFrog, Dcoetzee, Dfeuer, Dysprosia, Hao2lian, Shizhao, Phil Boswell, Robbot, Fredrik, Altenmann, Humpback~enwiki, Jleedev, Tobias Bergemann, Enochlau, Connelly, Giftlite, Sepreece, BenFrantzDale, Brona, Dratman, Leonard G., Pgjan002, LiDaobing, Sebbe, Karl-Henner, Andreas Kaufmann, Tristero~enwiki, Spundun, Will2k, Aplusbi, SickTwist, Giraffedata, Ryan Stone, Zeta-woof, Iav, Hawke666, Cjcollier, Fawcett5, Denniss, Cburnett, RJFJR, H2g2bob, Kenyon, Silverdirk, Joriki, Mindmatrix, Merlinme, Ruud Koot, Urod, Gimbo13, Jtsiomb, Marudubshinki, Graham87, Qwertyus, OMouse, Drebs~enwiki, Rjwilmis, Hgkamath, ErikHauen, Toby Douglass, SLi, FlaBot, Margosbot~enwiki, Fragglet, Jameshfisher, Loading, SGreen~enwiki, YurikBot, Wavelength, Jengelh,

Rsrikanth05, Bovineone, Sesquiannual, Jaxl, Dlugosz, Coderzombie, Mikeblas, Blackllotus, Schellhammer, Regnaron~enwiki, Ripper234, JMBucknall, Lt-wiki-bot, Abu adam~enwiki, Smilindog2000, SmackBot, Pgk, Thumperward, Silly rabbit, DHN-bot~enwiki, Sct72, Khalil Sawant, Xiteer, Cybercobra, Philvarner, TheWarlock, Drak~enwiki, SashatoBot, Mgrand, N3bulous, Bezenek, Caviare, Dicklyon, Bellfry, Pqrstuv, Pranith, Supertigerman, Ahy1, Jodawi, Pmussler, Linuxrocks123, Epbr123, Ultimus, Abloomfi, Headbomb, AntiVandalBot, Hermel, Roleplayer, .anacondabot, Stdazi, David Eppstein, Lunakeet, Gwern, MartinBot, Glrx, Themania, IDogbert, Madhurantwani, Phishman3579, Warut, Smangano, Binnacle, Lukax, Potatoswater, KylieTastic, Bonadea, Funandtrvl, DoorsAjar, Jozue, Simoncropp, Laurier12, Bioskope, Yakov1122~enwiki, YonaBot, Sdenn, Stone628, Stanislav Nowak~enwiki, AlanUS, Hariva, Shyammurarka, Xevior, Uncle Milt, Nanobear~enwiki, Xmarios, Karlhendrikse, Kukolar, MiniStephan, Uniwalk, Versus22, Johnnuniq, Consed, C. A. Russell, Addbot, Joshhibschnan, Fcp2007, AgadaUrbanit, Tide rolls, Lightbot, Luckas-bot, Yobot, Fraggie81, AnomieBOT, Narlami, Cababunga, Maxis ftw, ChrisCPearson, Storabled, Zehntor, Tbvd, Xqbot, Nishantjr, RobotBOT, Kyle Hardgrave, Adamuu, FrescoBot, AstaBOT15, Karakak, Kmels, Banej, Userask, Hnn79, Gnathan87, MladenWiki, Pellucide, Belovedeagle, Patmorin, Sreeakshay, EmausBot, John of Reading, Hugh Aguilar, K6ka, Nomen4Omen, Mnogo, Awakenrz, Card Zero, Grandphuba, Kapil.xerox, Wikipedian to the max, 28bot, ClueBot NG, Xjianz, Wittjeff, Ontariolot, Widr, Hagoth, Pratyya Ghosh, Deepakabhyankar, Naxik, Dexbot, JingguoYao, Akerbos, Epicgenius, Weishi Zeng, Suelru, Monkbot, Spasticcodemonkey, Freitafr, Demagur and Anonymous: 301

- **AA tree** *Source:* <http://en.wikipedia.org/wiki/AA%20tree?oldid=655652247> *Contributors:* Damian Yerrick, FvdP, Charles Matthews, Dcoetzee, Rkleckner, Gazpacho, Qutezuce, Mecki78, Goochelaar, Triona, Nroets, RJFJR, Firsfron, MONGO, Qwertyus, Ash211, Koavf, Darguz Parsilvan, RobertG, Confuzzled, SmackBot, Optikos, Cybercobra, CmdrObot, Lurlock, Cydebot, Ppelleti, HyperQuantum, Gwern, Funandtrvl, Bloodyberry, Bporopat, Why Not A Duck, Stanislav Nowak~enwiki, Kukolar, Addbot, LinkFA-Bot, Balabiot, AliasXYZ, LucienBOT, Mu Mind, QjOn, Vromascanu, Vladislav.kuzkokov, Mnogo, Ijungreis, Asavageiv and Anonymous: 38
- **Scapegoat tree** *Source:* <http://en.wikipedia.org/wiki/Scapegoat%20tree?oldid=653433715> *Contributors:* FvdP, Edward, Dcoetzee, Ruakh, Dbenbenn, Tweenk, Sam Hocevar, Andreas Kaufmann, Rich Farmbrough, Jarsyl, Aplusbi, Oleg Alexandrov, Firsfron, Slike2, Qwertyus, Mathbot, Wknight94, SmackBot, Chris the speller, Cybercobra, MegaHasher, Vanisaac, AbsolutBildung, Thijssbot, Robert Ullmann, Themania, Danadocus, Joey Parrish, WillUther, Kukolar, SteveJothen, Addbot, Yobot, Citation bot, C.hahn, Patmorin, WikitanvirBot, Hankjo, Mnogo, ClueBot NG, AlecTaylor, Theemathas and Anonymous: 35
- **Splay tree** *Source:* <http://en.wikipedia.org/wiki/Splay%20tree?oldid=651054356> *Contributors:* Mav, BlckKngh, Xaonon, Christopher Mahan, FvdP, Edward, Michael Hardy, Nixdorf, Pnm, Drz~enwiki, Dcoetzee, Dfeuer, Dysprosia, Silvonen, Tjdw, Phil Boswell, Fredrik, Stephan Schulz, Giftlite, Wolfkeeper, CyborgTosser, Lqs, Wiml, Gscshoyru, Urhixidur, Karl Dickman, Andreas Kaufmann, Yonkeltron, Rich Farmbrough, Qutezuce, Aplusbi, Chbars, Tabletop, VsevolodSipakov, Graham87, Qwertyus, Rjwilmsi, Pako, Ligulem, Jameshfisher, Fresheneesz, Wavelength, Vecter, Romanc19s, Dlugosz, Abu adam~enwiki, Terber, HereToHelp, That Guy, From That Show!, SmackBot, Honza Záruba, Unyoyega, Apankrat, Silly rabbit, Octahedron80, Axlape, OrphanBot, Cybercobra, Philvarner, Ohconfucius, MegaHasher, Vanished user 9i39j3, Jamie King, Dicklyon, Freeside3, Martlau, Momet, Ahy1, VTBassMatt, Escarbot, Atavi, Coldzero1120, Eapache, KConWiki, David Eppstein, Ahmad87, Gwern, HPRappaport, Foober, Phishman3579, Dodno, Funandtrvl, Anna Lincoln, Rhanekom, Zuphilip, Russelj9, Svick, AlanUS, Nanobear~enwiki, Pointillist, Safek, Kukolar, XLinkBot, Dekart, Maverickwoo, Addbot, ט „ 717, Legobot, Yobot, Roman Munich, AnomieBOT, Erel Segal, 1exec1, Josh Guffin, Citation bot, Winniehell, Shmomuffin, Dzikasosna, FrescoBot, Snieltfeld, Citation bot 1, Jwillia3, Zeitfree, Sss41, MladenWiki, Sihag.deepak, Ybungabill, Crimer, Wyverald, Const86, EmausBot, Hannan1212, SlowByte, Mnogo, P2004a, Petrb, ClueBot NG, Wiki.ajaygautam, SteveAyre, Ontariolot, Antiqueight, Vagobot, Arunshankarb, Harijec, HueSatLum, FokkoDriesprong, Makecat-bot, Arunkumar nonascii, B.pradeep143, MazinIssa, Abc00786, Lf-barba, Craftbondpro, Mdburns, Fabio.pakk, BethNaught, Efortanely, BenedictEggers, Admodi and Anonymous: 129
- **T-tree** *Source:* <http://en.wikipedia.org/wiki/T-tree?oldid=650106533> *Contributors:* Damian Yerrick, Bryan Derksen, FvdP, Charles Matthews, Psychonaut, GreatWhiteNortherner, Acm, Andreas Kaufmann, Rich Farmbrough, Qwertyus, WriterHound, Gaius Cornelius, Dlugosz, Hholzgra, SmackBot, Gilliam, OrphanBot, Fuhghettaboutit, Cybercobra, Bezenek, Jamie Lokier, TAnthony, Hmendonca, Paxcoder, Jonah.harris, Ted nw, The Thing That Should Not Be, Kroepke, Alexbot, Addbot, Luckas-bot, Yobot, Xqbot, Omnipaedista, DrilBot, RjwilmsiBot, Mnogo, Snotbot, 220 of Borg, Nitrocaster (usurped), Monkbot and Anonymous: 19
- **Rope (computer science)** *Source:* [http://en.wikipedia.org/wiki/Rope\\_\(data\\_structure\)?oldid=636579284](http://en.wikipedia.org/wiki/Rope_(data_structure)?oldid=636579284) *Contributors:* Mrwojo, Dcoetze, Doradus, Wtanaka, Pengo, Tobias Bergemann, DavidCary, Andreas Kaufmann, Kimbly, CanisRufus, Spoon!, Vdm, Daira Hopwood, Qwertyus, Gringer, Mstroec, Emersoni, SteveWitham, Snaix920, SmackBot, Thumperward, Cybercobra, Almkglor, Smith609, Limaner, Vanisaac, Jokes Free4Me, Eric Le Bigot, Alaibot, Headbomb, Lfstevens, Ubershmekel, Chkno, Fridolin, Redundance, Jacroe, Dhruvbird, FireyFly, ArtemGr, Justin545, AzraelUK, Dsimic, Addbot, DOI bot, Luckas-bot, AnomieBOT, Alvin Seville, Mfwitten, Ybungabill, Preetum, ClueBot NG, Trivialsegfault, Meng Yao, Fwielst, BattyBot, Frosty, Daiminya, Conceptualizing and Anonymous: 33
- **Top Tree** *Source:* [http://en.wikipedia.org/wiki/Top\\_tree?oldid=632557835](http://en.wikipedia.org/wiki/Top_tree?oldid=632557835) *Contributors:* Pgan002, Rjwilmsi, Chris the speller, Darren Strash, The.lorrr, Crown Prince, Hippo.69, GrantWishes and Anonymous: 3
- **Tango tree** *Source:* <http://en.wikipedia.org/wiki/Tango%20tree?oldid=640979156> *Contributors:* AnonMoos, Giraffedata, RHaworth, Qwertyus, Rjwilmsi, Vecter, Jengelh, Grafen, Malcolma, Rayhe, SmackBot, C.Fred, Chris the speller, Iridescent, Alaibot, Headbomb, Nick Number, Acroterion, Nyttend, Philg88, Inomyabcs, ImageRemovalBot, Sfan00 IMG, Nathan Johnson, Jasper Deng, Yobot, AnomieBOT, Erel Segal, Anand Oza, FrescoBot, Σ, RenamedUser01302013, Card Zero, Ontariolot, Do not want, Tango tree, DoctorKubla and Anonymous: 17
- **Van Emde Boas tree** *Source:* <http://en.wikipedia.org/wiki/Van%20Emde%20Boas%20tree?oldid=651137645> *Contributors:* B4hand, Michael Hardy, Kragen, Charles Matthews, Dcoetze, Doradus, Phil Boswell, Dbenbenn, Bender235, BACbKA, Nickj, Qwertyus, Rjwilmsi, Jeff02, Quuxplusone, Fresheneesz, Argav, Piet Delport, Cedar101, Gulliveig, SmackBot, Cybercobra, A5b, David Cooke, Neelix, Cydebot, Cyhawk, Snoopy67, David Eppstein, Panarchy, Dangercrow, Svick, Adrianwn, Kaba3, Addbot, Lightbot, Luckas-bot, Yobot, Fx4m, Mangarah, Brutaldeluxe, Patmorin, Gailcarmichael, EmausBot, Jackrae, ElhamKhodaee, MatthewIreland, Theemathas, RandomSort, Peter238 and Anonymous: 22
- **Cartesian tree** *Source:* <http://en.wikipedia.org/wiki/Cartesian%20tree?oldid=654065272> *Contributors:* Giftlite, Andreas Kaufmann, Tabletop, GrEp, Cybercobra, HasanDiwan, David Eppstein, Bbi5291, Cobi, Pichpitch, Addbot, Citation bot, Citation bot 1, RjwilmsiBot, ChuispastonBot and Anonymous: 6
- **Treap** *Source:* <http://en.wikipedia.org/wiki/Treap?oldid=656576249> *Contributors:* Edward, Poor Yorick, Jogloran, Itai, Jleedev, Eequor, Andreas Kaufmann, Qef, Milkmandan, Saccade, Wsloand, Oleg Alexandrov, Jörg Knappen~enwiki, Ruud Koot, Hdante, Behdad, Qwertyus, Arbor, Gustavb, Regnaron~enwiki, James.nvc, SmackBot, KnowledgeOfSelf, Chris the speller, Cybercobra, MegaHasher, Pfh, Yzt, Jsaxton86, Cydebot, Blaisorblade, Escarbot, RainbowCrane, David Eppstein, AHMartin, Bajsejohannes, Justin W Smith, Kukolar, Hans Adler, Addbot, Luckas-bot, Yobot, Erel Segal, Rubinbot, Citation bot, Benemq, Gilo1969, Miym, Brutaldeluxe, Cshinyee, C02134, ICEAGE, MaxDel, Patmorin, Cdb273, MoreNet, ChuispastonBot, Chmarkine, Naxik, Lsmll and Anonymous: 28

- **B-tree** *Source:* <http://en.wikipedia.org/wiki/B-tree?oldid=661328752> *Contributors:* Kpjäs, Bryan Derksen, FvdP, Mrwojo, Spiff~enwiki, Edward, Michael Hardy, Rp, Chadloder, Minesweeper, JWSchmidt, Ciphergoth, BAxelrod, Alaric, Charles Matthews, Dcoetzee, Dysprosia, Hao2lian, Ed g2s, Tjdw, AaronSw, Carbuncle, Wtanaka, Fredrik, Altenmann, Liotier, Bkell, Dmn, Tobias Bergemann, Giflite, Uday, Wolfkeeper, Lee J Haywood, Levin, Curps, Joconnor, Ketil, Jorge Stolfi, AlistairMcMillan, Nayuki, Neilc, Pgdn002, Gdr, Cbraga, Knutux, Stephan Leclercq, Peter bertok, Andreas Kaufmann, Chmod007, Kate, Ta bu shi da yu, Slady, Rich Farmbrough, Guanabot, Leibniz, Qutezuce, Talleean, Slike, Dpotter, Mrnaz, SickTwist, Wipe, R. S. Shaw, HasharBot~enwiki, Alansohn, Anders Kaseorg, ABCD, Wtmitchell, Wsloand, MIT Trekkie, Kinema, Postrach, Mindmatrix, Decrease789, Ruud Koot, Qwertyus, FreplySpang, Rjwilmsi, Kinu, Strake, Sandman@llgp.org, FlaBot, Psyphe, Ysangkok, Fragglet, Joe07734, Makku, Fresheneesz, Antimatter15, CiaPan, Daev, Chobot, Vyrograph, YurikBot, Bovineone, Ethan, PrologFan, Mikeblas, EEMIV, LeonardoRob0t, SmackBot, Cutter, Ssbohio, Btwied, Dany-luis, Mhss, Chris the speller, Bluebot, Oli Filth, Malbrain, Stevemidgley, Cybercobra, AlyM, Jeff Wheeler, Battamer, Ck lostsword, Zearin, Bezenek, Flying Bishop, Loadmaster, Dicklyon, P199, Inquisitus, Norm mit, Noodlez84, Lamdk, FatalError, Ahy1, Aubrey Jaffer, Beeson, Cydebot, PKT, ContivityGoddess, Headbomb, I do not exist, Alfalfahotshots, AntiVandalBot, Luna Santin, Widefox, Jirka6, Lfsteven, Lklundin, The Fifth Horseman, MER-C, .anacondabot, Nyq, Yakushima, Hbent, MoAgnome, Ptheoch, CarlFeynman, Glrx, Trusilver, Altes, Phishman3579, Jy00912345, Priyank bolia, GoodPeriodGal, DorganBot, MartinRinehart, Michael Angelkovich, VolkovBot, Oshwah, Appoose, Kovianyo, Don4of4, Dlae, Jesin, Billingham, Uw.Antony, Aednichols, Joahnnes, Ham Pastrami, JCLately, Jojalozzo, Ctxppc, Dravecky, Anakin101, Hariva, Wantnot, ClueBot, Rpajares, Simon04, Junk98df, Abrech, Kukolar, Johannes Animous, XLinkBot, Paushali, Addbot, CanadianLinuxUser, AnnaFrance, LinkFA-Bot, Jjdawson7, Verbal, Lightbot, Krano, Teles, Twimoki, Luckas-bot, Quadrescence, Yobot, AnomieBOT, Gptelles, Materialscientist, MorgothX, Xtremejamies183, Xqbot, Nishantjr, Matttoothman, Sandeep.a.v, Merit 07, Almabot, GrouchoBot, Eddvella, January2009, Jacosi, SirSeal, Hobsonlane, Bladefistx2, Mfwitten, Redrose64, Fgdafsdgfdsgfd, Trappist the monk, Patmorin, RjwilmsiBot, MachineRebel, John lindgren, DASHBot, Wkailey, John of Reading, Wout.mertens, John ch fr, Psychobobbins, Ctail, Fabriciodosanjossilva, TomYHChan, Mnogo, NGPriest, Tuolumne0, ClueBot NG, Betzaar, Oldsharp, Widr, DanielKlein24, Bor4kip, RMcPhillip, Meurondb, BG19bot, WinampLlama, Cp3149, Andytwigg, JoshuSasori, YFdyhbot, Dexbot, Seanhalle, Lsmll, Tentinator, M Murphy1993, Skr15081997, IvayloS and Anonymous: 363
- **B+ tree** *Source:* <http://en.wikipedia.org/wiki/B%2B%20tree?oldid=661089697> *Contributors:* Bryan Derksen, Cherezov, Tim Starling, Pnm, Eurleif, CesarB, Marc omorain, Josh Cherry, Vikreykja, Lupo, Dmn, Giflite, Inkling, WorldsApart, Neilc, Lightst, Arrow~enwiki, WhiteDragon, Two Bananas, Scrool, Leibniz, Zenohockey, Nyenyc, Cmdrjameson, TheProject, Obradovic Goran, Happyvalley, Mdd, Arthena, Yaml, TZOTZIOY, Knutties, Oleg Alexandrov, RHaworth, LrdChaos, LOL, Decrease789, GregorB, PhilippWeissenbacher, Ash211, Penumbra2000, Gurch, Degeberg, Intr, Fresheneesz, Chobot, Bornhj, Encyclops, Bovineone, Capi, Luc4~enwiki, Mikeblas, Foeckler, Snarius, Cedar101, LeonardoRob0t, Jbalint, Jsnx, Arny, DomQ, Mhss, Hongooi, Rrburke, Cybercobra, Itmozart, Nat2, Leksey, Tlesher, Julthe, Cychoi, UncleDoggie, Yellowstone6, Ahy1, Unixguy, CmdrObot, Leijohn, Jwang01, Ubuntu2, I do not exist, Nu-world, Ste4k, JAnDbot, Txomin, CommonsDelinker, Garcida5658, Afaviram, Mfedyk, Priyank bolia, Mqchen, Mrcowden, VolkovBot, OliviaGuest, Mdmkolbe, Muro de Aguas, Singaldruv, Highlandsun, SheffieldSteel, MRLacey, S.Örvarr.S, SieBot, Tresiden, YonaBot, Yungoe, Amaravashishth, Mogentianae, Imachchu, ClueBot, K14m, Boing! said Zebedee, Tuxthepenguin933, SchreiberBike, Max613, Raatikka, Addbot, TutterMouse, Thunderpenguin, Favonian, AgadaUrbanit, Matěj Grabovský, Bluebusy, Twimoki, Luckas-bot, Matthew D Dillon, Yobot, ColinTempler, Vevek, AnomieBOT, Materialscientist, LilHelpa, Nishantjr, Nqzero, Ajarov, Mydimle, Pinethicket, Eddie595, Reaper Eternal, MikeDierken, Holy-foek, Kastauyra, Igor Yalovecky, EmausBot, Immunize, Wout.mertens, Tommy2010, K6ka, James.walmsley, Bad Romance, Fabrictramp(public), QEDK, Ysoroka, Grundprinzip, ClueBot NG, Vedantkumar, MaximalIdeal, Anchor89, Giovanni Kock Bonetti, BG19bot, Lowercase Sigma, Chmarkine, NorthernSilencer, Michaelcomella, AshishMb2012, Perkinsb1024, EvergreenFir, Alexjlockwood, Graham477, Andylamp and Anonymous: 232
- **Dancing tree** *Source:* <http://en.wikipedia.org/wiki/Dancing%20tree?oldid=627523176> *Contributors:* Zeno Gantner, Inkling, AlistairMcMillan, Andreas Kaufmann, DanielCD, Qutezuce, ArnoldReinhold, Koody~enwiki, TheParanoidOne, Qwertyus, Rjwilmsi, Bsiegel, Ronodch, YurikBot, FrenchIsAwesome, Dlugosz, SmackBot, PeterSymonds, Computer Guru, Audriusa, Royboycrashfan, Cybercobra, 16@r, Clarityfiend, Cydebot, WhatamIdoing, Addbot, Xqbot, Mats33, H3llBot and Anonymous: 9
- **2-3 tree** *Source:* [http://en.wikipedia.org/wiki/2%2E2%80%933\\_tree?oldid=646935174](http://en.wikipedia.org/wiki/2%2E2%80%933_tree?oldid=646935174) *Contributors:* Robbot, Altenmann, Utcursch, Slady, Curtis14, Wsloand, Qwertyus, Awotter, Bgwhite, DGaw, SmackBot, Apanag, BiT, Bluebot, Cybercobra, Dicklyon, Jodawi, Asenine, Japo, ABF, TXiKiBoT, SieBot, Diaa abdelmoneim, Addbot, LaaknorBot, Luckas-bot, Yobot, E 3521, Yurymik, GrouchoBot, Chrismiceli, Adlerbot, AmphBot, Σ, Cpy.Prefers.You, EmausBot, V2desco, ZéroBot, Usability, ChuispastonBot, Jochen Burghardt, Xxiggy and Anonymous: 32
- **2-3-4 tree** *Source:* [http://en.wikipedia.org/wiki/2%2E2%80%933%2E2%80%934\\_tree?oldid=655298646](http://en.wikipedia.org/wiki/2%2E2%80%933%2E2%80%934_tree?oldid=655298646) *Contributors:* Ghakko, Paul Ebermann, OliD~enwiki, Jogloran, Ashdurbat, DHN, Michael Devore, Talrias, Andreas Kaufmann, Slady, Mdd, ABCD, Blahedo, Heida Maria, Drbreznjev, Ruud Koot, Twthmoses, Marudubshink, Qwertyus, Bgwhite, Jengelh, Goffrie, Schellhammer, AceDevil, Oli Filth, Dfletter, Cybercobra, Dicklyon, Jodawi, Andmatt, Alfalfahotshots, GromXXVII, Jhm15217, David Eppstein, Aopf, Ericwstark, ImageRemovalBot, Davidsawyer1, Jvdries, Cmuhlenk, Apham, WikHead, Addbot, Yobot, Terronis, Citation bot, Xqbot, Nishantjr, HostZ, Chrismiceli, Rabiddog51sb, Patmorin, Dadr, KristofersC, Davkutalek, Helpful Pixie Bot, BG19bot, Sbanskota08, AbdullahBinghunaiem, Waveletus, Arghyadeb, Malignor, TheGreatAvatar and Anonymous: 62
- **Queaps** *Source:* <http://en.wikipedia.org/wiki/Queap?oldid=607753106> *Contributors:* Michael Hardy, Pgdn002, Qwertyus, Rjwilmsi, Cydebot, David Eppstein, UnicornTapestry, DexDor and Cerabot~enwiki
- **Fusion tree** *Source:* <http://en.wikipedia.org/wiki/Fusion%20tree?oldid=585893722> *Contributors:* Edemaine, CesarB, Charles Matthews, Dcoetzee, ZeroOne, Oleg Alexandrov, SmackBot, Cybercobra, Cydebot, Alaibot, Nick Number, David Eppstein, Lamro, Czcollier, Decoratrix, Gmharhar, Vladfi and Anonymous: 6
- **Bx-tree** *Source:* <http://en.wikipedia.org/wiki/Bx-tree?oldid=646425700> *Contributors:* Tobias Bergemann, GregorB, Qwertyus, Rjwilmsi, Cheesewire, SmackBot, Cybercobra, CmdrObot, JohnCD, MC10, OrenBochman, Magioladitis, Mild Bill Hiccup, UnCatBot, 5 albert square, Twimoki, Bxtree, Chensu, Rats Chase Cats, ClueBot NG, MaricaBog and Anonymous: 7
- **Heap (data structure)** *Source:* [http://en.wikipedia.org/wiki/Heap%20\(data%20structure\)?oldid=661292903](http://en.wikipedia.org/wiki/Heap%20(data%20structure)?oldid=661292903) *Contributors:* Derek Ross, LC~enwiki, Christian List, Boleslav Bobcik, DrBob, B4hand, Frecklefoot, Paddu, Jimfbleak, Notheruser, Kragen, Jll, Aragorn2, Charles Matthews, Timwi, Dcoetzee, Dfeuer, Dysprosia, Doradus, Jogloran, Shizhao, Cannona, Robbot, Noldoaran, Fredrik, Sbisolo, Vikingstad, Giflite, DavidCary, Wolfkeeper, Mellum, Tristaneid, Pgdn002, Beland, Two Bananas, Pinguin.tk~enwiki, Andreas Kaufmann, Abdull, Oskar Sigvardsson, Wiesmann, Yuval madar, Qutezuce, Tristan Schmelcher, Ascánder, Iron Wallaby, Spoon!, Mdd, Musiphil, Guy Harris, Sligocki, Suruena, Derbeth, Wsloand, Oleg Alexandrov, Mahanga, Mindmatrix, LOL, Prophile, Daira Hopwood, Ruud Koot, Apokrif, Tom W.M., Graham87, Qwertyus, Drpaule, Psyphe, Mathbot, Quuxplusone, Krun, Fresheneesz, Chobot, YurikBot, Wavelength, RobotE,

