

Introduction to Data Structures and Algorithms with C++ and Java

Olcay Taner Yıldız

January 5, 2025

Preface

This book is designed as an introductory text book for the Data Structures and Algorithms course. Data structures and algorithms is a standard course taught in many universities teaching computer engineering and/or computer science. In general, an introduction to programming course is taught as a prerequisite before the data structures course, and afterwards an algorithm course is taught to complete the problem solving objective.

The book is designed as a 40 hour course work. An example course schedule can be designed as: Link list, tree, graph and sorting algorithms require 6 hours; algorithm analysis, stack, queue, hashing, and disjoint set require 3 hours. The first chapter of the book gives an introductory info about algorithm analysis, afterwards the main data structures are explained in the coming 8 chapters (from chapter 2 to chapter 9). Although the last chapter, i.e. sorting algorithms, is not a data structure, it is included since sorting is taught in many universities as a part of this course.

In each data structure chapter, we give the data structure codes, then we introduce the main operations on that data structure. One can implement the same data structure in many ways, therefore more than one implementation and its basic operations of the same implementation is given in some chapters. For example, link list is implemented as single link list, double link list and circular link list, and the basic operations insertion and deletion are given for all these implementations. As a last step, we give one or more example applications of the data structure, which helps the student to understand the concept more thoroughly in everyday applications.

As always, I dedicate this book to my wife, Aysel; and to my children Aysu İpek and Oğuz Kerem; for their love, support, and encouragement.

Contents

1	Algorithm Analysis	1
1.1	Big-O Notation	1
1.2	Analysis of Non-recursive Algorithms	5
1.3	Analysis of Recursive Algorithms	8
1.4	Master Theorem	12
1.5	Notes	14
1.6	Solved Exercises	14
1.7	Exercises	17
1.8	Problems	23
2	Linked List	31
2.1	Definition	33
2.2	Singly Linked List Operations	35
2.2.1	Insertion	35
	Insert First	36
	Insert Last	37
	Insert Middle	38
2.2.2	Search	39
2.2.3	Getting i 'th Node	40
2.2.4	Deletion	40
	Delete First	41
	Delete Last	42

	Delete Middle	43
2.2.5	Number of Nodes in a Singly Linked List	45
2.3	Merging Two Singly Linked Lists	46
2.4	Doubly Linked List	47
2.5	Doubly Linked List Operations	50
2.5.1	Insertion	50
	Insert First	50
	Insert Last	52
	Insert Middle	54
2.5.2	Deletion	56
	Delete First	56
	Delete Last	57
	Delete Middle	58
2.6	Circular Linked List	60
2.6.1	Insertion	61
2.6.2	Deletion	63
2.7	Application: Polynomial Arithmetic	66
2.7.1	Adding Two Polynomials	67
2.7.2	Multiplying Two Polynomials	72
2.8	Solved Exercises	77
2.9	Exercises	83
2.10	Problems	85
3	Stack	93
3.1	Array Implementation	93
3.1.1	Insertion	94
3.1.2	Deletion	96
3.2	Linked List Implementation	98
3.2.1	Insertion	99
3.2.2	Deletion	100
3.3	Application: Evaluating Mathematical Expressions	102
3.3.1	Mathematical Expressions	103
3.3.2	Postfix Representation	105
3.3.3	Evaluation an Expression Given Its Postfix Form	105
3.3.4	Conversion from Infix to Postfix	110
3.4	Solved Exercises	114
3.5	Exercises	117
3.6	Problems	118

4	Queue	123
4.1	Array Implementation	124
4.1.1	Insertion	125
4.1.2	Deletion	126
4.2	Linked List Implementation	128
4.2.1	Insertion	128
4.2.2	Deletion	130
4.3	Application: Darts	131
4.3.1	Breadth First Search	133
4.3.2	Solving Darts Problem Using Breadth First Search . .	134
4.4	Solved Exercises	136
4.5	Exercises	139
4.6	Problems	140
5	Binary Search Trees	145
5.1	Definition	146
5.2	Binary Search Tree Operations	150
5.2.1	Search	150
5.2.2	Minimum and Maximum Elements	154
5.2.3	Insertion	156
5.2.4	Deletion	157
5.3	Traversals	161
5.4	Non-recursive Traversal	163
5.5	AVL Trees	166
5.5.1	Definition	166
5.5.2	Rotations	170
	Single Rotation	170
	Double Rotation	174
5.5.3	Insertion	177
5.6	B+ Tree	178
5.6.1	Definition	180
5.6.2	Search	181
5.6.3	Insertion	184
5.7	Application: Tree Index	190
5.7.1	Tree Index Structure	191
5.7.2	Filling the Binary Search Tree	191
5.7.3	Query	194
5.8	Notes	196