Vector, NawlinWiki, DarkPhoenix, B7j0c, Moe Epsilon, Mlouns, LeoNerd, Bota47, Schellhammer, Lt-wiki-bot, Abu adam~enwiki, Ketil3, HereToHelp, SmackBot, Reedy, Tgdwyer, Eskimbot, Took, Thumperward, Oli Filth, Silly rabbit, Nbarth, Ilyathemuromets, Jmn-batista, Cybercobra, Mlpkr, Prasi90, Itmozart, Atkinson 291, NinjaGecko, SabbeRubbish, Loadmaster, Hiiiiiiiiiiiiiiiiiiii, Jurohi, Jafet, Ahy1, Eric Le Bigot, Flamholz, Cydebot, Max sang, Christian75, Grubbiv, Thijs!bot, OverLeg, Ablonus, BMB, Plaga701, Jirkaf, Mgioladitis, 28421u2232nfencenc, David Eppstein, Inhumandecency, Kibiru, Bradgib, Andre.holzner, Jfroelich, Public Menace, Cobi, STBotD, Cool 1 love, VolkovBot, JhsBot, Wingedsubmariner, Rhanekom, Quietbritishjim, SieBot, Ham Pastrami, Svick, Jonlandrum, Ken123BOT, AncientPC, Startswithth, ClueBot, Garyzx, Uncle Milty, Bender2k14, Kukolar, Xcez-be, Addbot, Psyced, Nate Wessel, Chzz, Numbo3-bot, Konryd, Chipchap, Bluebusy, Luckas-bot, Timeroot, KamikazeBot, DavidHarkness, AnomieBOT, Alwar.sumit, Jim1138, Burakov, ArthurBot, DannyAsher, Xqbot, Control.valve, GrouchoBot, Лев Дубовой, Mcmlxxx1, Kxx, C7protal, Mark Renier, Wikatmino, Sae1962, Gruntler, AaronEmi, ImPerfection, Patmorin, CobraBot, Akim Demaille, Stryder29, RjwilmsiBot, EmausBot, John of Reading, WikitanvirBot, Sergio91pt, Hari6389, Ernishman, Jaseemabd, Chris857, ClueBot NG, Manizzle23, Incompetence, Softsundude, Joel B. Lewis, Samuel Marks, Mediator Scientiae, Racerdogjack, Chmarkine, Hadi Payami, PatheticCopyEditor, Hupili, ChrisGualtieri, Rarkenin, Frosty, Clevera, FenixFeather, P.t-the.g, Theemathas, Sunny1304, Tim.sebring, Ginsuloft, Azx0987, Chaticramos, Evhonz, Nbro, Sequoia 42 and Anonymous: 169

Kwamikagami, Diomidis Spinellis, EmilJ, Shoujun, Giraffedata, Hugewolf, CyberSkull, Diego Moya, Loreto~enwiki, Stillnotelf, Velella, Blahedo, Runtime, Tr00st, Gmaxwell, Simetrical, MattGiua, Gerbrant, Graham87, BD2412, Qwertyus, Rjwilmsi, Drpaul, Sperxios, Me and, Piet Delport, Dantheon, Gaius Cornelius, Nad, Mikeblas, Danielx, TMott, SmackBot, Slamb, Honza Záruba, Karl Stroetmann, Jim baker, BiT, Ennorehling, Eug, Neurodivergent, MalafayaBot, Drewnoakes, Otus, Malbrain, Kaimiddleton, Cybercobra, Leaflord, ThePianoGuy, Denshade, Edlee, Johnny Zoo, MichaelPloujnikov, Cydebot, Electrum, Farzaneh, Bsdaemon, Deborahjay, Headbomb, Widefox, Maged918, KMeyer, Nosbig, Deflective, Raanoo, Ned14, David Eppstein, FuzziusMaximus, Micahcowan, Francis Tyers, Pavel Fus, 97198, Dankogai, Funandtrvl, Bse3, Kyle the bot, Nissenbenyitskhak, Jmacglashan, C0dergirl, Sergio01, Ham Pastrami, Enrique.benimeli, Svick, AlanUS, Jldwig, Startswithj, Anupchowdry, Para15000, Pombredanne, JeffDonner, Estirabot, Stephengmatthews, Johnuniq, Dscholte, XLinkBot, Dsimic, Deineka, Addbot, Cowgod14, MrOllie, Yaframa, OLEnglish, یاری, Legobot, Luckas-bot, Yobot, Nashn, AnomieBOT, AmritasyaPutra, Royote, Citation bot, Ivan Kuckir, Coding.mike, GrouchoBot, ModiaShutosh, RibotBOT, Shadowjams, Pauldinhqd, FrescoBot, Mostafa.vafi, X7q, Jonasbn, Citation bot 1, Chenopodiaceous, Base698, GeypycGn, Miracle Pen, Pmdusso, Diana, Cutelyaware, WillNess, RjwilmsiBot, EmausBot, DanielWaterworth, Bleakgadfly, HolyCookie, Let4time, ClueBot NG, Jbragadeesh, Adityasinghhhhh, Athaphong, ۱۷, Helpful Pixie Bot, Sangdol, Sboosali, Dvanatta, Junkyardsparkle, Jochen Burghardt, Kirpo, Vsethuooo, RealFoxX, Arfaest Ealdwritere, AntonDevil, Painted Fox, Ramiyam, Bwgs14, Iokevins, Angelababy00 and Anonymous: 162

- **Radix tree** *Source:* <http://en.wikipedia.org/wiki/Radix%20tree?oldid=655924501> *Contributors:* Cwitty, Edward, CesarB, Dcoetze, AaronSw, Javidjamae, Gwalla, Bhyde, Andreas Kaufmann, Quetzuce, Brim, Guy Harris, Noosphere, Daira Hopwood, GregorB, Qwertyus, Yurik, Adoniscik, Me and, Piet Delport, Dogcow, Gulliveig, Modify, C.Fred, DBeyer, Optikos, Srvhrs, Malbrain, Frap, Cybercobra, MegaHasher, Khazar, Nausher, Makyen, Babbling.Brook, Dicklyon, DavidDecotigny, Ahly1, Cydebot, Morteheu, Coffee2theorems, Tedickey, Rocchini, Phishman3579, Jy00912345, SparsityProblem, Burkeaj, Cobi, VolkovBot, Jamelan, Abatishchev, Para15000, Sameemir, Arkanosis, Safek, XLinkBot, Hetori, Rgruijan, Dsimic, Addbot, Ollydbg, Lightbot, Npgall, Drachmae, Citation bot, Shy Cohen, Pauldinhqd, FrescoBot, Citation bot 1, SpmRmvBot, ICEAGE, MastiBot, Hesamwls, Puffin, TYelliot, Helpful Pixie Bot, ChrisGualtieri, Saffles, Awnedion, Crow, Simonfakir and Anonymous: 75
- **Suffix tree** *Source:* <http://en.wikipedia.org/wiki/Suffix%20tree?oldid=656788295> *Contributors:* AxelBoldt, Michael Hardy, Delirium, Alifio, Charles Matthews, Dcoetze, Jgoloran, Sho Uemura, Gifflite, P0nc, Sundar, Two Bananas, Andreas Kaufmann, Squash, Kbhb3rd, Bcat, Shoujun, Christian Kreibich, R. S. Shaw, Jemfinch, Blahma, Mechonbarsa, RJFJR, Wsloand, Oleg Alexandrov, Ruud Koot, Dionyziz, JMCOREY, Ffaarr, Bgwhite, Vecter, Ru.spider, TheMandarin, Nils Grimsmo, Lt-wiki-bot, TheTaxman, DmitriyV, Heavyrain2408, SmackBot, C.Fred, TripleF, Cybercobra, Ninjagecko, ThePianoGuy, Nux, Beetstra, MTSbot~enwiki, Requestion, MaxEnt, Cydebot, Jleunissen, Thiijs!bot, Jhclark, Headbomb, Leafman, MER-C, CobaltBlue, Johnbibby, A3nm, David Eppstein, Bbi5291, Andre.holzner, Doranchak, Dhruvbird, Jamelan, NVar, Xevior, ClueBot, Garyzx, Para15000, Safek, Xodarap00, Stephengmatthews, XLinkBot, Addbot, Deselaers, DOI bot, RomanPszonka, Chamal N, Nealjc, Yobot, Npgall, Kilom691, Senvey, Citation bot, Eumolpo, Xqbot, Gilo1969, Sky Attacker, X7q, Citation bot 1, SpmRmvBot, Skyerise, Illya.havsiyevych, RedBot, Xutaodeng, Luismsgomes, Mavam, RjwilmsiBot, 12hugo34, Grondilu, Ronni1987, EdoBot, ClueBot NG, Kasirbot, Andrew Helwer, Jochen Burghardt, Cos2, Anurag.x.singh and Anonymous: 77
- **Suffix array** *Source:* <http://en.wikipedia.org/wiki/Suffix%20array?oldid=656122883> *Contributors:* Edward, Mjordan, BenRG, Kiwibird, Tobias Bergemann, Gifflite, Mboverload, Vksit, Beland, Karol Langner, Andreas Kaufmann, MeltBanana, Malcolm rowe, Arnabdotorg, Nroets, RJFJR, Ruud Koot, Qwertyus, Gaius Cornelius, Nils Grimsmo, Bkil, SmackBot, TripleF, Malbrain, Chris83, Thiijs!bot, Headbomb, JoaquinFerrero, Wolfgang-gerlach~enwiki, David Eppstein, Cobi, Singleheart, Jwarhol, Garyzx, Alexbot, XLinkBot, Addbot, EchoBlaze94, Matěj Grabovský, Yobot, FrescoBot, Libor Vilímek, Chuancong, Gailcarmichael, Saketkc, ZéroBot, Dennis714, Andrew Helwer, ChrisGualtieri, JingguoYao, SteenthIWbot, StephanErb, Gauvain, Cos2, Hvaara, Anurag.x.singh and Anonymous: 50
- **Compressed suffix array** *Source:* <http://en.wikipedia.org/wiki/Compressed%20suffix%20array?oldid=622236160> *Contributors:* Andreas Kaufmann, RHaworth, Ruud Koot, Headbomb, Yobot, Stringologist, BG19bot, DoctorKubla, Evanrosen, G.ottaviano, Phantomcaterpillar and Anonymous: 3
- **FM-index** *Source:* <http://en.wikipedia.org/wiki/FM-index?oldid=654424665> *Contributors:* Andreas Kaufmann, Ruud Koot, SDC, Noodlez84, Magioladitis, Tnabtaf, Mild Bill Hiccup, Sun Creator, Yobot, Raven1977, Omnipaedita, Fingerz, Vyahhi, Orenburg1, Cventus, G.ottaviano, Phantomcaterpillar and Anonymous: 14
- **Generalized suffix tree** *Source:* <http://en.wikipedia.org/wiki/Generalized%20suffix%20tree?oldid=634260910> *Contributors:* Michael Hardy, Dcoetze, Bernhard Bauer, Gifflite, Andreas Kaufmann, Wsloand, Ruud Koot, Yurik, Nils Grimsmo, JanCeuleers, Joey-das-WBF, Ninjagecko, Cydebot, Bbi5291, Malik Shabazz, Aditya, Citation bot, Nodirt, Anurag.x.singh and Anonymous: 16
- **B-trie** *Source:* <http://en.wikipedia.org/wiki/B-trie?oldid=612576933> *Contributors:* Andreas Kaufmann, RJFJR, DebackerL, Merosonox, Fram, Cydebot, Headbomb, David Eppstein, R'n'B, Bob1960events, MindstormsKid, 1ForTheMoney, Hetori, Addbot, AvicBot, BattyBot and Anonymous: 1
- **Judy array** *Source:* <http://en.wikipedia.org/wiki/Judy%20array?oldid=645199206> *Contributors:* The Anome, Pnm, Ejrh, Dcoetze, Furykef, Jason Quinn, Andreas Kaufmann, Minghong, Gmaxwell, Nigosh, Freshneez, Alynnna Kasmira, Fintler, Bluebot, Malbrain, Cybercobra, EIFY, Doug Bell, JoeBot, Mellery, Cydebot, Alaibot, RainbowCrane, Uberdude85, Nave.notnilc, Djmckee1, XHire, Dlrohrer2003, Garyzx, Y Less, Rolingyid, C. A. Russell, Addbot, Cowgod14, Mortense, AnomieBOT, Cyphoidbomb, Kallikanzarid, Jesse V., Mattbierner, EmausBot, Voomoo, ClueBot NG, JudayArry, B.suhasini, Tony Tan, Justincheng12345-bot, ChrisGualtieri, Danmoberly, Iokevins and Anonymous: 25
- **Ctrie** *Source:* <http://en.wikipedia.org/wiki/Ctrie?oldid=639788716> *Contributors:* Ebruchez, Palmcluster, Qwertyus, Chickencha, A3nm, Sun Creator, Yobot, Midinasturasz, Axel22, AnomieBOT, Jesse V., BG19bot, ChrisGualtieri, Liz and Anonymous: 10
- **Directed acyclic word graph** *Source:* [http://en.wikipedia.org/wiki/Deterministic\\_acyclic\\_finite\\_state\\_automaton?oldid=651175972](http://en.wikipedia.org/wiki/Deterministic_acyclic_finite_state_automaton?oldid=651175972) *Contributors:* Damian Yerrick, RL, Phil Boswell, Blkonrad, Watcher, Andreas Kaufmann, Smyth, Bo Lindbergh, Marasmusine, Mandarax, Qwertyus, Rjwilmsi, Bgwhite, Gwernol, Thelb4, SmackBot, BiT, Sadads, JonHarder, Radagast83, Rofl, Balrog, Smhanov, Chimz, MC10, Headbomb, David Eppstein, Chkno, CommonsDelfinker, Archie172, Steel1943, ClueBot, Norman Ramsey, Nightmaare, NoblerSavager, Citation bot 1, Bokeh.Senssei, RjwilmsiBot, Ort43v, AvocatoBot, Hoang.tran, Deltahedron, Fizzlesbach, Jochen Burghardt, Monkbot, Angelababy00 and Anonymous: 22
- **Ternary search tree** *Source:* <http://en.wikipedia.org/wiki/Ternary%20search%20tree?oldid=657384171> *Contributors:* Booyabazooka, Selket, Jds, Shadypalm88, Antaeus Feldspar, Cwolfsheep, Qwertyus, Wavelength, Xuul, Richardj311, SmackBot, Bazonka, Cybercobra, Rschwieb, SkyWalker, Zureks, Cydebot, Alaibot, JustAGal, Raanoo, Magioladitis, Tedickey, Klapautius, Potatoswatter, Maghnus, Trent1799, Arkanosis, Pyroflame 91, Addbot, Twimoki, AnomieBOT, Knowtheory, FrescoBot, ZéroBot, ClueBot NG, Fxsjy, Rangilo Gujarati, BG19bot, Mayank.kulkarni, Aisteco, Hdanak, Awnedion, Patrick O'Jackson, Abdsahin, Monmohans and Anonymous: 71

- **And-or tree** *Source:* <http://en.wikipedia.org/wiki/And%2080%93or%20tree?oldid=650725849> *Contributors:* AxelBoldt, Hooperbloob, Intgr, Pete.Hurd, Jpbowen, Garion96, Rwww, Addshore, Allamrbo, Ryan Roos, Nfwu, CBM, Cydebot, Mblumber, Mnp~enwiki, Routolio, Justmeherenow, Fratrep, Classicalcon, Logperson, Addbot, Yobot, H taammoli, Jamjam337, Saumaun, ClueBot NG, Widr and Anonymous: 5
- **(a,b)-tree** *Source:* [http://en.wikipedia.org/wiki/\(a%2Cb\)-tree?oldid=659730649](http://en.wikipedia.org/wiki/(a%2Cb)-tree?oldid=659730649) *Contributors:* Skysmith, Centrx, Linas, Qwertyus, Lockley, Gardar Rurak, Bovineone, Nick, Onodevo, Bluebot, JonHarder, Cybercobra, Skapur, Cydebot, Alaibot, MER-C, David Eppstein, AlkyAlky, Dinybot, Flyer22, Cacadril, Addbot, Sapeur, Yxejamir, ElNuevoEinstein, Hardwigg, BG19bot, Odenkos, Caliburn, Al Safi Shahid Tuma and Anonymous: 5
- **Link/cut tree** *Source:* <http://en.wikipedia.org/wiki/Link/cut%20tree?oldid=648933634> *Contributors:* Giftlite, Rkleckner, Pgan002, GregorB, Bgwhite, Goffrie, Cybercobra, Martlau, Cydebot, Alaibot, David Eppstein, Yobot, Erel Segal, John of Reading, FactorialG, Frietjes, ChrisGualtieri, Drrilll and Anonymous: 15
- **SPQR tree** *Source:* <http://en.wikipedia.org/wiki/SPQR%20tree?oldid=616019689> *Contributors:* Andreas Kaufmann, Pol098, Rjwilmsi, SmackBot, Cybercobra, Riedel~enwiki, Harrigan, MaxEnt, Cydebot, Headbomb, A3nm, David Eppstein, Auntof6, Kintaro, Kakila, DOI bot, Twri, LilHelpa, EmausBot, WikitanvirBot, Eng 104\*, Khazar2 and Anonymous: 4
- **Spaghetti stack** *Source:* [http://en.wikipedia.org/wiki/Parent\\_pointer\\_tree?oldid=644134525](http://en.wikipedia.org/wiki/Parent_pointer_tree?oldid=644134525) *Contributors:* The Anome, Edward, Karada, Eric119, Alf, Timwi, Dcoetzee, Furrykef, Ruakh, Somercet, Tagishsimon, Antandrus, Tyc20, Congruence, Grenavitar, Woohookitty, Ruud Koot, Qwertyus, It's-is-not-a-genitive, Eyal0, Skaller~enwiki, SmackBot, TimBentley, Cybercobra, Dreftymac, Ripounet, Lindsey Kuper, Blaisorblade, Dougher, Gwern, Tom Gundtofte-Bruun, EmptyString, Jerryobject, Pichpitch, Addbot, Masharabinovich, BenzolBot, JnRouvinac, John of Reading, ZéroBot, Sean r lynch, Helpful Pixie Bot and Anonymous: 16
- **Disjoint-set data structure** *Source:* <http://en.wikipedia.org/wiki/Disjoint-set%20data%20structure?oldid=660818378> *Contributors:* The Anome, Michael Hardy, Dominus, LittleDan, Charles Matthews, Dcoetzee, Grendelkhan, Pakaran, Giftlite, Pgan002, Jonel, Deewiant, Andreas Kaufmann, Qutezuce, SamRushing, Nyenyc, Beige Tangerine, Msh210, Bigaln2, ReyBrujo, LOL, Bkkbrad, Ruud Koot, Qwertyus, Kasei-jin~enwiki, Rjwilmsi, Salix alba, Intgr, Freshenesz, Wavelength, Sceptre, NawlinWiki, Spike Wilbury, Kevtrice, Spirk0, Ripper234, Cedar101, Tevildo, SmackBot, Isaac Dupree, Oli Filth, Nikaustr, Lambiam, Archimerged, IanLiu, Dr Greg, Superjoe30, Edward Vielmetti, Gfonsecab, Headbomb, Kenahoo, Stellmach, David Eppstein, Chkno, Glrx, Rbrewer42, Kyle the bot, Jamelan, Oaf2, MasterAchilles, Boydski, Alksentr, Adrianwn, Vanisheduser12a67, DumZiBoT, Cldoyle, Dekart, Addbot, Shmilymann, Lightbot, Chipchap, Tonycao, Yobot, Erel Segal, Rubinbot, Sz-iwbot, Citation bot, Fantasticfears, Backpackadam, Williacb, HRoestBot, MathijsM, Akim Demaille, Rednas1234, EmausBot, Zhouji2010, ZéroBot, Wmayner, ChuispastonBot, Mankarse, Nullzero, Aleskotnik, Andreschulz, FutureTrillionaire, Qunwangcs157, Andyhowlett, Simonemainardi, William Di Luigi, Kimi91, Sharma.illusion, Kbh95, Kennysong, R.J.C.vanHaaften, Ahg simon and Anonymous: 70
- **Space partitioning** *Source:* <http://en.wikipedia.org/wiki/Space%20partitioning?oldid=606615427> *Contributors:* Michael Hardy, Phil Boswell, Bernhard Bauer, Altenmann, Giftlite, Oleg Alexandrov, Ruud Koot, Reedbeta, Nlu, Kjmathew, PaulJones, SmackBot, Timrb, Unyoyega, CBM, Headbomb, Glen, Adavidb, M-le-mot-dit, Flyinglemon, Yumf, Arminahmady, Addbot, ArthurBot, Erik9bot, Mod mmg, Frostyandy2k, Niky cz, ClueBot NG, Cwks and Anonymous: 10
- **Binary space partitioning** *Source:* <http://en.wikipedia.org/wiki/Binary%20space%20partitioning?oldid=639533255> *Contributors:* Bryan Derksen, The Anome, Tarquin, B4hand, Michael Hardy, Chan siuman, Angela, Charles Matthews, Dcoetzee, Dysprosia, Percivall~enwiki, Fredrik, Altenmann, Wonghang, WiseWoman, Frencheigh, Wmahan, Mdob, Cbraga, Abdull, Leithian, Zetawoof, CyberSkull, Chrisjohnson, Kdau, Stephan Leeds, Harej, Oleg Alexandrov, Ariadie, LOL, Tabletop, GregorB, Dionyziz, Yar Kramer, Reedbeta, FlaBot, QuasarTE, Kri, Chobot, YurikBot, Wavelength, Amanaplanacanalpanama, Wiki alf, Palmin, KnightRider~enwiki, Kelvie, Miquonranger03, Cybercobra, Jafet, Cydebot, Thij's!bot, Headbomb, Operator link, JAnDbot, Spodi, NuclearFriend, David Eppstein, M-le-mot-dit, Jamesontai, VolkovBot, Svick, JohnnyMrNinja, WikiLaurent, ClueBot, Mild Bill Hiccup, DragonBot, Prikipedia, DanielPharos, DumZiBoT, Addbot, RPHv, Amritchowdhury, Lightbot, Luckas-bot, Yobot, TaBOT-zerem, AnomieBOT, Noxin911, Citation bot, Twri, ArthurBot, Xqbot, Obiwhonn, Gbrui, Brutaldeluxe, Immibis, Frescobot, LogiNevermore, Brucenaylor, Gyunt, AquaGeneral, Bomazi, Cgbuff, Jkwchui, ChrisGualtieri, NtpNtp, TreeMan100, Immonster, Mogie Bear and Anonymous: 58
- **Segment tree** *Source:* <http://en.wikipedia.org/wiki/Segment%20tree?oldid=635996004> *Contributors:* Dean p foster, Csernica, Diomidis Spinellis, XLerate, RussBot, Gareth Jones, SmackBot, Cybercobra, Dicklyon, AdultSwim, Pqrstuv, Cydebot, Thij's!bot, Jj137, Tgeairn, Phishman3579, Mikhail Dvorkin, Pdcook, Tortoise 74, NoSyu, Dhruvbird, Optigan13, Alfredo J. Herrera Lago, Portalian, XLinkBot, Addbot, Lightbot, Yobot, AnomieBOT, Yintianjiao, X7q, Dstrash, Adlerbot, Jonesey95, MoreNet, Rahmtin rotabi, Aida147, MerllwBot, Theemathas, Michaelkmccandless and Anonymous: 26
- **Interval tree** *Source:* <http://en.wikipedia.org/wiki/Interval%20tree?oldid=653978331> *Contributors:* Dean p foster, Dcoetzee, Andreas Kaufmann, Raboof, RJFJR, Isomeme, Qwertyus, Smithph, Gam3, Bgwhite, Porao, Gareth Jones, Breuwi, SmackBot, Dicklyon, Cydebot, Sytelus, Magioladitis, Raduberinde, 28421u2232nfencenc, David Eppstein, Rodgling, CommonsDelinker, Phishman3579, Cobi, Zhybear, Tortoise 74, VolkovBot, Mkcmkc, Emschorsch, Jamelan, Mobiusinversion, Jojalozzo, Mloskot, Svick, CharlesGillingham, Justin545, SchreiberBike, Jwpat7, Saeed.gnu, Addbot, Favonian, Yobot, AnomieBOT, Digulla, LilHelpa, Xqbot, Kungi01, Brutaldeluxe, Frescobot, X7q, Spakin, Cosmoskramer, Gareth Griffith-Jones, Daiyuda, CostinulAT, Carlwitt, Iceboy8 and Anonymous: 44
- **Range tree** *Source:* <http://en.wikipedia.org/wiki/Range%20tree?oldid=651093085> *Contributors:* Dcoetzee, Smalljim, Caesura, RJFJR, Breuwi, Andreas Fabri, Cybercobra, Cydebot, Alaibot, Sanchom, David Eppstein, XLinkBot, Addbot, Yobot, Infvwl, Rfmge and Anonymous: 17
- **Bin (computational geometry)** *Source:* [http://en.wikipedia.org/wiki/Bin%20\(computational%20geometry\)?oldid=610733976](http://en.wikipedia.org/wiki/Bin%20(computational%20geometry)?oldid=610733976) *Contributors:* Michael Hardy, Andycjp, Lockley, Vegaswikian, Ospalh, Arthur Rubin, SmackBot, Lifebaka, Chris the speller, AndrewHowse, David Eppstein, Yumf, Agent007ravi and Anonymous: 3
- **K-d tree** *Source:* <http://en.wikipedia.org/wiki/K-d%20tree?oldid=661163867> *Contributors:* Bryan Derksen, Paul A, CesarB, Dcoetzee, Grendelkhan, Wwheeler, Bearcat, Altenmann, Connelly, Giftlite, Achurch, BenFrantzDale, Neilc, Andreas Kaufmann, Madduck, Rich Farmbrough, Skinkie~enwiki, Huan086, TheParanoidOne, Don Reba, SteinbDJ, Wsloand, Woohookitty, Ruud Koot, Waldir, Btyner, BD2412, Qwertyus, Rjwilmsi, Revpj, Vegaswikian, Reedbeta, Kri, YurikBot, Wavelength, Angus Lepper, Biscwit, Gareth Jones, Zwobot, Tetracube, Cedar101, Amit man, SmackBot, Arkitus, Jfmiller28, MclD, Nintendude, Ilyathemuromets, Meepster, Cybercobra, Jeff Wheeler, Mwtoews, NikolasCo, Dicklyon, Mulder416sBot, Braddodson, Ceran, Karloman2, Jokes Free4Me, Szabolcs Nagy, Borax~enwiki, Equentil, Cydebot, KiwiSunset, Casbah~enwiki, Lfstevens, Pixor, Dosttiey, BrotherE, David Eppstein, User A1, Gwern, Leyo, SJP, Lamp90, Chaos5023, TXiKiBoT, MusicScience, EQ5afN2M, Formerly the IP-Address 24.22.227.53, Svick, Bajsejohannes,

Genieser, Justin W Smith, The Thing That Should Not Be, Uncle Milty, Yumf, Algomaster, Uranor, Karlhendrikse, PixelBot, Colingodsey, Stepheng3, XLinkBot, C. lorenz, Addbot, Ts3r0, Mohamedadaly, Favonian, Sfandino, اناری, Luckas-bot, Yobot, Ptbotgourou, AnomicBOT, MYguel, Citation bot, TheAMmollusc, Kirigiri, Miym, Brandonwood, FrescoBot, D'ohBot, Citation bot 1, Bunyk, Ritoken, MladenWiki, Ficuep, Chad.burrs, EmausBot, Iamchenzetian, Bosik GN, RenamedUser01302013, ZéroBot, Sgoder, Cymru.lass, ClueBot NG, Kkddkkddd, Wcherowi, Hofmic, Ronen.katz, Meniv, Trololololololo, Jtweng45, Rummelsworth, Matt think so, AwamerT, BattyBot, Timothy Gu, Monkbot, Le Creusot and Anonymous: 148

- **Implicit k-d tree** *Source:* <http://en.wikipedia.org/wiki/Implicit%20k-d%20tree?oldid=497261567> *Contributors:* Paul A, Andreas Kaufmann, BD2412, Rjwilmsi, Kri, Meawoppl, Cybercobra, Tim Q. Wells, Dicklyon, Iridescent, Cydebot, Nick Number, R'n'B, KoenDelaere, Svick, Tesi1700, ImageRemovalBot, Genieser, Rockfang, Karlhendrikse, Yobot, Bduvenhage and Anonymous: 2
- **Min/max kd-tree** *Source:* <http://en.wikipedia.org/wiki/Min/max%20kd-tree?oldid=602703067> *Contributors:* Andreas Kaufmann, JokesFree4Me, Cydebot, Alaibot, Nick Number, Signalhead, KoenDelaere, Genieser, Rockfang, Yobot, Daniel Strobusch, FrescoBot, Bduvenhage and Anonymous: 3
- **Adaptive k-d tree** *Source:* <http://en.wikipedia.org/wiki/Adaptive%20k-d%20tree?oldid=577755646> *Contributors:* Qutezuce, Wsloand, Onodevo, Bluebot, Cydebot, David Eppstein, Favonian, RjwilmsiBot and Anonymous: 1
- **Quadtree** *Source:* <http://en.wikipedia.org/wiki/Quadtree?oldid=660458485> *Contributors:* Bryan Derksen, Danny, Tomo, Ronz, Julesd, Dcoetze, Dysprosia, Fredrik, Bkell, Enochlau, Giftlite, Lancept, Johnflux, Spottedowl, 4pq1injbok, Happyvalley, Jason Davies, Interiot, Arthena, Cdc, RJFJR, Wsloand, Oleg Alexandrov, AllanBz, Maglev2, Piet Delport, Gaius Cornelius, Yrithinnd, Aburad, ALoopingIcon, Scope creep, Peterjanbroomans, SmackBot, Jprg1966, Aeturnus, Frap, GeorgeMoney, Cybercobra, Nickblack, Cydebot, Gimmetrow, Headbomb, Lfsteven, Hermann.tropf, David Eppstein, DerHexer, Tony1212, Miko3k, RHaden, XLinkBot, Addbot, DOI bot, Cuaxdon, Wojciech mula, Timeroot, DutchCanadian, Citation bot, Backpackadam, TechnologyTrial, Shencoop, FrescoBot, RedBot, MorganRod-erick, BZRatfink, Dgreenheck, Kkddkkddd, Hyperplane, Ben morphett, MilerWhite, IluvatarBot, DrWolfen, Padenton, Khazar2, Saffles, Kdkumd, Rebcabin42, DissidentAggressor and Anonymous: 67
- **Octree** *Source:* <http://en.wikipedia.org/wiki/Octree?oldid=654245561> *Contributors:* Bryan Derksen, SimonP, CesarB, Dysprosia, Furykef, Fredrik, Giftlite, Wolfkeeper, Jacob grace, DragonflySixtyseven, Andreas Kaufmann, WhiteTimberwolf, Kgaughan, Viriditas, Kierano, Nomis80, Mdd, Arthena, Scott5114, Melaen, Wsloand, MIT Trekkie, Bkkbrad, Ruud Koot, Sadangel, AySz88, Ffaarr, Kri, Exe, YurikBot, Sscomp2004, Nothings, SmackBot, Alksub, Scott Paeth, Eike Welk, Bluebot, Alanf777, Cybercobra, Olsonse, GeordieMcBain, Cydebot, Alaibot, June8th, Lfsteven, MagiMaster, JaGa, Rei-bot, Noerrorsfound, KoenDelaere, ImageRemovalBot, Alexbot, Rahul220, Moberg, XLinkBot, MystBot, Addbot, Balabot, Luckas-bot, Yobot, Wonderfl, Litherum, Alienskull, 23u982j32r29j92, AnomicBOT, ArthurBot, Xqbot, TechnologyTrial, Skyerise, EmausBot, Claystu, JosephCatrambone, Gromobir, ClueBot NG, Defiantredpill, Indiana State, Luc Jaulin, Toki782, Monkbot, Sdsar and Anonymous: 63
- **Linear octrees** *Source:* [http://en.wikipedia.org/wiki/Linear\\_octree?oldid=471784463](http://en.wikipedia.org/wiki/Linear_octree?oldid=471784463) *Contributors:* Pgan002, Cydebot, Eekster, Jncraton and Anonymous: 1
- **Z-order curve** *Source:* <http://en.wikipedia.org/wiki/Z-order%20curve?oldid=656846685> *Contributors:* Michael Hardy, Pnm, Kku, CesarB, Sverdrup, Giftlite, BenFrantzDale, Beland, Andreas Kaufmann, Cariaso, Bluap, Hesperian, Sligocki, Ynhockey, Einstein9073, Vi-vaEmilyDavies, Joriki, Rjwilmsi, Lendorien, Kri, Zotel, Wavelength, RussBot, Black Falcon, Robertd, SmackBot, Ephraim33, Lambiam, Paul Foxworthy, CmdrObot, CBM, Fisherjs, Lfsteven, Hermann.tropf, Magioladitis, David Eppstein, Edratzer, Tonyskjellum, Wpegden, DnetSvg, SoxBot III, Addbot, Yobot, Citation bot, ArthurBot, Shadowjams, Kkddkkddd, Daniel Minor, Jimw338, Patrick87 and Anonymous: 17
- **UB-tree** *Source:* <http://en.wikipedia.org/wiki/UB-tree?oldid=627523459> *Contributors:* Andreas Kaufmann, Rich Farmbrough, Qwertyus, Welsh, Cybercobra, CmdrObot, Honnza, Cydebot, Travelbird, Lfsteven, Hermann.tropf, Svick, Addbot, Jason.surratt and Anonymous: 5
- **R-tree** *Source:* <http://en.wikipedia.org/wiki/R-tree?oldid=630752803> *Contributors:* The Anome, Bernhard Bauer, Altenmann, Hadal, Nick Pisarro, Jr., Curps, CALR, Qutezuce, Oik, Skinkie-enwiki, Foobaz, Minghong, Happyvalley, Mdd, MIT Trekkie, Stolee, MacTed, G.hartig, AllanBz, Alejos, Margosbot-enwiki, Sperxios, Chobot, YurikBot, RussBot, Jengelh, Alynnna Kasmira, Peterjanbroomans, SmackBot, Hydrogen Iodide, Elwood j blues, Bluebot, EncMstr, Милан Јелисавчић, Cybercobra, Mwtoews, Soumyasch, Pqrstuv, FatalError, CmdrObot, Jodawi, Lfsteven, Radim Baca, BrotherE, David Eppstein, Gwern, AAA!, Tony1212, Funandtrvl, Huisho, Freekh, Aednichols, Svick, Okoky, ImageRemovalBot, Drsadeq, Addbot, Mortense, Jason.surratt, Download, Lightbot, Krano, Matěj Grabovský, Jarble, Yobot, AnomicBOT, Ziyuang, Twri, ArthurBot, Obersachsebot, NoldorinElf, Citation bot 2, TobeBot, Trappist the monk, Cutelyaware, Dzhioev, Imbcmth, Chire, ClueBot NG, Drobilla, TRKeen, Chmarkine, Digital Organism, Barakafrit, Tastyjew123, Thomaswikiusername and Anonymous: 85
- **R+ tree** *Source:* <http://en.wikipedia.org/wiki/R%2B%20tree?oldid=565296367> *Contributors:* Tim Starling, Fuzzie, CesarB, Creidieki, Brian0918, Bluap, Minghong, Pearl, Happyvalley, RJFJR, Turnstep, AllanBz, Kerowyn, Chobot, SmackBot, Nintendude, NikolasCo, Jodawi, Alaibot, Headbomb, Lfsteven, Magioladitis, Funandtrvl, Aednichols, OsamaBinLogin, Svick, Alksentrs, WikHead, Addbot, Yobot, Westquote, J.pellicoli, Pmdusso and Anonymous: 10
- **R\* tree** *Source:* <http://en.wikipedia.org/wiki/R%20tree?oldid=628442133> *Contributors:* The Anome, Neilc, Minghong, Happyvalley, Mdd, MZMcBride, Frouaix, Chobot, SmackBot, Jodawi, Lfsteven, Magioladitis, Reedy Bot, Funandtrvl, Aednichols, Svick, Virtuald, Addbot, Lightbot, Yobot, AnomicBOT, Chire, Kounoupis, Stefan.krausekjaer, Dzonon 111, Ilfelle and Anonymous: 27
- **Hilbert R-tree** *Source:* <http://en.wikipedia.org/wiki/Hilbert%20R-tree?oldid=607162408> *Contributors:* Michael Hardy, Andreas Kaufmann, D6, Mdd, Hammertime, BD2412, Wavelength, SmackBot, Mwtoews, Niczar, Belizefan, SimonD, Jodawi, Solidpoint, Johnfn, Erechtheus, JCarlos, Funandtrvl, Niteris-enwiki, Svick, Okoky, Robin K. Albrecht, Damsgård, Yobot, Twri, Chire, FelixR23 and Anonymous: 11
- **X-tree** *Source:* <http://en.wikipedia.org/wiki/X-tree?oldid=544075925> *Contributors:* Danceswithzerglings, Hyperfusion, Eubot, Drumguy8800, SmackBot, Bluebot, Cybercobra, Dicklyon, Alaibot, Lfsteven, JL-Bot, Mild Bill Hiccup, Addbot, RjwilmsiBot, Chire and Anonymous: 5
- **Metric tree** *Source:* <http://en.wikipedia.org/wiki/Metric%20tree?oldid=660173445> *Contributors:* Bryan Derksen, Edward, Rjwilmsi, Brighterorange, Staffelde, SmackBot, Srchvrs, Cybercobra, Cydebot, Alaibot, Monkeyget, David Eppstein, Nsk92, Yobot, AnomicBOT, FrescoBot, Chire, Krajevski, Mrzuniga333 and Anonymous: 6
- **Vp-tree** *Source:* [http://en.wikipedia.org/wiki/Vantage-point\\_tree?oldid=643855307](http://en.wikipedia.org/wiki/Vantage-point_tree?oldid=643855307) *Contributors:* Bryan Derksen, Hadal, Rjwilmsi, Pburka, Piet Delport, Mcld, Bluebot, Srchvrs, Cybercobra, Dicklyon, Cydebot, Alaibot, Magioladitis, User A1, Sbjesse, Yobot, Aborisov, Brutaldeluxe, FrescoBot, RjwilmsiBot, Wpiechowski, Arthoviedo, Chsamt27, Monkbot and Anonymous: 10

- **BK-tree** *Source:* <http://en.wikipedia.org/wiki/BK-tree?oldid=627605824> *Contributors:* Bryan Derksen, Ehamberg, Quuxplusone, Piet Delport, Staffelde, DoriSmith, SmackBot, Fikus, Srvhrs, Cybercobra, Wossi, Cydebot, Endpoint, Alaibot, A3nm, Mkarlesky, Philip Trueman, Volkan YAZICI, ClueBot, Pauladin, Jesse V., Jfmantis, Templatetypedef, Ryanne jwood13, Patapizza, Loveoflearning1 and Anonymous: 9
- **Hash table** *Source:* <http://en.wikipedia.org/wiki/Hash%20table?oldid=660061425> *Contributors:* Damian Yerrick, AxelBoldt, Zundark, The Anome, BlckKnght, Sandos, Rgamble, LapoLuchini, AdamRetchless, Imran, Mrwojo, Frecklefoot, Michael Hardy, Nixdorf, Pnm, Axlrosen, TakuyaMurata, Ahoerstemeier, Nanshu, Dcoetze, Dysprosia, Furrykef, Omegatron, Wernher, Bevo, Tjdw, Pakaran, Secret-london, Robbot, Fredrik, Tomchiuk, R3m0t, Altenmann, Ashwin, UtherSRG, Miles, Giftlite, DavidCary, Wolfkeeper, BenFrantzDale, Everyking, Waltpohl, Jorge Stolfi, Wmahan, Neilc, Pgdn002, CryptoDerk, Knutux, Bug-enwiki, Sonjaaa, Teacup, Beland, Watcher, DNewhall, ReiniUrban, Sam Hocevar, Derek Parnell, Askewchan, Kogorman, Andreas Kaufmann, Kautuv, Shuchung-enwiki, T Long, Hydrox, Cfайлde, Luqui, Wrp103, Antaeus Feldspar, Khalid, Raph Levien, JustinWick, CanisRufus, Shanes, Iron Wallaby, Krakhan, Bobo192, Davidgothberg, Larry V, Sleske, Helix84, Mdd, Varuna, Baka toroi, Anthony Appleyard, Sligocki, Drbreznjev, DSatz, Akuchling, TShilo12, Nuno Tavares, Woohoo kitty, LOL, Linguica, Paul Mackay-enwiki, Davidfstr, GregorB, Meneth, Graham87, Kbdank71, Tostie14, Rjwilmsi, Scandum, Koavf, Kinu, Filu-enwiki, Nneonneo, FlaBot, Ecb29, Fraggle, Ingr, Fresheneesz, Antaeus Feldspar, YurikBot, Wavelength, RobotE, Mongol, RussBot, Me and, CesarB's unprivileged account, Lavenderbunny, Gustavb, Mipadi, Cryptoid, Mike.aizatsky, Gareth Jones, Piccolomomo-enwiki, CecilWard, Nethgirb, Gadget850, Bota47, Sebleblanc, Deeday-UK, Sycomonkey, Ninly, Gulliveig, Th1rt3en, CWenger, JLaTondre, ASchmoo, Kungfuadam, SmackBot, Apanag, Obakeneko, PizzaMargherita, Alksub, Eskimbot, RobotJcb, C4chandu, Arpitm, Neurodivergent, EncMstr, Crie, Deshraj, Tackline, Frap, Mayrel, Radagast83, Cybercobra, Decltype, HFuruse, Rich.lewis, Esb, Acdx, MegaHasher, Doug Bell, Derek farn, IronGargoyle, Josephsieh, Peter Horn, Pagh, Saxon, Tawkerbot2, Ouishoebean, CRGreathouse, Aly1, MaxEnt, Seizethedave, Cgma, Not-just-yeti, Thermon, OtterSmith, Ajo Mama, Stannered, AntiVandalBot, Hosamaly, Thailyn, Pixor, JAnDbot, MER-C, Epeefleche, Dmbstudio, SiobhanHansa, Wikilolo, Bongwarrior, QrczakMK, Josephskeller, Tedickey, Schwarzbichler, Cic, Allstarecho, David Eppstein, Oravec, Gwern, Magnus Bakken, Glrx, Narendrak, Tikiwont, Mike.lifeguard, Luxem, NewEnglandYankee, Cobi, Cometsstyle, Winecellar, VolkovBot, Simulationnelson, Floodyberry, Anurmi-enwiki, BotKung, Collin Stocks, JimJewett, Nighthkaos, Spinningspark, Abatishchev, Helios2k6, Kehrbykid, Kbrose, PeterCanthropus, Gerakibot, Trivbe, Digwuren, Svick, JL-Bot, ObfuscatePenguin, ClueBot, Justin W Smith, ImperfectlyInformed, Adrianwn, Mild Bill Hiccup, Niceguyedc, JJuran, Groxx, Berean Hunter, Eddof13, Johnuniq, Arlolra, XLinkBot, Hetori, Pichpich, Paulsheer, TheTraveler3, MystBot, Karuthedam, Wolkyim, Addbot, Gremel123, Scientus, CanadianLinuxUser, MrOllie, Numbo3-bot, Om Sao, Zorrobot, Jarble, Frehley, Legobot, Luckas-bot, Yobot, Denispir, KamikazeBot, Dmcomer, AnomieBOT, Erel Segal, Jim1138, Sz-iwbot, Citation bot, ArthurBot, Baliaime, Drilnoth, Arbalest Mike, Ched, Shadowjams, Kracekumar, FrescoBot, W Nowicki, X7q, Sae1962, Citation bot 1, Velociostrich, Simonsarris, Iekpo, Trappist the monk, SchreyP, Grapesoda22, Patmorin, Cutelyaware, JeepdaySock, Shafigoldwasser, Kastchei, DuineSidhe, EmausBot, Super48paul, Ibbn, DanielWaterworth, Mousehousemd, ZéroBot, Purple, Ticklemepink42, Paul Kube, Demonkoryu, Donner60, Carmichael, Phearthecal, Aberdein01, Teapeat, Rememberway, ClueBot NG, AznBurger, Incompetence, Rawafmail, Cntras, Rezabot, Jk2q3jrkls, Helpful Pixie Bot, Jan Spousta, MusikAnimal, SanAnMan, Pbrneau, AdventurousSquirrel, Triston J. Taylor, CitationCleanerBot, Happyuk, FeralOink, Spacemanaki, Aloksukhwani, Eminull, Deveedutta, Shmageggy, IgushevEdward, AlecTaylor, Mcom320, Thomas J. S. Greenfield, Razibot, Djszapi, QuantifiedElf, Chip Wildon Forster, Tmferrara, Tuketu7, Monkbot, Iokevins, Oleaster, Micahsaint, Mtnorthpoplar and Anonymous: 429
- **Hash function** *Source:* <http://en.wikipedia.org/wiki/Hash%20function?oldid=659455088> *Contributors:* Damian Yerrick, Derek Ross, Taw, BlckKnght, PierreAbbat, Miguel-enwiki, Imran, David spector, Dwheeler, Hfastedge, Michael Hardy, EddEdmondson, Ixfd64, Mdebets, Nanshu, J-Wiki, Jc-enwiki, Vanis-enwiki, Dcoetze, Ww, The Anomebot, Doradus, Robbot, Noldoaran, Altenmann, Mike-pelley, Connelly, Giftlite, Paul Richter, KelvSYC, Wolfkeeper, Obli, Everyking, TomViza, Brona, Malycetenar, Jorge Stolfi, Matt Crypto, Utcursch, Knutux, OverlordQ, Kusunose, Watcher, Karl-Henner, Talrias, Peter bertok, Quota, Eisnel, Jonmcauliffe, Rich Farmbrough, Antaeus Feldspar, Bender235, Chalst, Evand, PhilHibbs, Haxwell, Bobo192, Sklender, Davidgothberg, Boredzo, Helix84, CyberSkull, Atlant, Jeltz, Mmmready, Apoc2400, InShanee, Vellela, Jopxtion, ShawnVW, Kurivaim, MIT Trekkie, Redvers, Blaxthos, Kazvorpal, Brookie, Linas, Mindmatrix, GVOLTT, LOL, TheNightFly, Drostie, Pfunk42, Graham87, Qwertys, Toolan, Rjwilmsi, Seraphimblade, Pabix, LjL, Utuado, Nguyen Thanh Quang, FlaBot, Harmil, Gurch, Thenowhereman, Mathrick, Ingr, M7bot, Chobot, Roboto de Ajvol, YurikBot, Wavelength, RattusMaximus, RobotE, CesarB's unprivileged account, Stephenb, Pseudomonas, Andipi, Zeno of Elea, EngineerScotty, Mikeblas, Fender123, Bota47, Tachyon01, Ms2ger, Eurosong, Dinkel, Lt-wiki-bot, Ninly, Gulliveig, StealthFox, Claygate, Snoops-enwiki, QmunkE, Emc2, Appleseed, Tobi Kellner, That Guy, From That Show!, Jbalint, SmackBot, InverseHypercube, Bomac, KocjoBot-enwiki, BiT, Gilliam, Raghaw, Schmitemye, Mnb9rca, JesseStone, Oli Filth, EncMstr, Octahedron80, Nbarth, Kmag-enwiki, Malbrain, Chlewbot, Shingra, Midnightcomm, Lansey, Andrei Stroe, MegaHasher, Lambiam, Kuru, Alexcollins, Paulschou, RomanSpa, Chuck Simmons, KHAFFFFFFAAN, Erwin, Peyre, Vstarre, Pagh, MathStuf, ShakingSpirit, Agent X2, BrianRice, Courcelles, Juhachi, Neelix, Mblumber, SavantEdge, Adolphus79, Sytelus, Epbr123, Ultimus, Leedeth, Stualden, Folic Acid, AntiVandalBot, Xenophon (bot), JakeD409, Davorian, Powerdesi, JAnDbot, Epeefleche, Hamsterlopipthecus, Kirrages, Stangaa, Wikilolo, Coffee2theorems, Magioladitis, Pndfam05, Patelm, Nyttend, Kgfeischmann, Dappawit, Applrpn, STBot, R'n'B, Jfroelich, Francis Tyers, Demosta, Tgearin, J.delanoy, Maurice Carbaron, Svnsn, Wjaguar, L337 kyblmdstr, Globbet, Ontarioboy, Doug4, Meiskam, Jrmcdaniel, VolkovBot, Sjones23, Boute, TXiKiBoT, Christofpaar, GroveGuy, A4bot, Nxavar, Noformation, Cuddlyable3, Crashthatch, Wikiisawesome, Jediknil, Tastyllama, Skarz, LittleBenW, SieBot, WereSpielChequers, KrizbyZ, Xelgen, Flyer22, Iamhigh, Dhb101, IsaacAA, OKBot, Svick, FusionNow, BitCrazed, ClueBot, Cab.jones, Ggia, Unbuttered Parsnip, Garyzx, Mild Bill Hiccup, شری, Dkf11, SamHartman, Alexbot, Erebus Morgaine, Diaa abdelmoneim, Wordsputtogether, Tonysan, Rishi.bedi, XLinkBot, Kotha arun2005, Dthomsen8, MystBot, Karuthedam, SteveJothen, Addbot, Butterwell, TutterMouse, Dranorter, MrOllie, CarsracBot, AndersBot, Jeaise, Lightbot, Luckas-bot, Fraggie81, AnomieBOT, Erel Segal, Materialscientist, Citation bot, Twri, ArthurBot, Xqbot, Capricorn42, Matttoothman, M2millenium, Theclapp, RibotBOT, Alvin Seville, MerlLinkBot, FrescoBot, Nageh, MichealH, TruthIIPower, Haenous, Geoffreybernardo, Pinethicket, 10metreh, Mhgtg, Dinamik-bot, Vrenator, Keith Cascio, Phil Spectre, Jeffrrd10, Updatehelper, Kastchei, EmausBot, Timtempleton, Gfoley4, Mayazcherquo, MarkWegman, Dewritech, Jachto, John Cline, White Trillium, Fæ, Akerans, Paul Kube, Music Sorter, Donner60, Senator2029, Teapeat, Sven Mangaud, Shi Hou, Rememberway, ClueBot NG, Incompetence, Neuroneutron, Monchoman45, Cntras, Widr, Mtking, Bluechimera0, HMSSolent, Wikisian, JamesNZ, GarbledLecture933, Harpreet Osahan, Glacialfox, Winston Chuen-Shih Yang, ChrisGualtieri, Tech77, Jeff Erickson, Jonahugh, Lindsaywinkler, Tmferrara, Cattycat95, Tolcs, Frogger48, Eddiearin123, Philnap, Kanterme, Laberinto15, MatthewBuchwalder, Computilizer, Mark22207, GlennLawyer, Gcarvelli, BlueFenixReborn, Some Gadget Geek, Siddharthgondhi and Anonymous: 456
- **Open addressing** *Source:* <http://en.wikipedia.org/wiki/Open%20addressing?oldid=651972276> *Contributors:* LOL, Fresheneesz, O keyes, Alaibot, Headbomb, Gildos, Cobi, Skwa, Addbot, Crazy2be, Download, اماني, Citation bot, DataWraith, Jfmantis, Guzzrocha, Willfindyou, Nicmd and Anonymous: 10