5.9	Solved Exercises	196
5.10	Exercises	201
5.11	Problems	203
6	Hashing	209
6.1	Hash Table	209
6.1.1	Definition	209
6.1.2	Hash Functions	210
6.2	Separate Chaining	212
6.2.1	Search	213
6.2.2	Insertion	214
6.2.3	Deletion	215
6.3	Open Addressing	216
6.3.1	Linear Probing	216
6.3.2	Quadratic Probing	220
6.3.3	Double Hashing	221
6.4	Rehashing	223
6.5	Application: Dart	225
6.6	Application: Hash Index	228
6.6.1	Hash Index Structure	228
6.6.2	Filling the Hash Table	228
6.6.3	Query	228
6.7	Notes	230
6.8	Solved Exercises	230
6.9	Exercises	234
6.10	Problems	236
7	Heap	241
7.1	Definition	243
7.2	Heap Operations	245
7.2.1	Deletion	245
7.2.2	Insertion	249
7.2.3	Search	251
7.2.4	Update	252
7.3	d -ary Heap	254
7.3.1	Deletion from d -ary Heap	257
7.3.2	Insertion into d -ary Heap	259
7.4	Application: Dart	264

7.5	Notes	268
7.6	Solved Exercises	268
7.7	Exercises	270
7.8	Problems	271
8	Disjoint Set	275
8.1	Definition	275
8.2	Disjoint Set Operations	278
8.2.1	Finding Set of An Element	278
8.2.2	Merging Two Disjoint Sets	279
8.3	Application: Cracking the Code	282
8.4	Solved Exercises	285
8.5	Exercises	288
8.6	Problems	289
9	Graph	293
9.1	Definition	295
9.1.1	Adjacency Matrix Representation	295
9.1.2	Adjacency List Representation	297
9.2	Add Edge	298
9.3	Application: Connected Components	301
9.3.1	Disjoint Set Approach	302
9.3.2	Depth First Search Approach	304
9.3.3	Breadth First Search Approach	308
9.4	Application: Shortest Path	312
9.4.1	Bellman-Ford Algorithm	312
9.4.2	Dijkstra Algorithm	314
9.4.3	Floyd-Warshall Algorithm	319
9.5	Application: Minimum Spanning Tree	322
9.5.1	Kruskal's Algorithm	324
9.5.2	Prim's Algorithm	330
9.6	Notes	334
9.7	Solved Exercises	335
9.8	Exercises	339
9.9	Problems	341

10 Sorting Algorithms	347
10.1 Insertion Sort	348
10.2 Selection Sort	350
10.3 Bubble Sort	351
10.4 Shell Sort	353
10.5 Heap Sort	356
10.6 Merge Sort	360
10.7 Quick Sort	363
10.8 Bucket Sort	367
10.9 Radix Sort	370
10.10Notes	371
10.11Solved Exercises	371
10.12Exercises	375
10.13Problems	376

List of Tables

1.1	The algorithm that finds the largest element of an array	2
2.1	Deleting the k 'th element from an array of size N	31
2.2	Inserting a new element at position k into an array of size N .	32
2.3	Searching number k in a sorted array of size N	33
2.4	Definition of Node	34
2.5	Definition of LinkedList	35
2.6	Inserting at the beginning of the singly linked list	36
2.7	Inserting at the end of the singly linked list	37
2.8	Inserting at the middle of a singly linked list	38
2.9	Searching a specific value in the linked list	39
2.10	Getting i 'th node of a linked list	40
2.11	Deleting from the beginning of the singly linked list	41
2.12	Deleting the last node from the singly linked list	42
2.13	Deleting from the middle of a singly linked list	44
2.14	Finding number of nodes in a linked list	45
2.15	Merging two linked lists	46
2.16	Definition of a node (For doubly linked list)	49
2.17	Definition of DoublyLinkedList	51
2.18	Inserting at the beginning of the doubly linked list	52
2.19	Inserting a new node at the end of the doubly linked list . . .	53

LIST OF TABLES

2.20	The algorithm that inserts a new node into the middle of the doubly linked list	54
2.21	Deleting from the beginning of the singly linked list	56
2.22	Deleting from the end of the doubly linked list	58
2.23	Deleting from the middle of a doubly linked list	59
2.24	Definition of circular linked list	62
2.25	Inserting at the beginning of the circular linked list	63
2.26	Deleting head node from the beginning of circular linked list .	64
2.27	Definition of a term of polynomial with single variable	67
2.28	The algorithm that finds the sum of two polynomials (C++) .	69
2.29	The algorithm that finds the sum of two polynomials (Java) .	70
2.30	The algorithm that finds the product of two polynomials (C++)	73
2.31	The algorithm that finds the product of two polynomials (Java)	74
3.1	The definition of an element in an array implemented stack . .	94
3.2	Definition of a stack, which contains integers and implemented using array	95
3.3	The algorithm that adds a new element on top of the stack . .	96
3.4	The algorithm that removes an element from the stack and returns it	97
3.5	Definition of a stack, which contains integers and implemented using linked list	99
3.6	The algorithm that adds a new element onto the stack	100
3.7	The algorithm that removes an element and returns it	101
3.8	Definition of a token in a mathematical expression	104
3.9	The algorithm that evaluates an expression given its postfix representation (C++)	107
3.10	The algorithm that evaluates an expression given its postfix representation (Java)	108
3.11	The algorithm that finds the postfix representation of an infix expression (C++)	111
3.12	The algorithm that finds the postfix representation of an infix expression (Java)	112
4.1	Definition of a queue, which contains integers and implemented using array	125
4.2	The algorithm that adds a new element to an array implemented queue	126

LIST OF TABLES

4.3	The algorithm that removes an element (and returns that element) from an array implemented queue	127
4.4	Definition of a queue, which contains integers and implemented using linked list	129
4.5	Adding a new element to the queue implemented using linked list	130
4.6	The algorithm that removes an element (and returns that element) from the queue implemented using linked list	131
4.7	Definition of a state in the Dart game problem	134
4.8	Solution of the Dart game problem using breadth first search .	135
5.1	Definition of node of a binary tree	150
5.2	Definition of binary search tree containing integers	150
5.3	The recursive algorithm that searches a given value in a binary search tree	151
5.4	The iterative algorithm that searches a given value in a binary search tree	152
5.5	Iterative algorithm that finds the minimum element in a binary search tree	154
5.6	Recursive algorithm that finds the minimum element in a binary search tree	154
5.7	Iterative algorithm that finds the maximum element in a binary search tree	155
5.8	Recursive algorithm that finds the maximum element in a binary search tree	155
5.9	The algorithm that adds a new node into the binary search tree	157
5.10	The algorithm that deletes a node from the binary search tree	159
5.11	Preorder traversal algorithm	161
5.12	Inorder traversal algorithm	162
5.13	Postorder traversal algorithm	162
5.14	The algorithm that finds the node count in a binary search tree (Stack)	164
5.15	The algorithm that finds the node count in a binary search tree (Queue)	165
5.16	Definition of an AVL tree node which contains integers	169
5.17	Definition of AVL tree	169
5.18	The single rotation algorithm to solve case 1	172
5.19	Single rotation algorithm to solve case 4	173