- **Lazy deletion** *Source:* <http://en.wikipedia.org/wiki/Lazy%20deletion?oldid=641109258> *Contributors:* Bearcat, Xezbeth, Fresheneesz, Alksub, CmdrObot, David Eppstein, The Doink, Muro Bot, A Generic User, Addbot, AManWithNoPlan, Geronimo 091 and Anonymous: 11
- **Linear probing** *Source:* <http://en.wikipedia.org/wiki/Linear%20probing?oldid=636374192> *Contributors:* Ubiquity, Bearcat, Enochlau, Andreas Kaufmann, Gazpacho, Discospinster, RJFJR, Linas, Tas50, CesarB's unprivileged account, SpuriousQ, Chris the speller, JonHarder, MichaelBillington, Sbluen, Jeberle, Negruil, Jngnyc, Alaibot, Thijs!bot, A3nm, David Eppstein, STBot, Themania, OliviaGuest, C. A. Russell, Addbot, Tedzdog, Patmorin, Infinityive, Dixtosa, Danmoberly and Anonymous: 15
- **Quadratic probing** *Source:* <http://en.wikipedia.org/wiki/Quadratic%20probing?oldid=654430798> *Contributors:* Aragorn2, Dcoetzee, Enochlau, Andreas Kaufmann, Rich Farmbrough, ZeroOne, Oleg Alexandrov, Ryk, Eobot, CesarB's unprivileged account, Robertvan1, Mikeblas, SmackBot, InverseHypercube, Cybercobra, Wizardman, Jdan, Simeon, Magioladitis, David Eppstein, R'n'B, Philip Trueman, Hatmatbat10, C. A. Russell, Addbot, Yobot, Bavla, Kmgpratyush, Donner60, ClueBot NG, Helpful Pixie Bot, Yashykt, Vaibhav1992 and Anonymous: 34
- **Double hashing** *Source:* <http://en.wikipedia.org/wiki/Double%20hashing?oldid=658911643> *Contributors:* AxelBoldt, CesarB, Angela, Dcoetzee, Usrnme h8er, RJFJR, Zawersh, Pfunk42, Gurch, CesarB's unprivileged account, Momeara, DasBrose~enwiki, Cobblet, SmackBot, Bluebot, Hashbrowncipher, JForget, Only2sea, Alaibot, Thijs!bot, WonderPhil, Philip Trueman, Oxfordwang, Extensive~enwiki, Mild Bill Hiccup, Addbot, Tcl16, Smallman12q, Amiceli, Imposing, Jesse V., ClueBot NG, Exercisephys, Bdawson1982, Kevin12xd and Anonymous: 32
- **Cuckoo hashing** *Source:* <http://en.wikipedia.org/wiki/Cuckoo%20hashing?oldid=653415239> *Contributors:* Arvindn, Dcoetzee, Phil Boswell, Nyh, Pps, DavidCary, Neilc, Cwolfsheep, Zawersh, Ej, Bgwhite, CesarB's unprivileged account, Zr2d2, Zerodamage, SmackBot, Mandyhan, Thumperward, Cybercobra, Pagh, Jafet, CRGreathouse, Alaibot, Headbomb, Hermel, David Eppstein, S3000, Themania, Wjaguar, Mark cummins, LiranKatzir, Svick, Justin W Smith, Hetori, Addbot, Alquantor, Lmonson26, Luckas-bot, Yobot, Valentatas, Kurauskas, Thore Husfeldt, W Nowicki, Citation bot 1, Userask, EmausBot, BuZZdEE.Buzz, Rcsprinter123, Bomazi, Yoavt, BattyBot, Usernameasdf, Monkbot and Anonymous: 42
- **Coalesced hashing** *Source:* <http://en.wikipedia.org/wiki/Coalesced%20hashing?oldid=604942554> *Contributors:* Jll, Dcoetzee, Andreas Kaufmann, Zawersh, Oleg Alexandrov, Klortho, Ian Pitchford, Fresheneesz, CesarB's unprivileged account, Pmdboi, Confuzzled, SmackBot, Jafet, Basawala, Cic, StewieK, Tassedethe, Algotime and Anonymous: 7
- **Perfect hash function** *Source:* <http://en.wikipedia.org/wiki/Perfect%20hash%20function?oldid=632785927> *Contributors:* Edward, Cimon Avaro, Dcoetzee, Fredrik, Giftlite, Neilc, E David Moyer, Burschik, LOL, Ruud Koot, JMCorey, ScottJ, Mathbot, Spl, CesarB's unprivileged account, Dtреббин, Dlugosz, Gareth Jones, Salrizvy, Johndburger, SmackBot, Srvhtrs, Otus, 4hodmt, MegaHasher, Pagh, Mudd1, Headbomb, Wikilolo, David Eppstein, Glrx, Cobi, Drkarger, Gajeam, PixelBot, Addbot, G121, Bbb23, AnomieBOT, FrescoBot, Daoudamjad, John of Reading, Prvak, Maysak, Voomoo, Arka sett, BG19bot, SteveT84, Mcichelli, Latin.ufmg and Anonymous: 32
- **Universal hashing** *Source:* <http://en.wikipedia.org/wiki/Universal%20hashing?oldid=655956996> *Contributors:* Mattflaschen, DavidCary, Neilc, ArnoldReinhold, EmilJ, Rjwilmsi, Sdornan, SeanMack, Chobot, Dmharvey, Gareth Jones, Guruparan18, Johndburger, Twintop, CharlesHBennett, SmackBot, Cybercobra, Copysan, DaniellLemire, Pagh, Dwmalone, Winxa, Jafet, Arnstein87, Marc W. Abel, Sytelus, Francois.boutines, Headbomb, Golgofrinchian, David Eppstein, Copland Stalker, Danadocus, Cyberjoac, Ulamgamer, Bender2k14, Rswarbrick, Addbot, RPHy, Yobot, Mpatrascu, Citation bot, Citation bot 1, TPReal, Patmorin, RjwilmsiBot, EmausBot, Dewritech, ClueBot NG, Helpful Pixie Bot, Cleo, BG19bot, Walrus068, BattyBot, ChrisGualtieri, Zolgharnein, Zenuine and Anonymous: 41
- **Linear hashing** *Source:* <http://en.wikipedia.org/wiki/Linear%20hashing?oldid=614685210> *Contributors:* Dcoetzee, BrokenSegue, ENGIMA, Zawersh, Rjwilmsi, Personman, CesarB's unprivileged account, Olleicua, Bluebot, TenPoundHammer, MegaHasher, JultheP, Headbomb, Res2216firestar, David Eppstein, Zhuman, Addbot, CanadianLinuxUser, Gail, AnomieBOT, RjwilmsiBot, Spanduck, MrTux, Az186a, Piyush4748 and Anonymous: 24
- **Extendible hashing** *Source:* <http://en.wikipedia.org/wiki/Extendible%20hashing?oldid=589287141> *Contributors:* Kku, WorldsApart, JustinWick, Zawersh, Firsfron, Alex Kapranoff, MegaHasher, Alaibot, Headbomb, JohnBlackburne, Gloomy Coder, Svick, Treekids, Boing! said Zebedee, XLinkBot, Wolfeye90, Twimoki, Citation bot, Amiceli, RjwilmsiBot, John of Reading, Spanduck, Tommy2010, Planetary Chaos Redux, BattyBot, Lugia2453, Yashdv and Anonymous: 26
- **2-choice hashing** *Source:* <http://en.wikipedia.org/wiki/2-choice%20hashing?oldid=646282258> *Contributors:* Asparagus, Zawersh, Onodevo, Bluebot, JaGa, Glrx, Thomasda, Abatishchev, Pichpich, Yobot, Shashwat986, BG19bot, PaperKooper, Tmferrara and Anonymous: 1
- **Pearson hashing** *Source:* <http://en.wikipedia.org/wiki/Pearson%20hashing?oldid=644049668> *Contributors:* The Anome, Ed Poor, David spector, LittleDan, Charles Matthews, Phil Boswell, Fredrik, Bkonrad, Pbannister, PeterPearson, Avocado, Quuxplusone, Intgr, Epolk, Dlugosz, Cedar101, Frigoris, SmackBot, Bluebot, Stellar-TO, Oli Filth, Grapetonix, Dwmalone, Elryacko, Pipatron, Abednigo, Gwern, Glrx, Abatishchev, Creative1985, JL-Bot, Chieffuggybear, AnomieBOT, Dewritech and Anonymous: 13
- **Fowler–Noll–Vo hash function** *Source:* <http://en.wikipedia.org/wiki/Fowler%20%80%93Noll%20%80%93Vo%20hash%20function?oldid=649531941> *Contributors:* Damian Yerrick, The Anome, David spector, Jeremycole, Enochlau, Elf-friend, Jorge Stolfi, Uzume, Raph Levien, Boredzo, Apoc2400, Runtime, Ron Ritzman, Woohookitty, Qwertyus, Bubba73, Bachrach44, Rbarreira, Ospalh, Bluebot, Frap, Cybercobra, CRGreathouse, Mojo Hand, Landon Curt Noll, Gwern, Iambicking, Liko81, Raymondwinn, Svick, PipepBot, PixelBot, HumphreyW, Addbot, Luckas-bot, Yobot, Denispire, AnomieBOT, TruthIIPOWER, Phil Spectre, Filosoficos, GabrielReid, Traviswebb and Anonymous: 20
- **Bitstate hashing** *Source:* <http://en.wikipedia.org/wiki/Bitstate%20hashing?oldid=638339555> *Contributors:* Andreas Kaufmann, Jirislaby, Mild Bill Hiccup, UnCatBot, Paalappoo, Lemnaminor and Anonymous: 1
- **Bloom filter** *Source:* <http://en.wikipedia.org/wiki/Bloom%20filter?oldid=655269621> *Contributors:* Damian Yerrick, The Anome, Edward, Michael Hardy, Pnm, Wwwwolf, Thebramp, Charles Matthews, Dcoetzee, Furrykef, Phil Boswell, Fredrik, Chocolateboy, Babbage, Alan Liefing, Giftlite, DavidCary, ShaunMacPherson, Rchandra, Macrakis, Neilc, EvilGrin, James A. Donald, Two Bananas, Andreas Kaufmann, Anders94, Subrabbit, Smyth, Agl~enwiki, CanisRufus, Susvolans, Giraffedata, Drangon, Terrycojones, Mbloore, Yin~enwiki, Dzhim, GiovanniS, Galaxiaad, Mindmatrix, Shreevatsa, RzR~enwiki, Tabletop, Paynard, Ryan Reich, Pfunk42, Qwertyus, Ses4j, Rjwilmsi, Sdornan, Brighterorange, Vsriram, Quuxplusone, Chobot, Wavelength, Argav, Taejo, CesarB's unprivileged account, Msikma, Dtреббин, Wirthi, Cconnect, HereToHelp, Rubicantoto, Sbassi, SmackBot, Stev0, MalafayaBot, Cybercobra, Xiphoris, Drae, Galaad2, Jeremy Banks, Shakeelmahate, Requestion, Krauss, Farzaneh, Hilgerdenaar, Lindsay658, Hanche, Headbomb, NavenduJain, QuiteUnusual, Marokwitz,

Labongo, Bblfish, Igodard, Alexmadon, David Eppstein, STBot, Flexdream, Willpwillp, Osndok, Coolg49964, Jjldj, VolkovBot, Ferzkopp, LokiClock, Trachten, Rlauffer, SieBot, Emorrissey, Sswamida, Nnahmada, Svick, Justin W Smith, Gtoal, Rhubarbar, Quanstro, Pointillist, Shabbychef, Bender2k14, Sun Creator, AndreasBWagner, Sharma337, Dsimic, SteveJothen, Addbot, Mortense, Jerz4835, FrankAndProust, MrOllie, Lightbot, Legobot, Russianspy3, Luckas-bot, Yobot, Ptbotgourou, Amirobot, Gharb, AnomieBOT, Materialscientist, Citation bot, Naufraghi, Krj373, Osloom, X7q, Citation bot 1, Chenopodiaceous, HRoestBot, Jonesey95, Kronos04, Trappist the monk, Chronulator, Mavam, Buddeyp, RjwilmisBot, Liorithiel, Lesshaste, John of Reading, Drafiei, HiW-Bot, ZéroBot, Meng6, AManWithNoPlan, Ashish goel public, Jar354, ClueBot NG, Bpodgursky, Rezabot, Helpful Pixie Bot, DivineTraube, ErikDubbelboer, Exercisephys, Chmarkine, Williamdemeo, Akryzhn, Faizan, Lsmll, Everymorning, BloomFilterEditor, OriRottenstreich, Monkbot, Queelius, Epournaras and Anonymous: 184

- **Locality preserving hashing** *Source:* <http://en.wikipedia.org/wiki/Locality%20preserving%20hashing?oldid=501690233> *Contributors:* Jitse Niesen, Phil Boswell, BenFrantzDale, Mdd, Cataclysm, Ligulem, BetacommandBot, Zorakoid, Lourakis, Adverick~enwiki, Elaborating, Alok169 and Anonymous: 3
- **Zobrist hashing** *Source:* <http://en.wikipedia.org/wiki/Zobrist%20hashing?oldid=598540239> *Contributors:* Matthew Woodcraft, Charles Matthews, 777, Giftlite, Asimperson, ZeroOne, Evand, 99of9, Qwertyus, Dlugosz, Bluebot, Shirifan, DanielLemire, Jafet, Mattj2, WVhybrid, IanOsgood, Julian Cardona, Stephen Morley and Anonymous: 10
- **Rolling hash** *Source:* <http://en.wikipedia.org/wiki/Rolling%20hash?oldid=659461241> *Contributors:* DavidCary, Bo Lindbergh, CesarB's unprivileged account, Johndburger, SmackBot, Eug, Hegariz, A5b, DanielLemire, Martlau, Jafet, Alaibot, Bramschoenmakers, ClueBot, Cyril42e, Addbot, Yobot, FrescoBot, Makecat-bot, Obliviooustuna and Anonymous: 15
- **Hash list** *Source:* <http://en.wikipedia.org/wiki/Hash%20list?oldid=630578917> *Contributors:* M~enwiki, JonLS, Grm wnr, Homerjay, Davidgothberg, Ruud Koot, Jeff3000, Qwertyus, Digitalme, CesarB's unprivileged account, IstvanWolf, Harryboyles, CapitalR, CmdrObot, STBot, Addbot, Yobot, DataWraith, D climacus, K;ngdfhg, Jesse V., ChrisGualtieri and Anonymous: 13
- **Hash tree** *Source:* <http://en.wikipedia.org/wiki/Hash%20tree?oldid=649315442> *Contributors:* Qwertyus, Jarble, DPL bot and Anonymous: 1
- **Prefix hash tree** *Source:* <http://en.wikipedia.org/wiki/Prefix%20hash%20tree?oldid=656934074> *Contributors:* Dbenbenn, Intgr, Nad, Mcld, Cydebot, David Eppstein, The Thing That Should Not Be, Alexbot, Addbot, AnomieBOT, Miym, Jerluc and Anonymous: 4
- **Hash trie** *Source:* <http://en.wikipedia.org/wiki/Hash%20trie?oldid=641514466> *Contributors:* Gdr, Evand, Tr00st, Qwertyus, SmackBot, OrangeDog, Cydebot, MarshBot, Magioladitis, Pombredanne, Jarble, Jonesey95, Richlitt and Anonymous: 2
- **Hash array mapped trie** *Source:* <http://en.wikipedia.org/wiki/Hash%20array%20mapped%20trie?oldid=647630718> *Contributors:* Andreas Kaufmann, D6, Dgpop, Qwertyus, Quuxplusone, Nofxjunkie, HalfShadow, Wynand.winterbach, Gogo Dodo, Alaibot, Jrw@pobox.com, IMneme, Addbot, Yobot, Bunnyhop11, Axel22, AnomieBOT, Ehird, FrescoBot, Jesse V., DanielWaterworth, JensPetersen, Jessecooke, Erf5432, Chae slim, Jddixon and Anonymous: 10
- **Distributed hash table** *Source:* <http://en.wikipedia.org/wiki/Distributed%20hash%20table?oldid=661008084> *Contributors:* Bryan Derksen, The Anome, Edward, Nealmcb, TakuyaMurata, Alfio, Fcrick, Haakon, Ronz, Athymik~enwiki, Ehn, Hashar, Charles Matthews, Itai, Johnleach, GPHemsley, Jamesday, Phil Boswell, Bernhard Bauer, Yramesh, Hadal, UtherSRG, Seano1, Anthony, Diberri, Thv, Elf, ShaunMacPherson, Khalid hassani, ReinoutS, Naff89, Corti, Mike Rosoft, Jda, TedPavlic, Luqui, Wk muriithi, Irrbloss, Cwolfsheep, Godrickwok, Mdd, Gary, Mgrinich, CyberSkull, Minority Report, Apoc2400, EmilSit, Cburnett, Karnesky, The Belgain, Tabletop, Knuckles, Meneth, Eras-mus, Xiong Chiamiov, Enzo Aquarius, Search4Lancer, Xosé, X1987x, Intgr, Garas, Roboto de Ajvol, YurikBot, MMuzamills, Nad, Nethgirb, Cojoco, Eric.weigle, NeilN, Br~enwiki, Crystallina, KnightRider~enwiki, Erik Sandberg, SmackBot, F, Pgk, Mcld, Ohnoitsjamie, Bluebot, Morte, OrangeDog, Baa, Frap, Iammisc, JonHarder, FlyHigh, G-Bot~enwiki, Harryboyles, Oskilian, Nhorton, Roger pack, Michael miceli, Anescent, M.B~enwiki, Hu12, Gpierre, Dto, DJPhazer, Imeshev, Cydebot, Neustradamus, Thijs!bot, Ssspera, Marek69, TheJosh, CosineKitty, Nazlfrag, Bubba hotep, DerHexer, STBot, Shentino, Monkeyjunk, LordAnubisBOT, Hbpencil, OPless, Jnlin, VolkovBot, Jensocan, TelecomNut, Rogerdpack, Tomaxer, Bpringlemeir, Kbrose, Cacheonix, Alexbot, Thingg, DumZiBoT, Dsimic, MikeSchmid111, Addbot, Mabdul, Goofi1781, LaaknorBot, Ginosbot, Numbo3-bot, Yobot, Ptbotgourou, Old Death, Chy168, Götz, Squalho, Xqbot, Happyrabbit, Tomdo08, Miym, Chrismiceli, RobotBOT, Xubupt, W Nowicki, Sanpitch, Sae1962, Clsin, Louperibot, RedBot, Irvine.david, Tombeo, Liamzebedee, Maix, Noodles-sb, Ingenthhr, Brkt, Greywiz, EmausBot, Orphan Wiki, Md4567, Allquixotic, Scgtrp, Retsejesiw, Wayne Slam, ChuipastonBot, JaredThornbridge, CociBot, PetrIvanov, Mpias, Elauminri, John13535, Igstan, AstReach, Raghavendra.talur, Khiladi 2010, ChrisGualtieri, Altered Walter, Monkbot, JFregman, Graviton-Spark, Kevin at aerospike and Anonymous: 235
- **Consistent hashing** *Source:* <http://en.wikipedia.org/wiki/Consistent%20hashing?oldid=651527867> *Contributors:* Egil, DavidCary, Smyth, Dirkx, RoySmith, Skalet, Jamesfisher, Xavier Combelle, YurikBot, Wavelength, CesarB's unprivileged account, Gareth Jones, Nethgirb, DreamOfMirrors, Rene Mas, Dbenhur, Harrigan, Headbomb, JonGretar, Hut 8.5, BrotherE, STBot, Edratzer, SmallRepair, Uvaraj6, Extensive~enwiki, Rilak, Pbrandao, Addbot, Fyral, Numbo3-bot, Lightbot, Harald Haugland, Yobot, Gilo1969, Miym, X7q, Sae1962, DrilBot, Simongdkelly, Argv0, Tim Goddard, ZéroBot, McLaughlin77, Ardha jaya, Rohinidrathod, Richagandhewar, Mukkiepc, ChrisGualtieri and Anonymous: 50
- **Stable hashing** *Source:* <http://en.wikipedia.org/wiki/Stable%20hashing?oldid=591239250> *Contributors:* The Anome, Rpyle731, YUL89YYZ, Alex Kapranoff, CesarB's unprivileged account, Alasdair, SmackBot, TheBlueFlamingo, Minna Sora no Shita, Alaibot, Ssilverm, Addbot, Erik9bot and Anonymous: 2
- **Koorde** *Source:* <http://en.wikipedia.org/wiki/Koerde?oldid=568905267> *Contributors:* D6, Justinklebar, Bibliomaniac15, Bchociej, Kylu, Cydebot, VoABot II, JPG-GR, Kbrose, Suhhy, UnCatBot, Fwippy, Jorgeantibes, Bladegirl, Wekkolin, BG19bot and Anonymous: 5
- **Graph (abstract data type)** *Source:* [http://en.wikipedia.org/wiki/Graph%20\(abstract%20data%20type\)?oldid=657116454](http://en.wikipedia.org/wiki/Graph%20(abstract%20data%20type)?oldid=657116454) *Contributors:* Booyabazooka, Timwi, Dcoetze, Dysprosia, Zoicon5, Robbot, ZorroIII, KellyCoinGuy, Ruakh, Giftlite, P0nc, Jorge Stolfi, Tyr, Andreas Kaufmann, Max Terry, Simonfairfax, EmilJ, Any Key, R. S. Shaw, Jojit fb, Obradovic Goran, Liao, Rd232, Rabarberski, Epimethius, Oleg Alexandrov, Kendrick Hang, Ruud Koot, Chochopk, Joe Decker, Gmelli, Salix alba, Klortho, FedericoMenaQuintero, Chrisholland, Lt-wiki-bot, TFloto, Bruyninc~enwiki, RG2, SmackBot, Pieleric, Alink, JonHarder, Cybercobra, Jon Awbrey, Pbirnie, Kazubon~enwiki, Alaibot, Graphicalx, Stphung, Saimhe, QuiteUnusual, SiobhanHansa, Cooldudefx, David Eppstein, C4Cypher, ScaledLizard, NerdyNSK, Juliancolton, Idioma-bot, Gvanrossum, Natg 19, Rhanekom, Kbrose, Bentogoa, Pm master, Skippydo, Hariva, 31stCenturyMatt, ClueBot, ScNewcastle, UKoch, Excirial, Labraun90, Gallando, Rborrego, DumZiBoT, Avoided, Kate4341, Nmz787, Bluebusy, Ovi 1, AnomieBOT, Kristjan.Jonasson, FrescoBot, Hobsonlane, Sae1962, Aizquier, SimonFuhrmann, Electro, Canuckian89, DRAGON BOOSTER, Aaronzat, Cubancigar11, AvicAWB, ZweiOhren, Steven Hudak, A Aleg, ClueBot NG, MerllwBot, Helpful Pixie Bot, Bg9989, Maria kishore, Spydermanga and Anonymous: 104