LIST OF TABLES

5.20	The double rotation algorithm that solves case 2	174
5.21	The double rotation algorithm that solves case 3	175
5.22	The algorithm that inserts a new node into an AVL tree (C++)	178
5.23	The algorithm that inserts a new node into an AVL tree (Java)	179
5.24	Definition of B+ tree node	182
5.25	Definition of B+ tree	183
5.26	The algorithm that searches a given value in B+ tree	183
5.27	The algorithm that adds a new node into the index part of B+ tree (C++)	186
5.28	The algorithm that adds a new node into the index part of B+ tree (Java)	187
5.29	The algorithm that adds a new value into the data part of B+ tree	188
5.30	The student class that contains the student information (no, name, surname).	192
5.31	Filling the binary search tree using the information in the student file	193
5.32	The non-recursive function that finds the name and surname of the student with student number 18	194
5.33	The functions that finds the number of students whose num- bers are larger than 23	195
6.1	Definition of the hash table	211
6.2	An example hash function which can be used for integers . . .	211
6.3	An example hash function which can be used for strings . . .	212
6.4	Definition of an hash table where each element is a linked list	214
6.5	Searching a value in a hash table where each element is a linked list	214
6.6	Inserting a new element into the hash table where each element is a linked list	215
6.7	Deleting an element from an hash table where each element is a linked list	215
6.8	Searching a number in an hash table (linear probing)	217
6.9	Inserting an element into the open addressing hash table with linear probing	218
6.10	Deleting an element from an open addressing hash table with linear probing	219
6.11	Rehashing an hash table	223

LIST OF TABLES

6.12 Solving the Dart game problem using breadth first search (2) (C++)	226
6.13 Solving the Dart game problem using breadth first search (2) (Java)	227
6.14 Filling the hash table using the information in the student file	229
6.15 The function that finds the name and surname of the student with student number 18 using hash table	229
7.1 Definition of elements stored in the heap	243
7.2 Definition of heap	244
7.3 Percolating down from a node of the heap to restore the max- heap property (C++)	247
7.4 Percolating down from a node of the heap to restore the max- heap property (Java)	248
7.5 Restoring max-heap property after deleting the first element of the heap	249
7.6 The algorithm that add a new element to the heap	249
7.7 Percolating up from a node of the heap to restore the max- heap property	251
7.8 Searching a specific element in the heap	251
7.9 Updating the value of an element in the heap	255
7.10 Definition of d -ary heap	257
7.11 Percolating down from a node of the d -ary heap to restore the max-heap property	258
7.12 Percolating up from a node of the d -ary heap to restore the max-heap property	262
7.13 Solving the Dart game problem using breadth first search (3) (C++)	266
7.14 Solving the Dart game problem using breadth first search (3) (Java)	267
8.1 The definition of a set	277
8.2 The definition of the disjoint set data structure which has a set as its own element	278
8.3 The function that returns the index of a set of an element with a given index	279
8.4 The function that merges two sets given their indexes	281
8.5 The application of crack the code (Java)	284

LIST OF TABLES

9.1	Definition of a graph using adjacency matrix representation . .	296
9.2	Definition of edge for adjacency list representation	298
9.3	Definition of edge list for adjacency list representation	299
9.4	Definition of graph using adjacency list representation	299
9.5	Adding an edge to a graph with adjacency matrix representation	300
9.6	Adding an edge to a graph with adjacency list representation .	300
9.7	Disjoint set algorithm that finds connected components in a graph	303
9.8	Depth first search algorithm that finds the connected components in a given graph (C++)	306
9.9	Depth first search algorithm that finds the connected components in a given graph (Java)	307
9.10	Breadth first search algorithm that finds the connected components in a given graph (C++)	309
9.11	Breadth first search algorithm that finds the connected components in a given graph (Java)	310
9.12	Bellman-Ford algorithm that finds the shortest paths from an initial vertex to all other vertices in a graph	313
9.13	Dijkstra algorithm that finds the shortest paths from an initial vertex to all other vertices in a graph (C++)	316
9.14	Dijkstra algorithm that finds the shortest paths from an initial vertex to all other vertices in a graph (Java)	317
9.15	Floyd-Warshall algorithm that finds the shortest paths from all vertices to all vertices in a given graph	320
9.16	The algorithm that returns all edges in a graph of adjacency list representation as a linked list (C++)	325
9.17	The algorithm that returns all edges in a graph of adjacency list representation as a linked list (Java)	326
9.18	Kruskal algorithm that finds the minimum spanning tree of a given graph	327
9.19	Prim's algorithm that finds the minimum spanning tree of a given graph (C++)	331
9.20	Prim's algorithm that finds the minimum spanning tree of a given graph (Java)	332
10.1	Insertion Sort Algorithm	348
10.2	Selection Sort Algorithm	350
10.3	Bubble sort algorithm	352