- **Adjacency list** *Source:* <http://en.wikipedia.org/wiki/Adjacency%20list?oldid=650258645> *Contributors:* Michael Hardy, Booyabazooka, Kku, Schneelocke, Dcoetzee, Dysprosia, Jwpurple, Fredrik, MathMartin, Giftlite, Andreas Kaufmann, Chmod007, KrazeIke, Velella, Oleg Alexandrov, Qwertus, Ash211, NSR, Rdude, SmackBot, Chris the speller, Bluebot, Beetstra, Iridescent, Only2sea, David Eppstein, Cobi, Jamelan, Hariva, Justin W Smith, Garyzx, Addbot, Fgnievinski, Dtbullock, ThE cRaCkEr, Dminkovsky, Wikieditoroftoday, Citation bot, Twri, SPTWriter, Omnipaedita, Bobbertface, Hobsonlane, Craig Pemberton, Ricardo Ferreira de Oliveira, Patmorin, Venkatraghavang, Serketan, Sabrinamagers, MerIwBot, Chmarkine, Dessaya, SteenthIWbot, Stamptrader, Aviggiano and Anonymous: 31
- **Adjacency matrix** *Source:* <http://en.wikipedia.org/wiki/Adjacency%20matrix?oldid=652744218> *Contributors:* AxelBoldt, Tbackstr, Tomo, Michael Hardy, Booyabazooka, TakuyaMurata, Mbogelund, Schneelocke, Dcoetzee, Dysprosia, Reina riemann, Aleph4, Fredrik, MathMartin, Bkell, ElBenevolente, Giftlite, BenFrantzDale, Arthouse~enwiki, MarkSweep, Tomruen, Abdull, Rich Farmbrough, Bender235, Zaslav, Gauge, Rgdboer, Phils, Burn, Cburnett, Oleg Alexandrov, Timendum, YurikBot, Jpbown, Jogers, Sneftel, SmackBot, Ttzz, Kneufeld, Senfo, Abha Jain, Tamfang, Juffi, EVula, Dreadstar, Mdrine, Paulish, Tim Q. Wells, Beetstra, Hu12, Rhetth, CmdrObot, Only2sea, Thijs!bot, Headbomb, Olenielsen, Salgueiro~enwiki, Natelewis, Xeno, JamesBWatson, David Eppstein, Yonidebot, Squids and Chips, VolkovBot, LokiClock, YuryKirienko, Morre~enwiki, Periergeia, JackSchmidt, Garyzx, Watchduck, Bender2k14, Yoav HaCohen, Addbot, Fgnievinski, LaaknorBot, Debresser, Matěj Grabovský, Luckas-bot, Calle, Milk's Favorite Bot II, AnomieBOT, Twri, ArthurBot, SPTWriter, Miym, Bitsianshash, X7q, Jean Honorio, Slawomir Bialy, Chenopodiaceous, RobinK, TobeBot, Patmorin, John of Reading, WikitanvirBot, Felix Hoffmann, ZéroBot, ChuispastonBot, Wcherowi, Helpful Pixie Bot, Shilpi4560, Snowcream, DarafshBot, Kompot 3 and Anonymous: 73
- **And-inverter graph** *Source:* <http://en.wikipedia.org/wiki/And-inverter%20graph?oldid=650981267> *Contributors:* Michael Hardy, Pigmorsch, Jlang, Andreas Kaufmann, Linas, Jonathan de Boyne Pollard, Ketiltrot, Rjwilmsi, Trovatore, Mikeblas, Bluebot, Jon Awbrey, Igor Markov, Gregbard, Alan Mishchenko, Andy Crews, Appraiser, Aaron Hurst, Adrianwn, Niceguyedc, DOI bot, Ettrig, Yobot, AnomieBOT, Citation bot 1, ClueBot NG, Nobletripe, Pkkao2, Monkbot and Anonymous: 5
- **Binary decision diagram** *Source:* <http://en.wikipedia.org/wiki/Binary%20decision%20diagram?oldid=648110417> *Contributors:* Taw, Heron, Michael Hardy, Charles Matthews, Greenrd, David.Monnaux, Rorro, Michael Snow, Gtrmp, Laudaka, Andris, Ryan Clark, Sam Hocevar, McCart42, Andreas Kaufmann, Jkl, Kakesson, Uli, EmilJ, AshtonBenson, Mdd, Dirk Beyer~enwiki, Sade, IMeowbot, Ruud Koot, YHoshua, Bluumoose, GregorB, Matumio, Qwertus, Kinu, Brighterorange, YurikBot, KSchutte, Trovatore, Mikeblas, SmackBot, Jcarroll, Karlbrace, LouScheffer, Derek\_farn, CmdrObot, Pce3@ij.net, Jay.Here, Ajo Mama, Bobke, Hermel, Nouiz, Karsten Strehl, David Eppstein, Hitanshu D, Boute, Rohit.nadig, Karltk, Rdhettiner, AMCCosta, Trivialist, Sun Creator, Addbot, YannTM, Zorrobot, Yobot, Amirobot, Jason Recliner, Esq., SailorH, Twri, J04n, Bigfootsbigfoot, Britlak, MondalorBot, Sirutan, EleferenBot, Ort43v, Elaz85, Tijfo098, Helpful Pixie Bot, Calabre1992, BG19bot, Happyuk, Divy.dv, ChrisGualtieri, Denim1965, Lone boatman, Mark viking, Behanddeem, Melcous, Damoname, Cjdrake1, JMP EAX and Anonymous: 87
- **Binary moment diagram** *Source:* <http://en.wikipedia.org/wiki/Binary%20moment%20diagram?oldid=451611787> *Contributors:* Taw, Michael Hardy, Andreas Kaufmann, SmackBot, Jon Awbrey, Headbomb and Anonymous: 2
- **Zero-suppressed decision diagram** *Source:* <http://en.wikipedia.org/wiki/Zero-suppressed%20decision%20diagram?oldid=633663562> *Contributors:* Taw, Michael Hardy, Chris~enwiki, Doradus, Bkell, Andreas Kaufmann, Eep<sup>2</sup>, Graham87, Qwertus, SmackBot, Esbenrune, Rebooted, Jon Awbrey, Waggers, Lexfridman, TubularWorld, XLinkBot, Yobot, Ybngalobill, Nysol and Anonymous: 4
- **Propositional directed acyclic graph** *Source:* <http://en.wikipedia.org/wiki/Propositional%20directed%20acyclic%20graph?oldid=561376261> *Contributors:* Selket, Andreas Kaufmann, BD2412, Brighterorange, Trovatore, Bluebot, Nbarth, CmdrObot, RUN, MetsBot, Aagtbdoua, DRap, Mvinyals, Dennis714, Tijfo098 and Anonymous: 2
- **Graph-structured stack** *Source:* <http://en.wikipedia.org/wiki/Graph-structured%20stack?oldid=638537771> *Contributors:* Charles Matthews, Jaredwf, Fredrik, Andreas Kaufmann, Djispiewak, SpaceMoose, FlaBot, SmackBot, TimBentley, Sadads, Addbot, Erik9bot, RaptureBot and Anonymous: 4
- **Scene graph** *Source:* <http://en.wikipedia.org/wiki/Scene%20graph?oldid=656765401> *Contributors:* May, Zundark, Ap, Nknight, Michael Hardy, CamTarn, Mortene, Docu, BAxelrod, Furykef, Fredrik, Altenmann, Lowellian, P0lyglut, Kmote, Cbraga, Lancekt, Lockeownzj00, Beland, Andreas Kaufmann, CALR, Mecanismo, CanisRufus, Reinyday, Cmdrjameson, Tony Sidaway, Cpu111, GregorB, Rufous, Josh Parris, Mushin, RussBot, JLaTondre, SmackBot, Davepape, Tommstein, Commander Keane bot, Bluebot, SynergyBlades, WillC2 45220, Cybercobra, Engwar, CTho, DMacks, Jmdyck, Stephanie40, C4Cypher, Amatheny, KickAssClown, VolkovBot, Wsriley, Mayalld, Eman-Wilm, PeterV1510, Leroyluc, DzzD, Addbot, Rivelveloz, AnomieBOT, Arun Geo John, Martin Kraus, J04n, Zenaan, FrescoBot, Sae1962, Jfmantis, Anita.havele, ClueBot NG, Sqzx, HasanTeczan, Mongo314159 and Anonymous: 69
- **Big O notation** *Source:* <http://en.wikipedia.org/wiki/Big%20O%20notation?oldid=659922424> *Contributors:* Damian Yerrick, AxelBoldt, LC~enwiki, Brion VIBBER, Zundark, The Anome, Taw, Jeronimo, Andre Engels, Dachshund, Danny, Arvindn, Toby Bartels, Miguel~enwiki, B4hand, Isis~enwiki, Stevertigo, Patrick, Michael Hardy, Booyabazooka, DniQ, Ixfd64, Dcljr, TakuyaMurata, Eric119, Ahoerstemeier, Mpagano, Stevenj, Den fjätradre ankan~enwiki, Dean p foster, Poor Yorick, Mxn, Charles Matthews, Timwi, Dcoetzee, Dysprosia, Gutza, Doradus, Prumpf, Bagsc, Fibonacci, Head, McKay, Shizhao, Elwoz, Fvw, PuzzletChung, Robbot, Jaredwf, Fredrik, R3m0t, Altenmann, MathMartin, Henrygb, Bkell, Modeha, Diberri, Jleedev, Enochlau, Toshia, Connelly, Giftlite, Smig, Fennec, Gene Ward Smith, BenFrantzDale, Curps, Elias, CyborgTosser, Leonard G., Prosfilaes, Macrakis, Mobius, Neilc, Plutor, Zowch, Baronjonas, Aelvin, Andreas Kaufmann, Abdull, Adashiel, Dhuss, Rfl, Felix Wiemann, Guanabot, KneeLess, Simon Fenney, Luqui, Ascánder, Paul August, Dmr2, DcoetzeeBot~enwiki, ZeroOne, Danakil, Spitzak, El C, EmilJ, Whosyourjudas, Mike Schwartz, LeonardoGregianin, Tritium6, Haham hanuka, 4v4l0n42, Anthony Appleyard, Patrick Lucas, ABCD, Sligocki, Stefan.karpinski, Wtmitchell, Rebroad, Suruena, Vedant, Scuirinæ, Hlg, MIT Trekkie, Oleg Alexandrov, Simetrical, Linas, Mindmatrix, Shreevatsa, LOL, Oliphant, Jacobolus, Ruud Koot, Hdante, MFH, Dionyzz, Btyner, Graham87, Qwertus, NeoUrfahreran, JIP, AllanBz, DrHow, Rjwilmsi, Lugnad, JHMM13, R.e.b., Bubba73, Gadig, Drpaule, Leithp, Duagloth, Sydbarrett74, RexNL, Czar, Intgr, O Pavlos, Tardis, Comptotatoj, Kri, Chobot, Stephen Compall, Jpkotta, Wavelength, Borgx, Vector, Michael Slone, Koffieyahoo, Cookie4869~enwiki, Bhny, FauxFaux, Buster79, Nils Grimsmo, Yarin Kaul, Addps4cat, Smaines, Ott2, Arthur Rubin, RobertBorgersen, Davidwt, Draco flavus, Whouk, Zvika, Efnar, SmackBot, Maksim-e~enwiki, Apanag, InverseHypercube, Rovenhot, Nejko, Stimp, AnOddName, Gilliam, JoeKearney, Optikos, Bird of paradox, Alan smithee, SchfiftyThree, Nbarth, Gracenotes, Foxjwill, Shalom Yechiel, Cybercobra, Epachamo, Ligulembot, Michael Rogers, Lambiam, ArglebargleIV, Quendus, Mad Jaqk, ZAB, Breno, JoshuaZ, James.S, Elevenelevens, A-Ge0, Javit, Rschwieb, Alex Selby, Zero sharp, Tony Fox, A. Pichler, Rgiuly, Tawkerbot2, GeordieMcBain, Mapsax, CRGreathouse, Mcstrother, HenningThielemann, Mattbuck, Jowan2005, FilipeS, Gremagor, Pete4512, Jthillik, Mitchoyoshitaka, D4g0thur, Ultimus, Headbomb, Ichernev, WinBot, Ben pcc, BMB, Opelio, Tyco.skinner, Fayenatic london, VictorAnyakin, Josephjeevan, Hermel, JAnDbot, CosineKitty, Thenub314, Dricherby, PhilKnight, MSBOT, Eus Kevin, Arno Matthias, Ling.Nut, Riceplaytexas, MetsBot, David Eppstein, User A1, JoergenB, ChazBeckett, Raknarf44, Gjd001, ChrisForno,

Glrx, R'n'B, Jonathanzung, TheSeven, Derlay, Tarotcards, Mstuomel, Shoessss, Jwh335, CBKAtTopsails, Matiasholte, Skaraoke, NehpestTheFirst, Paxcoder, Szepi~enwiki, Philip Trueman, Anonymous Dissident, PaulTanenbaum, David Condrey, Rogerdpack, Lamro, C45207, PGWG, Spur, Quietbritishjim, SieBot, Shellgirl, TNARasslin, Wikibuki, Arunmoezh, Ernie shoemaker, Reinderien, Taemyr, KoenDelaere, User5910, JsePrometheus, Svick, CharlesGillingham, Melcombe, ClueBot, Justin W Smith, Alksentrs, P cuff, NovaDog, Vanisheduser12a67, Dadudadu, MarkOlah, Alexbot, Cbarlow3, Bergstra, Arjayay, Hans Adler, Andreasabel, JohnWStockwell, Qwfp, DumZiBoT, XLinkBot, Marc van Leeuwen, Stickee, Katsushi, PL290, Sameer0s, Sabalka, Addbot, EconoPhysicist, DFS454, Dr. Universe, SamatBot, Barak Sh, Calculuslover, Tide rolls, Matěj Grabovský, Leycec, Legobot, Luckas-bot, Yobot, Donfbreed, Kan8eDie, Writer on wiki, AnomieBOT, Jim1138, Royote, Citation bot, Jolsfa123, Flouran, Miym, Mrypsilon, Colfulus, SassoBot, Sophus Bie, Universalss, FrescoBot, ScotsmanRS, Kenfyre, Citation bot 1, Dark Charles, Sintharas, Ahmad Faridi, RedBot, Diego diaz espinosa, Île flottante, RobinK, H.ehsaan, Elephant in a tornado, Deeparnab, WavePart, EmausBot, Najeeb1010, Syncategoremeta, Netheril96, ZéroBot, Glassmage, AvicAWB, QEDK, RicoRico, Algoman101, Bomazi, Sapphorain, ClueBot NG, This lousy T-shirt, FiachraByrne, Widr, The-BiggestFootballFan, Helpful Pixie Bot, Curb Chain, Calabe1992, Koertefa, BG19bot, Walrus068, Meldraft, Guy vandegrift, Rockingravi, Brad7777, Yaroslav Nikitenko, BattyBot, ChrisGualtieri, ZiggyMo, RandomNoob143, Jochen Burghardt, Andyhowlett, Ngorade, ZX95, Zeitgeist2.718, Retrolord, Gupta.sumedha, Mammanmamamama, Melcous, Dhruven1600 and Anonymous: 527

- **Amortized analysis** *Source:* <http://en.wikipedia.org/wiki/Amortized%20analysis?oldid=656438688> *Contributors:* Michael Hardy, TakuyaMurata, Poor Yorick, Dcoetze, Altenmann, Giftlite, Brona, Andreas Kaufmann, Qutezuce, Talldean, Caesura, Joriki, Brazzy, Nneonneo, Eubot, Mathbot, Rbonvall, Laubrau~enwiki, Jimp, RussBot, PrologFan, CWenger, Allens, SmackBot, Torzsmokus, Pierre de Lyon, Magioladitis, User A1, Oravec, R'n'B, Bse3, Mantipula, BotKung, Svick, Safek, BarretB, Addbot, EconoPhysicist, Worch, Jibble, AnomieBOT, Stevo2001, Jangirke, FrescoBot, Mike22120, Vramasub, ClueBot NG, Josephshanak, MrBlok, Widr, Cerabot~enwiki, Mayfanning7, Pelzflorian, SurendraMatavalam, ScottDNelson and Anonymous: 28
- **Locality of reference** *Source:* <http://en.wikipedia.org/wiki/Locality%20of%20reference?oldid=660019962> *Contributors:* The Anome, Kurt Jansson, B4hand, Pnm, Ixfd64, TakuyaMurata, Charles Matthews, Werner, Fredrik, Ee00224, Tobias Bergemann, BenFrantzDale, Mboverload, ShakataGaNai, Andreas Kaufmann, Brianhe, Themusicgod1, Phils, Einstein9073, Zawersh, Kenyon, Firstfron, Uncle G, Chasmartin, Kbdank71, Ecb29, Mirror Vax, Intgr, BMF81, Piet Delpot, BOT-Superzerocool, Dinnno~enwiki, SmackBot, DTM, Nbarth, JonHarder, Radagast83, A5b, Derek farn, John, 16@r, CmdrObot, Cydebot, Not-just-yet, Headbom, NocNokNeo, Alphachimpbot, JPG-GR, Cic, Felix Andrews, R'n'B, Cpiral, Uttar, Randomalous, Cyberjoac, Naroza, Jruderman, Prohlep, Adrianwn, PixelBot, Dsimic, Addbot, Silverrocker, Ben Wraith, Lightbot, OIEnglish, Yobot, Maverick1715, MauritsBot, GrouchoBot, Costaluz, Stephen Morley, I dream of horses, Surement, KatelynJohann, Helwr, EmausBot, KuduIO, Donner60, Gilderien, Dfarrell07, Widr, Proxyma, Hillbillyholiday, Btlastic, Marklakata and Anonymous: 52
- **Standard Template Library** *Source:* <http://en.wikipedia.org/wiki/Standard%20Template%20Library?oldid=661280152> *Contributors:* Zundark, Merphant, Frecklefoot, Modster, Komap, Shoaler, Alfio, Glenn, Aragorn2, Andres, Tacvek, Furrykef, Mrjeff, PuzzletChung, Robbot, Josh Cherry, Chris-gore, Mushroom, Tobias Bergemann, BenFrantzDale, Zigger, Curps, Jorend, Rookkey, Jason Quinn, Neilc, LiDaobing, Marc Mongenet, Kutulu, Flex, JTN, Discospinster, Smyth, Pavel Vozenilek, Pt, Barfooz, Keno, Spoon!, Smalljim, Deryck Chan, Minghong, RoySmith, Tony Sidaway, K3rb, Forderud, Jkt, Bratsche, Rjwilmsi, MarSch, Nneonneo, Avinashm, Drrngrvy, FlaBot, Ian Pitchford, Mirror Vax, Stoph, Lawrencegold, Alvin-cs, Chobot, RussBot, Piet Delpot, DanielKO, Toffile, Gaius Cornelius, Emuka~enwiki, Moritz, Crasshopper, Zzuuzz, CWenger, Whaa?, Benhoyt, Tyler Oderkirk, SmackBot, Mmernex, Hydrogen Iodide, Vald, Pieleric, An-oddName, Jibjibjib, Thumperward, Tavianator, Spurrrymoses, Sairahulreddy, Decltype, Cubbi, A5b, Cibu, Vincenzo.romano, Rijkbenik, Soumyasch, Norm mit, TingChong Ma, Dreftymac, JoeBot, Gustavo.mori, Jesse Viviano, Sdorrance, Penbat, Omnicog, Cydebot, Torc2, Streetraider, Thijs!bot, Pulseczar, Escarbott, Ahannani, Wiki0709, NapoliRoma, Sebor, Wmbolle, Nyq, Bwagstaff, David Eppstein, Gwern, Bobby, Paercebal, Uiewo, Nwbeeson, Chemisor, Saziel, Xerxesmine, Brainfsck, Quietbritishjim, SieBot, OutlawSipper, ChessKnught, Leushenko, ClueBot, Hadas583, Alksentrs, Pedram.salehpoor, Jjur, Martin Moene, Aprox, Atallcostsky, Williams, XLinkBot, Oğuz Erjin, Wolkykim, AleyCZ, Deineka, Addbot, GoldenMedian, Alex.mccarthy, Jyellott, Enerjazzer, Ollydbg, Lihaas, SpBot, SamatBot, Lightbot, Luckas-bot, Yobot, Muthdesigner, JonKalb, AnomieBOT, Joule36e5, Jjcolotti, 1exec1, Citation bot, Geregen2, LilHelpa, Xqbot, J04n, Pomoxis, W Nowicki, Sae1962, Citation bot 1, WikitanvirBot, R2100, TheGeomaster, H3llBot, Demonkoryu, StasMalyga, Ipsign, ChuispastonBot, Yoursunny, Mankarse, Strcat, Helpful Pixie Bot, Mark Arsten, Compfreak7, Happyuk, Andrew Helwer, Tkgauri, Lone boatman, Aleks-ger, Sshekh, RogueClay, ScotXW, Pavel Senatorov, Monkbot, Abdeljalil and Anonymous: 138

## 14.2 Images

- **File:1-dimensional-range-query.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/0/09/1-dimensional-range-query.svg> *License:* CC0 *Contributors:* TOwn work *Original artist:* Matthew Eastman (talk)
- **File:1-dimensional-range-tree.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/d/d9/1-dimensional-range-tree.svg> *License:* CCO *Contributors:* Own work *Original artist:* Matthew Eastman (talk)
- **File:2-3-4\_tree\_insert\_1.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/61/2-3-4\\_tree\\_insert\\_1.svg](http://upload.wikimedia.org/wikipedia/commons/6/61/2-3-4_tree_insert_1.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Chrismiceli
- **File:2-3-4\_tree\_insert\_2.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/9/99/2-3-4\\_tree\\_insert\\_2.svg](http://upload.wikimedia.org/wikipedia/commons/9/99/2-3-4_tree_insert_2.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Chrismiceli
- **File:2-3-4\_tree\_insert\_3.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/e9/2-3-4\\_tree\\_insert\\_3.svg](http://upload.wikimedia.org/wikipedia/commons/e/e9/2-3-4_tree_insert_3.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Chrismiceli
- **File:2-3-4\_tree\_insert\_4.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/6d/2-3-4\\_tree\\_insert\\_4.svg](http://upload.wikimedia.org/wikipedia/commons/6/6d/2-3-4_tree_insert_4.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Chrismiceli
- **File:2-3\_insertion.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/44/2-3\\_insertion.svg](http://upload.wikimedia.org/wikipedia/commons/4/44/2-3_insertion.svg) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Diaa abdelmoneim
- **File:3dtree.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/b/b6/3dtree.png> *License:* GPL *Contributors:* ? *Original artist:* ?
- **File:6n-graph2.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/2/28/6n-graph2.svg> *License:* Public domain *Contributors:* ? *Original artist:* ?

- **File:8bit-dynamiclist.gif** *Source:* <http://upload.wikimedia.org/wikipedia/commons/1/1d/8bit-dynamiclist.gif> *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* Seahen
- **File:AA\_Tree\_Shape\_Cases.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/dd/AA\\_Tree\\_Shape\\_Cases.svg](http://upload.wikimedia.org/wikipedia/commons/d/dd/AA_Tree_Shape_Cases.svg) *License:* CC-BY-SA-3.0 *Contributors:* My own work (in emacs) based on ASCII art in Wikipedia:en:AA tree *Original artist:* Why Not A Duck
- **File:AA\_Tree\_Skew2.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/e0/AA\\_Tree\\_Skew2.svg](http://upload.wikimedia.org/wikipedia/commons/e/e0/AA_Tree_Skew2.svg) *License:* CC BY-SA 3.0 *Contributors:* own work based on GFDL'ed freehand drawing *Original artist:* User:Why Not A Duck based on work by Wikipedia:en:User:Rkleckner
- **File:AA\_Tree\_Split2.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/0e/AA\\_Tree\\_Split2.svg](http://upload.wikimedia.org/wikipedia/commons/0/0e/AA_Tree_Split2.svg) *License:* CC BY-SA 3.0 *Contributors:* own work based on GFDL'ed freehand drawing *Original artist:* User:Why Not A Duck based on work by Wikipedia:en:User:Rkleckner
- **File:AVL\_Tree\_Rebalancing.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/f/f5/AVL\\_Tree\\_Rebalancing.svg](http://upload.wikimedia.org/wikipedia/commons/f/f5/AVL_Tree_Rebalancing.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work based on Tree\_Rebalancing.gif *Original artist:* CyHawk
- **File:AVLtreef.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/0/06/AVLtreef.svg> *License:* Public domain *Contributors:* Own work *Original artist:* User:Mikm
- **File:Abrahamowitz&Stegun.page97.agr.jpg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/2/2e/Abrahamowitz%26Stegun.page97.agr.jpg> *License:* CC BY 2.5 *Contributors:* Self-photographed *Original artist:* agr
- **File:Ambox\_contradict.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/2e/Ambox\\_contradict.svg](http://upload.wikimedia.org/wikipedia/commons/2/2e/Ambox_contradict.svg) *License:* Public domain *Contributors:* self-made using Image:Emblem-contradict.svg *Original artist:* penubag, Rugby471
- **File:Ambox\_important.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/b/b4/Ambox\\_important.svg](http://upload.wikimedia.org/wikipedia/commons/b/b4/Ambox_important.svg) *License:* Public domain *Contributors:* Own work, based off of Image:Ambox scales.svg *Original artist:* Dsmurat (talk · contribs)
- **File:Ambox\_wikify.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/e1/Ambox\\_wikify.svg](http://upload.wikimedia.org/wikipedia/commons/e/e1/Ambox_wikify.svg) *License:* Public domain *Contributors:* Own work *Original artist:* penubag
- **File:AmortizedPush.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/e/e5/AmortizedPush.png> *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* ScottDNelson
- **File:An\_example\_of\_how\_to\_find\_a\_string\_in\_a\_Patricia\_trie.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/63/An\\_example\\_of\\_how\\_to\\_find\\_a\\_string\\_in\\_a\\_Patricia\\_trie.png](http://upload.wikimedia.org/wikipedia/commons/6/63/An_example_of_how_to_find_a_string_in_a_Patricia_trie.png) *License:* CC BY-SA 3.0 *Contributors:* Microsoft Visio *Original artist:* Saffles
- **File:And-inverter-graph.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/3/3c/And-inverter-graph.png> *License:* CC-BY-SA-3.0 *Contributors:* Self-painted with inkscape, exported to png *Original artist:* Florian Pigorsch.
- **File:Array\_of\_array\_storage.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/01/Array\\_of\\_array\\_storage.svg](http://upload.wikimedia.org/wikipedia/commons/0/01/Array_of_array_storage.svg) *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:AttenuatedBloomFilter.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/0/0c/AttenuatedBloomFilter.png> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Akryzhn
- **File:B-tree.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/6/65/B-tree.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work based on [1]. *Original artist:* CyHawk
- **File:BDD.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/9/91/BDD.png> *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* The original uploader was IMeowbot at English Wikipedia
- **File:BDD2pdag.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/f/f4/BDD2pdag.png> *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* RUN at English Wikipedia
- **File:BDD2pdag\_simple.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/9/90/BDD2pdag\\_simple.svg](http://upload.wikimedia.org/wikipedia/commons/9/90/BDD2pdag_simple.svg) *License:* CC-BY-SA-3.0 *Contributors:* Self made from BDD2pdag\_simple.png (here and on English Wikipedia) *Original artist:* User:Selket and User:RUN (original)
- **File:BDD\_Variable\_Ordering\_Bad.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/28/BDD\\_Variable\\_Ordering\\_Bad.svg](http://upload.wikimedia.org/wikipedia/commons/2/28/BDD_Variable_Ordering_Bad.svg) *License:* CC-BY-SA-3.0 *Contributors:* self-made using CrocoPat, a tool for relational programming, and GraphViz dot, a tool for graph layout *Original artist:* Dirk Beyer
- **File:BDD\_Variable\_Ordering\_Good.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/4b/BDD\\_Variable\\_Ordering\\_Good.svg](http://upload.wikimedia.org/wikipedia/commons/4/4b/BDD_Variable_Ordering_Good.svg) *License:* CC-BY-SA-3.0 *Contributors:* self-made using CrocoPat, a tool for relational programming, and GraphViz dot, a tool for graph layout *Original artist:* Dirk Beyer
- **File:BDD\_simple.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/14/BDD\\_simple.svg](http://upload.wikimedia.org/wikipedia/commons/1/14/BDD_simple.svg) *License:* CC-BY-SA-3.0 *Contributors:* self-made using CrocoPat, a tool for relational programming, and GraphViz dot, a tool for graph layout *Original artist:* Dirk Beyer
- **File:BIGMIN.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/0/02/BIGMIN.svg> *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia by User:Patrick87. *Original artist:* Original raster version (BIGMIN.jpg): Hermann Tropf at de.wikipedia
- **File:B\_tree\_insertion\_example.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/33/B\\_tree\\_insertion\\_example.png](http://upload.wikimedia.org/wikipedia/commons/3/33/B_tree_insertion_example.png) *License:* Public domain *Contributors:* I drew it :) *Original artist:* User:Maxtremus
- **File:Beap.jpg** *Source:* <http://upload.wikimedia.org/wikipedia/en/8/8f/Beap.jpg> *License:* Cc-by-sa-3.0 *Contributors:* Own work *Original artist:* Ilyathemuromets (talk) (Uploads)
- **File:Big-O-notation.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/8/89/Big-O-notation.png> *License:* Public domain *Contributors:* Screenshot from Graph Software *Original artist:* wikt:User:Fede\_Reghe
- **File:Bin\_computational\_geometry.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/20/Bin\\_computational\\_geometry.png](http://upload.wikimedia.org/wikipedia/commons/2/20/Bin_computational_geometry.png) *License:* Public domain *Contributors:* Own work *Original artist:* Yumf (talk) (Uploads)

- **File:BinaryTreeRotations.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/4/43/BinaryTreeRotations.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Josell7
- **File:Binary\_Heap\_with\_Array\_Implementation.JPG** *Source:* [http://upload.wikimedia.org/wikipedia/commons/c/c4/Binary\\_Heap\\_with\\_Array\\_Implementation.JPG](http://upload.wikimedia.org/wikipedia/commons/c/c4/Binary_Heap_with_Array_Implementation.JPG) *License:* CC0 *Contributors:* I (Chris857 (talk)) created this work entirely by myself. *Original artist:* Chris857 (talk)
- **File:Binary\_search\_tree.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/da/Binary\\_search\\_tree.svg](http://upload.wikimedia.org/wikipedia/commons/d/da/Binary_search_tree.svg) *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Binary\_search\_tree\_delete.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/46/Binary\\_search\\_tree\\_delete.svg](http://upload.wikimedia.org/wikipedia/commons/4/46/Binary_search_tree_delete.svg) *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Binary\_tree.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/f/f7/Binary\\_tree.svg](http://upload.wikimedia.org/wikipedia/commons/f/f7/Binary_tree.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Derrick Coetze
- **File:Binary\_tree\_in\_array.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/86/Binary\\_tree\\_in\\_array.svg](http://upload.wikimedia.org/wikipedia/commons/8/86/Binary_tree_in_array.svg) *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Binomial-heap-13.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/6/61/Binomial-heap-13.svg> *License:* CC-BY-SA-3.0 *Contributors:* de:Bild:Binomial-heap-13.png by de:Benutzer:Koethnig *Original artist:* User:D0ktorz
- **File:Binomial\_Trees.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/c/cf/Binomial\\_Trees.svg](http://upload.wikimedia.org/wikipedia/commons/c/cf/Binomial_Trees.svg) *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Binomial\_heap\_merge1.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/9/9f/Binomial\\_heap\\_merge1.svg](http://upload.wikimedia.org/wikipedia/commons/9/9f/Binomial_heap_merge1.svg) *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* Lemontea
- **File:Binomial\_heap\_merge2.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/e8/Binomial\\_heap\\_merge2.svg](http://upload.wikimedia.org/wikipedia/commons/e/e8/Binomial_heap_merge2.svg) *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* Lemontea
- **File:Bloom\_filter.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/ac/Bloom\\_filter.svg](http://upload.wikimedia.org/wikipedia/commons/a/ac/Bloom_filter.svg) *License:* Public domain *Contributors:* self-made, originally for a talk at WADS 2007 *Original artist:* David Eppstein
- **File:Bloom\_filter\_fp\_probability.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/ef/Bloom\\_filter\\_fp\\_probability.svg](http://upload.wikimedia.org/wikipedia/commons/e/ef/Bloom_filter_fp_probability.svg) *License:* CC BY 3.0 *Contributors:* Own work *Original artist:* Jerz4835
- **File:Bloom\_filter\_speed.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/c/c4/Bloom\\_filter\\_speed.svg](http://upload.wikimedia.org/wikipedia/commons/c/c4/Bloom_filter_speed.svg) *License:* Public domain *Contributors:* Transferred from en.wikipedia to Commons by RMcPhillip using CommonsHelper. *Original artist:* Alexmadon at English Wikipedia
- **File:Bplustree.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/3/37/Bplustree.png> *License:* CC BY 3.0 *Contributors:* Own work *Original artist:* Grundprinzip
- **File:Bracketing\_pairs.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/3d/Bracketing\\_pairs.svg](http://upload.wikimedia.org/wikipedia/commons/3/3d/Bracketing_pairs.svg) *License:* Public domain *Contributors:* Own work *Original artist:* David Eppstein
- **File:Bxtree.PNG** *Source:* <http://upload.wikimedia.org/wikipedia/commons/4/4c/Bxtree.PNG> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Bxtree
- **File:CPT-LinkedLists-addingnode.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/4/4b/CPT-LinkedLists-addingnode.svg> *License:* Public domain *Contributors:*
- **Singly\_linked\_list\_insert\_after.png** *Original artist:* Singly\_linked\_list\_insert\_after.png: Derrick Coetze
- **File:CPT-LinkedLists-deletingnode.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/d/d4/CPT-LinkedLists-deletingnode.svg> *License:* Public domain *Contributors:*
- **Singly\_linked\_list\_delete\_after.png** *Original artist:* Singly\_linked\_list\_delete\_after.png: Derrick Coetze
- **File:Cartesian\_tree.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/d5/Cartesian\\_tree.svg](http://upload.wikimedia.org/wikipedia/commons/d/d5/Cartesian_tree.svg) *License:* Public domain *Contributors:* Own work *Original artist:* David Eppstein
- **File:Cartesian\_tree\_range\_searching.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/de/Cartesian\\_tree\\_range\\_searching.svg](http://upload.wikimedia.org/wikipedia/commons/d/de/Cartesian_tree_range_searching.svg) *License:* Public domain *Contributors:* Own work *Original artist:* David Eppstein
- **File:Check\_if\_a\_point\_overlaps\_the\_intervals\_in\_S\_center\_of\_a\_centered\_interval\_tree.svg** *Source:* [http://upload.wikimedia.org/wikipedia/en/0/0d/Check\\_if\\_a\\_point\\_overlaps\\_the\\_intervals\\_in\\_S\\_center\\_of\\_a\\_centered\\_interval\\_tree.svg](http://upload.wikimedia.org/wikipedia/en/0/0d/Check_if_a_point_overlaps_the_intervals_in_S_center_of_a_centered_interval_tree.svg) *License:* CC-BY-SA-3.0 *Contributors:*  
I made this by Inkscape  
*Original artist:*  
Justin545
- **File:Circular\_buffer.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/b/b7/Circular\\_buffer.svg](http://upload.wikimedia.org/wikipedia/commons/b/b7/Circular_buffer.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_6789345.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/67/Circular\\_buffer\\_-\\_6789345.svg](http://upload.wikimedia.org/wikipedia/commons/6/67/Circular_buffer_-_6789345.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_6789AB5.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/b/ba/Circular\\_buffer\\_-\\_6789AB5.svg](http://upload.wikimedia.org/wikipedia/commons/b/ba/Circular_buffer_-_6789AB5.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_6789AB5\_empty.svg** *Source:* [http://upload.wikimedia.org/wikipedia/en/1/11/Circular\\_buffer\\_-\\_6789AB5\\_empty.svg](http://upload.wikimedia.org/wikipedia/en/1/11/Circular_buffer_-_6789AB5_empty.svg) *License:* Cc-by-sa-3.0 *Contributors:*  
Original from en:User:Cburnett, modified by de:User:DrZoom *Original artist:*  
en:User:Cburnett, modifications de:User:DrZoom

- **File:Circular\_buffer\_-\_6789AB5\_full.svg** *Source:* [http://upload.wikimedia.org/wikipedia/en/f/f4/Circular\\_buffer\\_-\\_6789AB5\\_full.svg](http://upload.wikimedia.org/wikipedia/en/f/f4/Circular_buffer_-_6789AB5_full.svg) *License:* Cc-by-sa-3.0 *Contributors:* Original from en:User:Cburnett, modified by de:User:DrZoom *Original artist:* en:User:Cburnett, modifications de:User:DrZoom
- **File:Circular\_buffer\_-\_6789AB5\_with\_pointers.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/05/Circular\\_buffer\\_-\\_6789AB5\\_with\\_pointers.svg](http://upload.wikimedia.org/wikipedia/commons/0/05/Circular_buffer_-_6789AB5_with_pointers.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_X789ABX.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/43/Circular\\_buffer\\_-\\_X789ABX.svg](http://upload.wikimedia.org/wikipedia/commons/4/43/Circular_buffer_-_X789ABX.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_XX123XX.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/d7/Circular\\_buffer\\_-\\_XX123XX.svg](http://upload.wikimedia.org/wikipedia/commons/d/d7/Circular_buffer_-_XX123XX.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_XX123XX\_with\_pointers.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/02/Circular\\_buffer\\_-\\_XX123XX\\_with\\_pointers.svg](http://upload.wikimedia.org/wikipedia/commons/0/02/Circular_buffer_-_XX123XX_with_pointers.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_XX1XXXX.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/89/Circular\\_buffer\\_-\\_XX1XXXX.svg](http://upload.wikimedia.org/wikipedia/commons/8/89/Circular_buffer_-_XX1XXXX.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_XXXX3XX.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/11/Circular\\_buffer\\_-\\_XXXX3XX.svg](http://upload.wikimedia.org/wikipedia/commons/1/11/Circular_buffer_-_XXXX3XX.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_empty.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/f/f7/Circular\\_buffer\\_-\\_empty.svg](http://upload.wikimedia.org/wikipedia/commons/f/f7/Circular_buffer_-_empty.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Circular\_buffer\_-\_mirror\_solution\_full\_and\_empty.svg** *Source:* [http://upload.wikimedia.org/wikipedia/en/d/d2/Circular\\_buffer\\_-\\_mirror\\_solution\\_full\\_and\\_empty.svg](http://upload.wikimedia.org/wikipedia/en/d/d2/Circular_buffer_-_mirror_solution_full_and_empty.svg) *License:* Cc-by-sa-3.0 *Contributors:* [http://en.wikipedia.org/wiki/File:Circular\\_buffer\\_-\\_6789AB5\\_empty.svg](http://en.wikipedia.org/wiki/File:Circular_buffer_-_6789AB5_empty.svg) *Original artist:* en:User:Cburnett
- **File:Circularly-linked-list.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/d/df/Circularly-linked-list.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Lasindi
- **File:CoalescedHash.jpg** *Source:* <http://upload.wikimedia.org/wikipedia/en/4/4c/CoalescedHash.jpg> *License:* PD *Contributors:* ? *Original artist:* ?
- **File:Commons-logo.svg** *Source:* <http://upload.wikimedia.org/wikipedia/en/4/4a/Commons-logo.svg> *License:* ? *Contributors:* ? *Original artist:* ?
- **File:CountAlgorithm.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/a/aa/CountAlgorithm.png> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* SaurabhKB
- **File:Crystal\_Clear\_app\_kedit.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/e8/Crystal\\_Clear\\_app\\_kedit.svg](http://upload.wikimedia.org/wikipedia/commons/e/e8/Crystal_Clear_app_kedit.svg) *License:* LGPL *Contributors:* Sabine MINICONI *Original artist:* Sabine MINICONI
- **File:Crystal\_Clear\_app\_network.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/49/Crystal\\_Clear\\_app\\_network.png](http://upload.wikimedia.org/wikipedia/commons/4/49/Crystal_Clear_app_network.png) *License:* LGPL *Contributors:* All Crystal Clear icons were posted by the author as LGPL on kde-look; *Original artist:* Everaldo Coelho and YellowIcon;
- **File:Ctrie-insert.png** *Source:* <http://upload.wikimedia.org/wikipedia/en/7/79/Ctrie-insert.png> *License:* CC0 *Contributors:* ? *Original artist:* ?
- **File:Cuckoo.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/d/de/Cuckoo.svg> *License:* CC BY-SA 3.0 *Contributors:* File: Cuckoo.png *Original artist:* Rasmus Pagh
- **File:DHT\_en.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/9/98/DHT\\_en.svg](http://upload.wikimedia.org/wikipedia/commons/9/98/DHT_en.svg) *License:* Public domain *Contributors:* Jnlin *Original artist:* Jnlin
- **File:Data\_Queue.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/52/Data\\_Queue.svg](http://upload.wikimedia.org/wikipedia/commons/5/52/Data_Queue.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* <a href='http://upload.wikimedia.org/wikipedia/commons/5/52/Data\_Queue.svg' class='internal' title='Data Queue.svg'>This Image</a> was created by User:Vegpuff.
- **File:Data\_stack.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/29/Data\\_stack.svg](http://upload.wikimedia.org/wikipedia/commons/2/29/Data_stack.svg) *License:* Public domain *Contributors:* made in Inkscape, by myself User:Bovie. Based on Image:Stack-sv.png, originally uploaded to the Swedish Wikipedia in 2004 by sv:User: Shrimp *Original artist:* User:Bovie
- **File:De\_bruijn\_graph-for\_binary\_sequence\_of\_order\_4.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/38/De\\_bruijn\\_graph-for\\_binary\\_sequence\\_of\\_order\\_4.svg](http://upload.wikimedia.org/wikipedia/commons/3/38/De_bruijn_graph-for_binary_sequence_of_order_4.svg) *License:* Public domain *Contributors:* <http://commons.wikimedia.org/wiki/File:Debruijngraph.gif> *Original artist:* SVG version: Wekkolin, Original version: english wikipedia user Michael\_Hardy
- **File:Deletion\_of\_internal\_binary\_tree\_node.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/31/Deletion\\_of\\_internal\\_binary\\_tree\\_node.svg](http://upload.wikimedia.org/wikipedia/commons/3/31/Deletion_of_internal_binary_tree_node.svg) *License:* CC0 *Contributors:* File:Deletion\_of\_internal\_binary\_tree\_node.JPG *Original artist:* Chris857
- **File:Directed.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/a/a2/Directed.svg> *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Directed\_Graph\_Edge.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/d7/Directed\\_Graph\\_Edge.svg](http://upload.wikimedia.org/wikipedia/commons/d/d7/Directed_Graph_Edge.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Johannes Rössel (talk)
- **File:Directed\_graph\_cyclic.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/1c/Directed\\_graph%2C\\_cyclic.svg](http://upload.wikimedia.org/wikipedia/commons/1/1c/Directed_graph%2C_cyclic.svg) *License:* Public domain *Contributors:* Transferred from de.wikipedia Transfer was stated to be made by User:Ddxc. *Original artist:* Original uploader was David W. at de.wikipedia