LIST OF TABLES

10.4 Shell sorting algorithm	354
10.5 Heap sort algorithm	356
10.6 Merge sort algorithm (C++)	361
10.7 Merge sort algorithm (Java)	362
10.8 Quicksort algorithm	364
10.9 Partitioning the array	365
10.10 Bucket sort algorithm	368

LIST OF TABLES

List of Figures

1.1	The growth of the 8 common functions in algorithm analysis (simple, logarithm, linear, log-linear, quadratic, cubic, exponential, square root)	4
1.2	Example cases for three representations	5
1.3	Division of a problem into subproblems in the master theorem	13
2.1	An example node containing integer data	33
2.2	An example linked list containing four items.	34
2.3	Inserting node 8 at the beginning of the linked list	36
2.4	Inserting node 8 at the end of the linked list	37
2.5	Inserting node 8 at the middle of the linked list	38
2.6	Deleting head node 9 from the linked list	41
2.7	Deleting last node 7 from the linked list	43
2.8	Deleting node 5 from the singly linked list	44
2.9	Merging linked lists	47
2.10	Node data structure which contains an integer (For doubly linked list)	49
2.11	An example doubly linked list and its corresponding array . .	50
2.12	Inserting at the beginning of the doubly linked list	52
2.13	Inserting at the end of the doubly linked list	53
2.14	Inserting in the middle of an example doubly linked list	55
2.15	Deleting head node from an example doubly linked list	57

LIST OF FIGURES

2.16	Deleting the last node from an example doubly linked list . . .	58
2.17	Deleting from the middle of a doubly linked list	60
2.18	An example circular linked list with corresponding array . . .	61
2.19	Inserting node 8 at the beginning of an example circular linked list	64
2.20	Deleting first node 9 from the beginning of an example circular linked list	65
2.21	Array representation of the polynomial $4x^5 + 3x^4 + 6x^2 - 8x - 7$	66
2.22	Linked list representation of the polynomial $8x^{74} - 3x^{42} -$ $12x^{23} + 4$	67
2.23	Adding polynomials $4x^5 + 3x^2 - 7x + 8$ and $2x^4 + 6x^2 + 7x$. .	71
2.24	Multiplication of polynomials $x^2 + 2x + 1$ and $x^2 - 2x$ (1. Step)	75
2.25	Multiplication of two polynomials $x^2 + 2x + 1$ and $x^2 - 2x$ (2. Step)	76
3.1	The array representation of a stack of size 8, which contains 6 elements	96
3.2	Adding an element on top of the stack	97
3.3	Removing an element from the stack	98
3.4	The linked list representation of a stack, which contains 6 elements	100
3.5	Adding an element onto the stack	101
3.6	Removing an element from the stack	102
3.7	Evaluation of the postfix expression $ab/c - de * + a *$ using a stack	109
3.8	Finding the postfix representation of the infix expression $A +$ $B * (C + D)$ using a stack	113
4.1	The queue data structure that can hold at most 12 elements .	124
4.2	Adding elements 17 and 3 to the queue structure given in Figure 4.1	126
4.3	Removing element 15 from the queue structure given in Figure 4.1	127
4.4	The queue structure that contains 5 elements and implemented using linked list	128
4.5	Adding element 17 to the queue structure given in Figure 4.4 .	130
4.6	Removing element 15 from the queue structure given in Figure 4.4	131