- **File:Directed\_graph\_disjoint.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/7/78/Directed\\_graph%2C\\_disjoint.svg](http://upload.wikimedia.org/wikipedia/commons/7/78/Directed_graph%2C_disjoint.svg) *License:* Public domain *Contributors:* Transferred from de.wikipedia Transfer was stated to be made by User:Ddxc. *Original artist:* Original uploader was David W. at de.wikipedia
- **File:Directed\_graph\_with\_branching\_SVG.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/a3/Directed\\_graph\\_with\\_branching\\_SVG.svg](http://upload.wikimedia.org/wikipedia/commons/a/a3/Directed_graph_with_branching_SVG.svg) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Limaner
- **File:Disambig\_gray.svg** *Source:* [http://upload.wikimedia.org/wikipedia/en/5/5f/Disambig\\_gray.svg](http://upload.wikimedia.org/wikipedia/en/5/5f/Disambig_gray.svg) *License:* Cc-by-sa-3.0 *Contributors:* ? *Original artist:* ?
- **File:Doubly-linked-list.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/5/5e/Doubly-linked-list.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Lasindi
- **File:Dsu\_disjoint\_sets\_final.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/ac/Dsu\\_disjoint\\_sets\\_final.svg](http://upload.wikimedia.org/wikipedia/commons/a/ac/Dsu_disjoint_sets_final.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* 93willy
- **File:Dsu\_disjoint\_sets\_init.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/67/Dsu\\_disjoint\\_sets\\_init.svg](http://upload.wikimedia.org/wikipedia/commons/6/67/Dsu_disjoint_sets_init.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* 93willy
- **File:Dynamic\_array.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/31/Dynamic\\_array.svg](http://upload.wikimedia.org/wikipedia/commons/3/31/Dynamic_array.svg) *License:* CC0 *Contributors:* Own work *Original artist:* Dcoetze
- **File>Edit-clear.svg** *Source:* <http://upload.wikimedia.org/wikipedia/en/f/f2/Edit-clear.svg> *License:* Public domain *Contributors:* The Tango! Desktop Project. *Original artist:* The people from the Tango! project. And according to the meta-data in the file, specifically: “Andreas Nilsson, and Jakub Steiner (although minimally).”
- **File:Emoji\_u1f4be.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/f/fb/Emoji\\_u1f4be.svg](http://upload.wikimedia.org/wikipedia/commons/f/fb/Emoji_u1f4be.svg) *License:* Apache License 2.0 *Contributors:* <https://code.google.com/p/noto/> *Original artist:* Google
- **File:Example\_of\_BSP\_tree\_construction\_-\_step\_1.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/5f/Example\\_of\\_BSP\\_tree\\_construction\\_-\\_step\\_1.svg](http://upload.wikimedia.org/wikipedia/commons/5/5f/Example_of_BSP_tree_construction_-_step_1.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons by Zahnradzacken. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_BSP\_tree\_construction\_-\_step\_2.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/84/Example\\_of\\_BSP\\_tree\\_construction\\_-\\_step\\_2.svg](http://upload.wikimedia.org/wikipedia/commons/8/84/Example_of_BSP_tree_construction_-_step_2.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons by Zahnradzacken. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_BSP\_tree\_construction\_-\_step\_3.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/4e/Example\\_of\\_BSP\\_tree\\_construction\\_-\\_step\\_3.svg](http://upload.wikimedia.org/wikipedia/commons/4/4e/Example_of_BSP_tree_construction_-_step_3.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons by Zahnradzacken. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_BSP\_tree\_construction\_-\_step\_4.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/4b/Example\\_of\\_BSP\\_tree\\_construction\\_-\\_step\\_4.svg](http://upload.wikimedia.org/wikipedia/commons/4/4b/Example_of_BSP_tree_construction_-_step_4.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_BSP\_tree\_construction\_-\_step\_5.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/46/Example\\_of\\_BSP\\_tree\\_construction\\_-\\_step\\_5.svg](http://upload.wikimedia.org/wikipedia/commons/4/46/Example_of_BSP_tree_construction_-_step_5.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_BSP\_tree\_construction\_-\_step\_6.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/de/Example\\_of\\_BSP\\_tree\\_construction\\_-\\_step\\_6.svg](http://upload.wikimedia.org/wikipedia/commons/d/de/Example_of_BSP_tree_construction_-_step_6.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_BSP\_tree\_construction\_-\_step\_7.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/58/Example\\_of\\_BSP\\_tree\\_construction\\_-\\_step\\_7.svg](http://upload.wikimedia.org/wikipedia/commons/5/58/Example_of_BSP_tree_construction_-_step_7.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_BSP\_tree\_construction\_-\_step\_8.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/e5/Example\\_of\\_BSP\\_tree\\_construction\\_-\\_step\\_8.svg](http://upload.wikimedia.org/wikipedia/commons/e/e5/Example_of_BSP_tree_construction_-_step_8.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_BSP\_tree\_construction\_-\_step\_9.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/59/Example\\_of\\_BSP\\_tree\\_construction\\_-\\_step\\_9.svg](http://upload.wikimedia.org/wikipedia/commons/5/59/Example_of_BSP_tree_construction_-_step_9.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_BSP\_tree\_traversal.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/08/Example\\_of\\_BSP\\_tree\\_traversal.svg](http://upload.wikimedia.org/wikipedia/commons/0/08/Example_of_BSP_tree_traversal.svg) *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* Chrisjohnson at English Wikipedia
- **File:Example\_of\_augmented\_tree\_with\_low\_value\_as\_the\_key\_and\_maximum\_high\_as\_extra\_annotation.png** *Source:* [http://upload.wikimedia.org/wikipedia/en/b/b0/Example\\_of\\_augmented\\_tree\\_with\\_low\\_value\\_as\\_the\\_key\\_and\\_maximum\\_high\\_as\\_extra\\_annotation.png](http://upload.wikimedia.org/wikipedia/en/b/b0/Example_of_augmented_tree_with_low_value_as_the_key_and_maximum_high_as_extra_annotation.png) *License:* GFDL *Contributors:* [https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy\\_data\\_structures\\_design.html#pbds.design.container.tree](https://gcc.gnu.org/onlinedocs/libstdc++/manual/policy_data_structures_design.html#pbds.design.container.tree) *Original artist:* Free Software Foundation
- **File:Extendible\_hashing\_1.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/4e/Extendible\\_hashing\\_1.svg](http://upload.wikimedia.org/wikipedia/commons/4/4e/Extendible_hashing_1.svg) *License:* CC BY 3.0 *Contributors:* Own work *Original artist:* Svick
- **File:Extendible\_hashing\_2.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/b/bf/Extendible\\_hashing\\_2.svg](http://upload.wikimedia.org/wikipedia/commons/b/bf/Extendible_hashing_2.svg) *License:* CC BY 3.0 *Contributors:* Own work *Original artist:* Svick
- **File:Extendible\_hashing\_3.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/7/71/Extendible\\_hashing\\_3.svg](http://upload.wikimedia.org/wikipedia/commons/7/71/Extendible_hashing_3.svg) *License:* CC BY 3.0 *Contributors:* Own work *Original artist:* Svick

- **File:Extendible\_hashing\_4.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/ad/Extendible\\_hashing\\_4.svg](http://upload.wikimedia.org/wikipedia/commons/a/ad/Extendible_hashing_4.svg) *License:* CC BY 3.0 *Contributors:* Own work *Original artist:* Svick
- **File:Extendible\_hashing\_5.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/7/7c/Extendible\\_hashing\\_5.svg](http://upload.wikimedia.org/wikipedia/commons/7/7c/Extendible_hashing_5.svg) *License:* CC BY 3.0 *Contributors:* Own work *Original artist:* Svick
- **File:Extendible\_hashing\_6.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/ee/Extendible\\_hashing\\_6.svg](http://upload.wikimedia.org/wikipedia/commons/e/ee/Extendible_hashing_6.svg) *License:* CC BY 3.0 *Contributors:* Own work *Original artist:* Svick
- **File:Fibonacci\_heap-decreasekey.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/09/Fibonacci\\_heap-decreasekey.png](http://upload.wikimedia.org/wikipedia/commons/0/09/Fibonacci_heap-decreasekey.png) *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Fibonacci\_heap.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/45/Fibonacci\\_heap.png](http://upload.wikimedia.org/wikipedia/commons/4/45/Fibonacci_heap.png) *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Fibonacci\_heap\_extractmin1.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/56/Fibonacci\\_heap\\_extractmin1.png](http://upload.wikimedia.org/wikipedia/commons/5/56/Fibonacci_heap_extractmin1.png) *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Fibonacci\_heap\_extractmin2.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/9/95/Fibonacci\\_heap\\_extractmin2.png](http://upload.wikimedia.org/wikipedia/commons/9/95/Fibonacci_heap_extractmin2.png) *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Figure1\_left.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/c/c4/Figure1\\_left.gif](http://upload.wikimedia.org/wikipedia/commons/c/c4/Figure1_left.gif) *License:* Public domain *Contributors:* Own work *Original artist:* Okoky at English Wikipedia
- **File:Figure1\_right.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/6c/Figure1\\_right.gif](http://upload.wikimedia.org/wikipedia/commons/6/6c/Figure1_right.gif) *License:* Public domain *Contributors:* Own work by the original uploader *Original artist:* Okoky at English Wikipedia
- **File:Figure2\_Hilbert.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/5f/Figure2\\_Hilbert.gif](http://upload.wikimedia.org/wikipedia/commons/5/5f/Figure2_Hilbert.gif) *License:* Public domain *Contributors:* Own work by the original uploader *Original artist:* Okoky at English Wikipedia
- **File:Figure3\_data\_rects.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/8a/Figure3\\_data\\_rects.gif](http://upload.wikimedia.org/wikipedia/commons/8/8a/Figure3_data_rects.gif) *License:* Public domain *Contributors:* Own work by the original uploader *Original artist:* Okoky at English Wikipedia
- **File:Figure4\_file\_structure.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/51/Figure4\\_file\\_structure.gif](http://upload.wikimedia.org/wikipedia/commons/5/51/Figure4_file_structure.gif) *License:* Public domain *Contributors:* Own work by the original uploader *Original artist:* Okoky at English Wikipedia
- **File:Four-level\_Z.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/c/cd/Four-level\\_Z.svg](http://upload.wikimedia.org/wikipedia/commons/c/cd/Four-level_Z.svg) *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* David Eppstein, based on a image by Hesperian.
- **File:FusionTreeSketch.gif** *Source:* <http://upload.wikimedia.org/wikipedia/commons/8/8a/FusionTreeSketch.gif> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Vladfi
- **File:Graph-structured\_stack\_1\_-\_jaredwf.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/eb/Graph-structured\\_stack\\_1\\_-\\_jaredwf.png](http://upload.wikimedia.org/wikipedia/commons/e/eb/Graph-structured_stack_1_-_jaredwf.png) *License:* Public domain *Contributors:* Transferred from en.wikipedia *Original artist:* Original uploader was Jaredwf at en.wikipedia
- **File:Graph\_single\_node.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/e2/Graph\\_single\\_node.svg](http://upload.wikimedia.org/wikipedia/commons/e/e2/Graph_single_node.svg) *License:* Public domain *Contributors:* Transferred from de.wikipedia Transfer was stated to be made by User:Ddxc. *Original artist:* Original uploader was David W. at de.wikipedia
- **File:HASHTB12.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/9/90/HASHTB12.svg> *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Hash\_list.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/7/7d/Hash\\_list.svg](http://upload.wikimedia.org/wikipedia/commons/7/7d/Hash_list.svg) *License:* Public domain *Contributors:* Hash\_list.png *Original artist:* Hash\_list.png: Original uploader was Davidgothberg at en.wikipedia
- **File:Hash\_table\_3\_1\_1\_0\_1\_0\_0\_SP.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/7/7d/Hash\\_table\\_3\\_1\\_1\\_0\\_1\\_0\\_0\\_SP.svg](http://upload.wikimedia.org/wikipedia/commons/7/7d/Hash_table_3_1_1_0_1_0_0_SP.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash\_table\_4\_1\_0\_0\_0\_0\_0\_LL.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/2e/Hash\\_table\\_4\\_1\\_0\\_0\\_0\\_0\\_LL.svg](http://upload.wikimedia.org/wikipedia/commons/2/2e/Hash_table_4_1_0_0_0_0_LL.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash\_table\_4\_1\_1\_0\_0\_0\_0\_LL.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/7/71/Hash\\_table\\_4\\_1\\_1\\_0\\_0\\_0\\_0\\_LL.svg](http://upload.wikimedia.org/wikipedia/commons/7/71/Hash_table_4_1_1_0_0_0_0_LL.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash\_table\_4\_1\_1\_0\_0\_1\_0\_LL.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/58/Hash\\_table\\_4\\_1\\_1\\_0\\_0\\_1\\_0\\_LL.svg](http://upload.wikimedia.org/wikipedia/commons/5/58/Hash_table_4_1_1_0_0_1_0_LL.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash\_table\_5\_0\_1\_1\_1\_1\_0\_LL.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/5a/Hash\\_table\\_5\\_0\\_1\\_1\\_1\\_1\\_0\\_LL.svg](http://upload.wikimedia.org/wikipedia/commons/5/5a/Hash_table_5_0_1_1_1_1_0_LL.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash\_table\_5\_0\_1\_1\_1\_1\_0\_SP.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/b/bf/Hash\\_table\\_5\\_0\\_1\\_1\\_1\\_1\\_0\\_SP.svg](http://upload.wikimedia.org/wikipedia/commons/b/bf/Hash_table_5_0_1_1_1_1_0_SP.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash\_table\_5\_0\_1\_1\_1\_1\_1\_LL.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/d0/Hash\\_table\\_5\\_0\\_1\\_1\\_1\\_1\\_1\\_LL.svg](http://upload.wikimedia.org/wikipedia/commons/d/d0/Hash_table_5_0_1_1_1_1_1_LL.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Jorge Stolfi
- **File:Hash\_table\_average\_insertion\_time.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/1c/Hash\\_table\\_average\\_insertion\\_time.png](http://upload.wikimedia.org/wikipedia/commons/1/1c/Hash_table_average_insertion_time.png) *License:* Public domain *Contributors:* Author's Own Work. *Original artist:* Derrick Coetze (User:Dcoetze)
- **File:HashedArrayTree16.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/0/04/HashedArrayTree16.svg> *License:* CC-BY-SA-3.0 *Contributors:* Original PNG by MegaHasher; SVG by author *Original artist:* Surachit at en.wikipedia
- **File:Heap\_add\_step1.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/ac/Heap\\_add\\_step1.svg](http://upload.wikimedia.org/wikipedia/commons/a/ac/Heap_add_step1.svg) *License:* Public domain *Contributors:* Drawn in Inkscape by Ilmari Karonen. *Original artist:* Ilmari Karonen
- **File:Heap\_add\_step2.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/16/Heap\\_add\\_step2.svg](http://upload.wikimedia.org/wikipedia/commons/1/16/Heap_add_step2.svg) *License:* Public domain *Contributors:* Drawn in Inkscape by Ilmari Karonen. *Original artist:* Ilmari Karonen

- **File:Heap\_add\_step3.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/51/Heap\\_add\\_step3.svg](http://upload.wikimedia.org/wikipedia/commons/5/51/Heap_add_step3.svg) *License:* Public domain *Contributors:* Drawn in Inkscape by Ilmari Karonen. *Original artist:* Ilmari Karonen
- **File:Heap\_delete\_step0.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/1c/Heap\\_delete\\_step0.svg](http://upload.wikimedia.org/wikipedia/commons/1/1c/Heap_delete_step0.svg) *License:* Public domain *Contributors:* [http://en.wikipedia.org/wiki/File:Heap\\_add\\_step1.svg](http://en.wikipedia.org/wiki/File:Heap_add_step1.svg) *Original artist:* Ilmari Karonen
- **File:Heap\_remove\_step1.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/ee/Heap\\_remove\\_step1.svg](http://upload.wikimedia.org/wikipedia/commons/e/ee/Heap_remove_step1.svg) *License:* Public domain *Contributors:* Drawn in Inkscape by Ilmari Karonen. *Original artist:* Ilmari Karonen
- **File:Heap\_remove\_step2.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/22/Heap\\_remove\\_step2.svg](http://upload.wikimedia.org/wikipedia/commons/2/22/Heap_remove_step2.svg) *License:* Public domain *Contributors:* Drawn in Inkscape by Ilmari Karonen. *Original artist:* Ilmari Karonen
- **File:Implicitmaxkdtree.gif** *Source:* <http://upload.wikimedia.org/wikipedia/commons/b/b7/Implicitmaxkdtree.gif> *License:* Public domain *Contributors:* Own work *Original artist:* User:Genieser
- **File:Insertion\_of\_binary\_tree\_node.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/9/95/Insertion\\_of\\_binary\\_tree\\_node.svg](http://upload.wikimedia.org/wikipedia/commons/9/95/Insertion_of_binary_tree_node.svg) *License:* CC0 *Contributors:* File:Insertion of binary tree node.JPG *Original artist:* Chris857
- **File:Internet\_map\_1024.jpg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/d2/Internet\\_map\\_1024.jpg](http://upload.wikimedia.org/wikipedia/commons/d/d2/Internet_map_1024.jpg) *License:* CC BY 2.5 *Contributors:* Originally from the English Wikipedia; description page is/was here. *Original artist:* The Opte Project
- **File:Interpolation\_example\_linear.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/67/Interpolation\\_example\\_linear.svg](http://upload.wikimedia.org/wikipedia/commons/6/67/Interpolation_example_linear.svg) *License:* Public domain *Contributors:* self-made in Gnuplot *Original artist:* Berland
- **File:KDTree-animation.gif** *Source:* <http://upload.wikimedia.org/wikipedia/commons/9/9c/KDTree-animation.gif> *License:* CC BY-SA 3.0 *Contributors:* Transferred from en.wikipedia; transfer was stated to be made by User:KindDragon33. *Original artist:* User\_A1. Original uploader was User A1 at en.wikipedia
- **File:Kdtree\_2d.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/b/bf/Kdtree\\_2d.svg](http://upload.wikimedia.org/wikipedia/commons/b/bf/Kdtree_2d.svg) *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia; transfer was stated to be made by User:KindDragon33. *Original artist:* Original uploader was KiwiSunset at en.wikipedia
- **File:Koorde\_lookup\_code.JPG** *Source:* [http://upload.wikimedia.org/wikipedia/en/a/a5/Koorde\\_lookup\\_code.JPG](http://upload.wikimedia.org/wikipedia/en/a/a5/Koorde_lookup_code.JPG) *License:* PD *Contributors:* self-made *Original artist:* Suhhy
- **File:Koorde\_lookup\_routing.JPG** *Source:* [http://upload.wikimedia.org/wikipedia/en/e/e0/Koorde\\_lookup\\_routing.JPG](http://upload.wikimedia.org/wikipedia/en/e/e0/Koorde_lookup_routing.JPG) *License:* PD *Contributors:* self-made *Original artist:* Suhhy
- **File:LampFlowchart.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/9/91/LampFlowchart.svg> *License:* CC-BY-SA-3.0 *Contributors:* vector version of Image:LampFlowchart.png *Original artist:* svg by Booyabazooka
- **File:Lebesgue-3d-step2.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/5/58/Lebesgue-3d-step2.png> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Robert Dickau
- **File:Lebesgue-3d-step3.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/d/da/Lebesgue-3d-step3.png> *License:* CC BY-SA 3.0 *Contributors:* self-made, using Mathematica 6 *Original artist:* Robert Dickau
- **File:Leftist-trees-S-value.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/c/ce/Leftist-trees-S-value.svg> *License:* Public domain *Contributors:* Own work (Original text: *self-made*) *Original artist:* Computergeeksjw (talk)
- **File:LinkCutAccess1.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/a/aa/LinkCutAccess1.png> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Drrilll
- **File:LinkCutAccess2.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/7/72/LinkCutAccess2.png> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Drrilll
- **File:Linkcuttree1.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/9/98/Linkcuttree1.png> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Drrilll
- **File:M-tree\_built\_with\_MMRAd\_split.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/7/7b/M-tree\\_built\\_with\\_MMRAd\\_split.png](http://upload.wikimedia.org/wikipedia/commons/7/7b/M-tree_built_with_MMRAd_split.png) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Chire
- **File:MTF\_Algorithm.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/08/MTF\\_Algorithm.png](http://upload.wikimedia.org/wikipedia/commons/0/08/MTF_Algorithm.png) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* SaurabhKB
- **File:Max-Heap.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/3/38/Max-Heap.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Ermishin
- **File:Mergefrom.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/0/0f/Mergefrom.svg> *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Min-heap.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/6/69/Min-heap.png> *License:* Public domain *Contributors:* Transferred from en.wikipedia; Transfer was stated to be made by User:LeaW. *Original artist:* Original uploader was Vikingstad at en.wikipedia
- **File:Min-height-biased-leftist-tree-initialization-part1.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/3/35/Min-height-biased-leftist-tree-initialization-part1.png> *License:* Public domain *Contributors:* Own work *Original artist:* User:Buss
- **File:Min-height-biased-leftist-tree-initialization-part2.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/a/ae/Min-height-biased-leftist-tree-initialization-part2.png> *License:* Public domain *Contributors:* Own work *Original artist:* User:Buss

- **File:Min-height-biased-leftist-tree-initialization-part3.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/3/37/Min-height-biased-leftist-tree-initialization-part3.png> *License:* Public domain *Contributors:* Own work *Original artist:* User:Buss
  - **File:N-ary\_to\_binary.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/c/cd/N-ary\\_to\\_binary.svg](http://upload.wikimedia.org/wikipedia/commons/c/cd/N-ary_to_binary.svg) *License:* Public domain *Contributors:* Own work *Original artist:* CyHawk
  - **File:Normal\_Binary\_Tree.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/6a/Normal\\_Binary\\_Tree.png](http://upload.wikimedia.org/wikipedia/commons/6/6a/Normal_Binary_Tree.png) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Vaibhavvc1092
  - **File:Octree2.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/2/20/Octree2.svg> *License:* CC-BY-SA-3.0 *Contributors:* Own work *Original artist:* WhiteTimberwolf, PNG version: Nü
  - **File:Office-book.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/a/a8/Office-book.svg> *License:* Public domain *Contributors:* This and myself. *Original artist:* Chris Down/Tango project
  - **File:Patricia\_trie.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/ae/Patricia\\_trie.svg](http://upload.wikimedia.org/wikipedia/commons/a/ae/Patricia_trie.svg) *License:* CC BY 2.5 *Contributors:* Own work *Original artist:* Claudio Rocchini
  - **File:Point\_quadtree.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/8b/Point\\_quadtree.svg](http://upload.wikimedia.org/wikipedia/commons/8/8b/Point_quadtree.svg) *License:* Public domain *Contributors:* self-made; originally for a talk at the 21st ACM Symp. on Computational Geometry, Pisa, June 2005 *Original artist:* David Eppstein
  - **File:PointerImplementationOfATrie.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/5/5d/PointerImplementationOfATrie.svg> *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Qwertys
  - **File:ProgramCallStack2.png** *Source:* <http://upload.wikimedia.org/wikipedia/en/a/a7/ProgramCallStack2.png> *License:* PD *Contributors:* ? *Original artist:* ?
  - **File:Pseudo-code.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/4/4d/Pseudo-code.png> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Drill
  - **File:Purely\_functional\_list\_after.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/2e/Purely\\_functional\\_list\\_after.svg](http://upload.wikimedia.org/wikipedia/commons/2/2e/Purely_functional_list_after.svg) *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia; Transfer was stated to be made by User:sevela.p. *Original artist:* Original uploader was VineetKumar at en.wikipedia
  - **File:Purely\_functional\_list\_before.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/9/98/Purely\\_functional\\_list\\_before.svg](http://upload.wikimedia.org/wikipedia/commons/9/98/Purely_functional_list_before.svg) *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia; Transfer was stated to be made by User:sevela.p. *Original artist:* Original uploader was VineetKumar at en.wikipedia
  - **File:Purely\_functional\_tree\_after.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/56/Purely\\_functional\\_tree\\_after.svg](http://upload.wikimedia.org/wikipedia/commons/5/56/Purely_functional_tree_after.svg) *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia; Transfer was stated to be made by User:sevela.p. *Original artist:* Original uploader was VineetKumar at en.wikipedia
  - **File:Purely\_functional\_tree\_before.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/9/9c/Purely\\_functional\\_tree\\_before.svg](http://upload.wikimedia.org/wikipedia/commons/9/9c/Purely_functional_tree_before.svg) *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia; Transfer was stated to be made by User:sevela.p. *Original artist:* Original uploader was VineetKumar at en.wikipedia
  - **File:Quad\_tree\_bitmap.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/a0/Quad\\_tree\\_bitmap.svg](http://upload.wikimedia.org/wikipedia/commons/a/a0/Quad_tree_bitmap.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Wojciech Mula
  - **File:Queap.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/d/dc/Queap.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Patrice Arruda
  - **File:Question\_book-new.svg** *Source:* [http://upload.wikimedia.org/wikipedia/en/9/99/Question\\_book-new.svg](http://upload.wikimedia.org/wikipedia/en/9/99/Question_book-new.svg) *License:* Cc-by-sa-3.0 *Contributors:*
- Created from scratch in Adobe Illustrator. Based on Image:Question book.png created by User:Equazcion *Original artist:* Tkgd2007
- **File:R\*-tree\_built\_using\_topological\_split.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/4/46/R%2A-tree\\_built\\_using\\_topological\\_split.png](http://upload.wikimedia.org/wikipedia/commons/4/46/R%2A-tree_built_using_topological_split.png) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Chire
  - **File:R\*-tree\_bulk\_loaded\_with\_sort-tile-recursive.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/1e/R%2A-tree\\_bulk\\_loaded\\_with\\_sort-tile-recursive.png](http://upload.wikimedia.org/wikipedia/commons/1/1e/R%2A-tree_bulk_loaded_with_sort-tile-recursive.png) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Chire
  - **File:R-tree.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/6/6f/R-tree.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Skinkie, w:en:Radim Baca
  - **File:R-tree\_built\_with\_Ang-Tan\_linear\_split.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/8a/R-tree\\_built\\_with\\_Ang-Tan\\_linear\\_split.png](http://upload.wikimedia.org/wikipedia/commons/8/8a/R-tree_built_with_Ang-Tan_linear_split.png) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Chire
  - **File:R-tree\_built\_with\_Greenes\_Split.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/1d/R-tree\\_built\\_with\\_Greenes\\_Split.png](http://upload.wikimedia.org/wikipedia/commons/1/1d/R-tree_built_with_Greenes_Split.png) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Chire
  - **File:R-tree\_built\_with\_Guttman'{}s\_linear\_split.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/02/R-tree\\_built\\_with\\_Guttman%27s\\_linear\\_split.png](http://upload.wikimedia.org/wikipedia/commons/0/02/R-tree_built_with_Guttman%27s_linear_split.png) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Chire
  - **File:R-tree\_with\_Guttman'{}s\_quadratic\_split.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/5b/R-tree\\_with\\_Guttman%27s\\_quadratic\\_split.png](http://upload.wikimedia.org/wikipedia/commons/5/5b/R-tree_with_Guttman%27s_quadratic_split.png) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Chire
  - **File:RTree-Visualization-3D.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/5/57/RTree-Visualization-3D.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Chire
  - **File:RTree\_2D.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/00/RTree\\_2D.svg](http://upload.wikimedia.org/wikipedia/commons/0/00/RTree_2D.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Chire
  - **File:Red-black\_tree\_delete\_case\_2\_as\_svg.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/5c/Red-black\\_tree\\_delete\\_case\\_2\\_as\\_svg.svg](http://upload.wikimedia.org/wikipedia/commons/5/5c/Red-black_tree_delete_case_2_as_svg.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Abloomfi
  - **File:Red-black\_tree\_delete\_case\_3\_as\_svg.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/a0/Red-black\\_tree\\_delete\\_case\\_3\\_as\\_svg.svg](http://upload.wikimedia.org/wikipedia/commons/a/a0/Red-black_tree_delete_case_3_as_svg.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Abloomfi

- **File:Red-black\_tree\_delete\_case\_4\_as\_svg.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/3d/Red-black\\_tree\\_delete\\_case\\_4\\_as\\_svg.svg](http://upload.wikimedia.org/wikipedia/commons/3/3d/Red-black_tree_delete_case_4_as_svg.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Abloomfi
- **File:Red-black\_tree\_delete\_case\_5\_as\_svg.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/36/Red-black\\_tree\\_delete\\_case\\_5\\_as\\_svg.svg](http://upload.wikimedia.org/wikipedia/commons/3/36/Red-black_tree_delete_case_5_as_svg.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Abloomfi
- **File:Red-black\_tree\_delete\_case\_6\_as\_svg.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/9/99/Red-black\\_tree\\_delete\\_case\\_6\\_as\\_svg.svg](http://upload.wikimedia.org/wikipedia/commons/9/99/Red-black_tree_delete_case_6_as_svg.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Abloomfi
- **File:Red-black\_tree\_example.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/66/Red-black\\_tree\\_example.svg](http://upload.wikimedia.org/wikipedia/commons/6/66/Red-black_tree_example.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* en:User:Cburnett
- **File:Red-black\_tree\_example\_(B-tree\_analogy).svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/7/72/Red-black\\_tree-example\\_%28B-tree\\_analogy%29.svg](http://upload.wikimedia.org/wikipedia/commons/7/72/Red-black_tree-example_%28B-tree_analogy%29.svg) *License:* CC-BY-SA-3.0 *Contributors:* This vector image was created with Inkscape. *Original artist:* fr:Utilisateur:Verdy\_p
- **File:Red-black\_tree\_insert\_case\_3.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/d6/Red-black\\_tree\\_insert\\_case\\_3.svg](http://upload.wikimedia.org/wikipedia/commons/d/d6/Red-black_tree_insert_case_3.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Abloomfi
- **File:Red-black\_tree\_insert\_case\_4.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/89/Red-black\\_tree\\_insert\\_case\\_4.svg](http://upload.wikimedia.org/wikipedia/commons/8/89/Red-black_tree_insert_case_4.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Abloomfi
- **File:Red-black\_tree\_insert\_case\_5.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/dc/Red-black\\_tree\\_insert\\_case\\_5.svg](http://upload.wikimedia.org/wikipedia/commons/d/dc/Red-black_tree_insert_case_5.svg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Abloomfi
- **File:Red Black Shape Cases.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/f/f9/Red\\_Black\\_Shape\\_Cases.svg](http://upload.wikimedia.org/wikipedia/commons/f/f9/Red_Black_Shape_Cases.svg) *License:* CC-BY-SA-3.0 *Contributors:* My own work (in emacs) based on ASCII art in Wikipedia:en:AA tree *Original artist:* Why Not A Duck
- **File:SCD\_algebraic\_notation.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/b/b6/SCD\\_algebraic\\_notation.svg](http://upload.wikimedia.org/wikipedia/commons/b/b6/SCD_algebraic_notation.svg) *License:* CC BY-SA 3.0 *Contributors:*
- **Chess\_board\_blank.svg** *Original artist:* Chess\_board\_blank.svg: \*AAA\_SVG\_Chessboard\_and\_chess\_pieces\_04.svg: ILA-boy
- **File:SPQR\_tree\_2.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/55/SPQR\\_tree\\_2.svg](http://upload.wikimedia.org/wikipedia/commons/5/55/SPQR_tree_2.svg) *License:* CC0 *Contributors:* Own work *Original artist:* David Eppstein
- **File:Segment\_tree\_instance.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/e/e5/Segment\\_tree\\_instance.gif](http://upload.wikimedia.org/wikipedia/commons/e/e5/Segment_tree_instance.gif) *License:* Public domain *Contributors:* Own work *Original artist:* User:Alfredo J. Herrera Lago
- **File:Simple\_cycle\_graph.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/26/Simple\\_cycle\\_graph.svg](http://upload.wikimedia.org/wikipedia/commons/2/26/Simple_cycle_graph.svg) *License:* Public domain *Contributors:* Transferred from en.wikipedia *Original artist:* Booyabazooka at en.wikipedia
- **File:Single\_node1.jpg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/5/5d/Single\\_node1.jpg](http://upload.wikimedia.org/wikipedia/commons/5/5d/Single_node1.jpg) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* IshitaMundada
- **File:Singly-linked-list.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/6/6d/Singly-linked-list.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Lasindi
- **File:Singly\_linked\_list.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/37/Singly\\_linked\\_list.png](http://upload.wikimedia.org/wikipedia/commons/3/37/Singly_linked_list.png) *License:* Public domain *Contributors:* Copied from en. Originally uploaded by Dcoetzee. *Original artist:* Derrick Coetze (User:Dcoetze)
- **File:SkewHeapMerge1.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/1/19/SkewHeapMerge1.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Quintaylor
- **File:SkewHeapMerge2.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/9/90/SkewHeapMerge2.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Quintaylor
- **File:SkewHeapMerge3.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/a/ad/SkewHeapMerge3.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Quintaylor
- **File:SkewHeapMerge4.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/4/4b/SkewHeapMerge4.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Quintaylor
- **File:SkewHeapMerge5.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/c/c9/SkewHeapMerge5.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Quintaylor
- **File:SkewHeapMerge6.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/7/7d/SkewHeapMerge6.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Quintaylor
- **File:SkewHeapMerge7.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/4/4d/SkewHeapMerge7.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Quintaylor
- **File:Skip\_list.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/86/Skip\\_list.svg](http://upload.wikimedia.org/wikipedia/commons/8/86/Skip_list.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Wojciech Mula
- **File:Skip\_list\_add\_element-en.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/2c/Skip\\_list\\_add\\_element-en.gif](http://upload.wikimedia.org/wikipedia/commons/2/2c/Skip_list_add_element-en.gif) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Artyom Kalinin
- **File:Software\_spinner.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/82/Software\\_spinner.png](http://upload.wikimedia.org/wikipedia/commons/8/82/Software_spinner.png) *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia; transfer was stated to be made by User:Rockfang. *Original artist:* Original uploader was CharlesC at en.wikipedia
- **File:Spaghettistack.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/e/e5/Spaghettistack.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Ealf
- **File:Splay\_tree\_zig.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/2c/Splay\\_tree\\_zig.svg](http://upload.wikimedia.org/wikipedia/commons/2/2c/Splay_tree_zig.svg) *License:* CC-BY-SA-3.0 *Contributors:*
- **Zig.gif** *Original artist:* Zig.gif: User:Regnaron
- **File:Stacks\_jaredwf.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/0/09/Stacks\\_jaredwf.png](http://upload.wikimedia.org/wikipedia/commons/0/09/Stacks_jaredwf.png) *License:* CC-BY-SA-3.0 *Contributors:* [http://en.wikipedia.org/wiki/File:Stacks\\_jaredwf.png](http://en.wikipedia.org/wiki/File:Stacks_jaredwf.png) *Original artist:* jaredwf

- **File:Suffix\_tree\_ABAB\_BABA.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/6/64/Suffix\\_tree\\_ABAB\\_BABA.svg](http://upload.wikimedia.org/wikipedia/commons/6/64/Suffix_tree_ABAB_BABA.svg) *License:* Public domain *Contributors:*
- **Suffix\_tree\_ABAB\_BABA.png** *Original artist:*
- derivative work: Johannes Rössel (talk)
- **File:Suffix\_tree\_BANANA.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/d2/Suffix\\_tree\\_BANANA.svg](http://upload.wikimedia.org/wikipedia/commons/d/d2/Suffix_tree_BANANA.svg) *License:* Public domain *Contributors:* own work (largely based on PNG version by Nils Grimsmo) *Original artist:* Maciej Jaros (commons: Nux, wiki-pl: Nux) (PNG version by Nils Grimsmo)
- **File:Symmetric\_group\_4;\_Cayley\_graph\_1,5,21\_(Nauru\_Petersen);\_numbers.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/27/Symmetric\\_group\\_4%3B\\_Cayley\\_graph\\_1%2C5%2C21\\_%28Nauru\\_Petersen%29%3B\\_numbers.svg](http://upload.wikimedia.org/wikipedia/commons/2/27/Symmetric_group_4%3B_Cayley_graph_1%2C5%2C21_%28Nauru_Petersen%29%3B_numbers.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Lipedia
- **File:Symmetric\_group\_4;\_Cayley\_graph\_1,5,21\_(adjacency\_matrix).svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/17/Symmetric\\_group\\_4%3B\\_Cayley\\_graph\\_1%2C5%2C21\\_%28adjacency\\_matrix%29.svg](http://upload.wikimedia.org/wikipedia/commons/1/17/Symmetric_group_4%3B_Cayley_graph_1%2C5%2C21_%28adjacency_matrix%29.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Lipedia
- **File:Symmetric\_group\_4;\_Cayley\_graph\_4,9;\_numbers.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/86/Symmetric\\_group\\_4%3B\\_Cayley\\_graph\\_4%2C9%3B\\_numbers.svg](http://upload.wikimedia.org/wikipedia/commons/8/86/Symmetric_group_4%3B_Cayley_graph_4%2C9%3B_numbers.svg) *License:* Public domain *Contributors:*
- **GrapheCayley-S4-Plan.svg** *Original artist:* GrapheCayley-S4-Plan.svg: Fool (talk)
- **File:Symmetric\_group\_4;\_Cayley\_graph\_4,9\_(adjacency\_matrix).svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/22/Symmetric\\_group\\_4%3B\\_Cayley\\_graph\\_4%2C9\\_%28adjacency\\_matrix%29.svg](http://upload.wikimedia.org/wikipedia/commons/2/22/Symmetric_group_4%3B_Cayley_graph_4%2C9_%28adjacency_matrix%29.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Lipedia
- **File:T-tree-1.png** *Source:* <http://upload.wikimedia.org/wikipedia/commons/a/a9/T-tree-1.png> *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia *Original artist:* Original uploader was Hholzgra at en.wikipedia
- **File:T-tree-2.png** *Source:* <http://upload.wikimedia.org/wikipedia/en/c/ca/T-tree-2.png> *License:* Cc-by-sa-3.0 *Contributors:* ? *Original artist:* ?
- **File:Text\_document\_with\_red\_question\_mark.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/a4/Text\\_document\\_with\\_red\\_question\\_mark.svg](http://upload.wikimedia.org/wikipedia/commons/a/a4/Text_document_with_red_question_mark.svg) *License:* Public domain *Contributors:* Created by bdesham with Inkscape; based upon Text-x-generic.svg from the Tango project. *Original artist:* Benjamin D. Esham (bdesham)
- **File:ThreadTree\_Inorder\_Array.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/1f/ThreadTree\\_Inorder\\_Array.png](http://upload.wikimedia.org/wikipedia/commons/1/1f/ThreadTree_Inorder_Array.png) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Yeskw
- **File:ThreadTree\_Inorder\_Array123456789.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/2c/ThreadTree\\_Inorder\\_Array123456789.png](http://upload.wikimedia.org/wikipedia/commons/2/2c/ThreadTree_Inorder_Array123456789.png) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Yeskw
- **File:Threaded\_Binary\_Tree.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/8b/Threaded\\_Binary\\_Tree.png](http://upload.wikimedia.org/wikipedia/commons/8/8b/Threaded_Binary_Tree.png) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Vaibhavvc1092
- **File:Threaded\_tree.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/7/7a/Threaded\\_tree.svg](http://upload.wikimedia.org/wikipedia/commons/7/7a/Threaded_tree.svg) *License:* Public domain *Contributors:* self-made based on public domain Image:Sorted binary tree.svg *Original artist:* R. S. Shaw
- **File:Top\_tree.jpg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/f/fe/Top\\_tree.jpg](http://upload.wikimedia.org/wikipedia/commons/f/fe/Top_tree.jpg) *License:* Public domain *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* Afrozenator at English Wikipedia
- **File:Transpose\_Algorithm.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/30/Transpose\\_Algorithm.png](http://upload.wikimedia.org/wikipedia/commons/3/30/Transpose_Algorithm.png) *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* SaurabhKB
- **File:TreapAlphaKey.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/4/4b/TreapAlphaKey.svg> *License:* CC0 *Contributors:* Own work, with labels to match bitmap version *Original artist:* Qef
- **File:Tree\_0001.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/25/Tree\\_0001.svg](http://upload.wikimedia.org/wikipedia/commons/2/25/Tree_0001.svg) *License:* Public domain *Contributors:* Own work *Original artist:* MYguel
- **File:Tree\_Rebalancing.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/c/c4/Tree\\_Rebalancing.gif](http://upload.wikimedia.org/wikipedia/commons/c/c4/Tree_Rebalancing.gif) *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia; transferred to Commons by User:Common Good using CommonsHelper. *Original artist:* Original uploader was Mtanti at en.wikipedia
- **File:Tree\_Rotations.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/15/Tree\\_Rotations.gif](http://upload.wikimedia.org/wikipedia/commons/1/15/Tree_Rotations.gif) *License:* CC-BY-SA-3.0 *Contributors:* Transferred from en.wikipedia to Commons. *Original artist:* Mtanti at English Wikipedia
- **File:Tree\_rotation.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/23/Tree\\_rotation.png](http://upload.wikimedia.org/wikipedia/commons/2/23/Tree_rotation.png) *License:* CC-BY-SA-3.0 *Contributors:* EN-Wikipedia *Original artist:* User:Ramasamy
- **File:Tree\_rotation\_animation\_250x250.gif** *Source:* [http://upload.wikimedia.org/wikipedia/commons/3/31/Tree\\_rotation\\_animation\\_250x250.gif](http://upload.wikimedia.org/wikipedia/commons/3/31/Tree_rotation_animation_250x250.gif) *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Tar-Elessar
- **File:Trie-vs-minimal-acyclic-fa.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/9/9c/Trie-vs-minimal-acyclic-fa.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Chkno
- **File:Trie\_example.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/b/be/Trie\\_example.svg](http://upload.wikimedia.org/wikipedia/commons/b/be/Trie_example.svg) *License:* Public domain *Contributors:* own work (based on PNG image by Deco) *Original artist:* Booyabazooka (based on PNG image by Deco). Modifications by Superm401.
- **File:Unbalanced\_binary\_tree.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/a9/Unbalanced\\_binary\\_tree.svg](http://upload.wikimedia.org/wikipedia/commons/a/a9/Unbalanced_binary_tree.svg) *License:* Public domain *Contributors:* Own work *Original artist:* Me (Intgr)
- **File:Unbalanced\_scales.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/f/fe/Unbalanced\\_scales.svg](http://upload.wikimedia.org/wikipedia/commons/f/fe/Unbalanced_scales.svg) *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Unrolled\_linked\_lists\_(1-8).PNG** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/16/Unrolled\\_linked\\_lists\\_%281-8%29.PNG](http://upload.wikimedia.org/wikipedia/commons/1/16/Unrolled_linked_lists_%281-8%29.PNG) *License:* CC BY-SA 3.0 *Contributors:* Made by uploader (ref:[1]) *Original artist:* Shigeru23

- **File:VList\_example\_diagram.png** *Source:* [http://upload.wikimedia.org/wikipedia/commons/d/dc/VList\\_example\\_diagram.png](http://upload.wikimedia.org/wikipedia/commons/d/dc/VList_example_diagram.png) *License:* Public domain *Contributors:* ? *Original artist:* Derrick Coetze (User:Dcoetze)
- **File:VebDiagram.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/6/6b/VebDiagram.svg> *License:* Public domain *Contributors:* Own work *Original artist:* Gailcarmichael
- **File:Vector\_Rope\_concat.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/a/a0/Vector\\_Rope\\_concat.svg](http://upload.wikimedia.org/wikipedia/commons/a/a0/Vector_Rope_concat.svg) *License:* CC0 *Contributors:* Own work *Original artist:* Gringer (<a href='//commons.wikimedia.org/wiki/User\_talk:Gringer' title='User talk:Gringer'>talk</a>)
- **File:Vector\_Rope\_example.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/8/8a/Vector\\_Rope\\_example.svg](http://upload.wikimedia.org/wikipedia/commons/8/8a/Vector_Rope_example.svg) *License:* CC BY-SA 3.0 *Contributors:* [http://en.wikipedia.org/wiki/File:Rope\\_example.jpg](http://en.wikipedia.org/wiki/File:Rope_example.jpg) *Original artist:* Meng Yao
- **File:Vector\_Rope\_index.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/20/Vector\\_Rope\\_index.svg](http://upload.wikimedia.org/wikipedia/commons/2/20/Vector_Rope_index.svg) *License:* CC0 *Contributors:* Own work *Original artist:* Gringer (<a href='//commons.wikimedia.org/wiki/User\_talk:Gringer' title='User talk:Gringer'>talk</a>)
- **File:Vector\_Rope\_split.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/f/fd/Vector\\_Rope\\_split.svg](http://upload.wikimedia.org/wikipedia/commons/f/fd/Vector_Rope_split.svg) *License:* CC0 *Contributors:* ? *Original artist:* ?
- **File:Waldburg\_Ahnentafel.jpg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/2/26/Waldburg\\_Ahnentafel.jpg](http://upload.wikimedia.org/wikipedia/commons/2/26/Waldburg_Ahnentafel.jpg) *License:* Public domain *Contributors:* <http://www.ahneninfo.com/de/ahnentafel.htm> *Original artist:* Anonymous
- **File:Wiki\_letter\_w.svg** *Source:* [http://upload.wikimedia.org/wikipedia/en/6/6c/Wiki\\_letter\\_w.svg](http://upload.wikimedia.org/wikipedia/en/6/6c/Wiki_letter_w.svg) *License:* Cc-by-sa-3.0 *Contributors:* ? *Original artist:* ?
- **File:Wiki\_letter\_w\_cropped.svg** *Source:* [http://upload.wikimedia.org/wikipedia/commons/1/1c/Wiki\\_letter\\_w\\_cropped.svg](http://upload.wikimedia.org/wikipedia/commons/1/1c/Wiki_letter_w_cropped.svg) *License:* CC-BY-SA-3.0 *Contributors:*
- **Wiki\_letter\_w.svg** *Original artist:* Wiki\_letter\_w.svg: Jarkko Piironen
- **File:Wikibooks-logo-en-noslogan.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/d/df/Wikibooks-logo-en-noslogan.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* User:Bastique, User:Ramac et al.
- **File:Wikibooks-logo.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/f/fa/Wikibooks-logo.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* User:Bastique, User:Ramac et al.
- **File:Wikiquote-logo.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/f/fa/Wikiquote-logo.svg> *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Wikisource-logo.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/4/4c/Wikisource-logo.svg> *License:* CC BY-SA 3.0 *Contributors:* Rei-artin *Original artist:* Nicholas Moreau
- **File:Wikiversity-logo-Snorky.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/1/1b/Wikiversity-logo-en.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Snorky
- **File:Wiktionary-logo-en.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/f/f8/Wiktionary-logo-en.svg> *License:* Public domain *Contributors:* Vector version of Image:Wiktionary-logo-en.png. *Original artist:* Vectorized by Fvasconcellos (talk · contribs), based on original logo tossed together by Brion Vibber
- **File:Z-curve.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/3/30/Z-curve.svg> *License:* Public domain *Contributors:* Own work *Original artist:* David Eppstein
- **File:Zigzag.gif** *Source:* <http://upload.wikimedia.org/wikipedia/commons/6/6f/Zigzag.gif> *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Zigzig.gif** *Source:* <http://upload.wikimedia.org/wikipedia/commons/f/fd/Zigzig.gif> *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Zipcodes-Germany-AngTanSplit.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/1/19/Zipcodes-Germany-AngTanSplit.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Chire
- **File:Zipcodes-Germany-GuttmanRTree.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/0/0e/Zipcodes-Germany-GuttmanRTree.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Chire
- **File:Zipcodes-Germany-RStarTree.svg** *Source:* <http://upload.wikimedia.org/wikipedia/commons/c/c7/Zipcodes-Germany-RStarTree.svg> *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Chire

## 14.3 Content license

- Creative Commons Attribution-Share Alike 3.0