LIST OF FIGURES

4.7	An example Dart board and getting 100 in five shots	132
4.8	Applying two stages of the breadth first search in the Dart game problem	133
5.1	An example tree consisting of 14 nodes	145
5.2	A binary search tree consisting of six nodes	147
5.3	The tree consisting of 6 nodes. This tree does not satisfy the binary search tree property. The node on the left side of node 3 is not smaller than 3, although it is required to be smaller. .	147
5.4	A balanced binary search tree of depth 4, which consists of 15 nodes	148
5.5	A binary search tree of depth 4, which consists of 6 nodes . . .	149
5.6	Searching a value in a binary search tree	153
5.7	Searching for the minimum element in a binary search tree . .	155
5.8	Searching for the maximum element in a binary search tree . .	156
5.9	Inserting 13 into an example binary search tree. The nodes that are traversed while inserting 13 are marked with red . . .	158
5.10	Deleting the root node of an example binary search tree	160
5.11	The visit order of the nodes of different traversal algorithm in binary search tree	163
5.12	Three different binary search trees which are constructed by inserting numbers 1 through 7 into the tree in different orders	167
5.13	Binary search trees. (a) is an AVL tree (b) is not AVL tree. .	168
5.14	Single rotation to solve case 1	171
5.15	Single rotation to solve case 4	172
5.16	An example single rotation to solve case 4	173
5.17	Double rotation that solves the case 2	175
5.18	The double rotation that solves case 3	176
5.19	An example double rotation that is applied to solve case 3 . .	176
5.20	An example B+ tree of degree $d = 2$ storing 15 values	180
5.21	An example B+ tree of degree $d = 1$ storing 13 values	181
5.22	Searching 30 in the B+ tree given in Figure 5.21	184
5.23	Inserting 22 to the tree shown in Figure 5.20	189
5.24	Adding 22 to the tree shown in Figure 5.21	189
5.25	The binary search tree filled with the information in the example student file	193
6.1	An example hash table which contains 5 elements	210

LIST OF FIGURES

6.2	An example hash table where each element is a linked list . . .	213
6.3	Open addressing hash table with linear probing	216
6.4	An open addressing hash table with quadratic probing	221
6.5	An open addressing hash table with double hashing	222
6.6	Rehashing of the original hash table in Figure 6.1	224
7.1	An example max-heap	242
7.2	An example min-heap	242
7.3	Array representation of an example heap	244
7.4	Restoring the max-heap property after removing the first element of the heap	246
7.5	Restructring the heap after adding a new element	250
7.6	Changing the value of an element from 14 to 18	253
7.7	Changing the value of an element from 14 to 3	254
7.8	An example 3-ary heap	255
7.9	An example 4-art heap	256
7.10	Restoring the max-heap property of the example 3-ary heap given above after removing its first element	260
7.11	Restoring the max-heap property of the example 4-ary heap given above after removing its first element	261
7.12	Restructuring the 3-ary heap after inserting a new element . .	263
7.13	Restructuring the 4-ary heap after inserting a new element . .	264
8.1	An example disjoint set data structure containing 8 elements .	276
8.2	The array representation of a disjoint set structure containing 8 elements and 3 sets	276
8.3	Merging two sets with indexes 2 and 4 in Figure 8.1 in two different ways	280
8.4	Merging two sets with indexes 1 and 2 in Figure 8.1 in two different ways	280
9.1	An example directed graph consisting of six vertices and seven edges	293
9.2	An example undirected graph consisting of five vertices and four edges	294
9.3	An example weighted graph consisting of five vertices and five edges	295

LIST OF FIGURES

9.4	Adjacency matrix representation of the graphs shown in Figures 9.1 and 9.2	295
9.5	Adjacency list representation of the graphs shown in Figures 9.1 and 9.2	297
9.6	An example seven country map	302
9.7	Finding the connected components using the disjoint set approach in the graph given in Figure 9.6	304
9.8	Finding the connected components in the graph in Figure 9.6 using the depth first search approach	305
9.9	Finding the connected components in the graph in Figure 9.6 using the breadth first search approach	311
9.10	Application of Bellman-Ford algorithm on a graph of 5 vertices	315
9.11	Application of Dijkstra algorithm on a graph of 5 vertices . . .	318
9.12	Application of 4 steps of Floyd-Warshall algorithm on a graph of 5 vertices	321
9.13	A weighted graph consisting of six vertices	323
9.14	Two trees spanning the graph in Figure 9.13	324
9.15	Two minimum spanning trees for the graph in Figure 9.13 . .	324
9.16	Application of Kruskal algorithm to the graph in Figure 9.13 .	328
9.17	The disjoint set structure used in applying Kruskal algorithm to the graph in Figure 9.13	329
9.18	Application of Prim's algorithm to the graph in Figure 9.13 .	333
9.19	Min-heap T in the application of Prim's algorithm in Figure 9.18	334
10.1	Insertion sort of an array of six elements	349
10.2	Selection sort of an array of six elements	351
10.3	Bubble sort of an array of six elements	353
10.4	Shell sort of an array of six elements	355
10.5	Sorting an example array using heap sort (1)	358
10.6	Sorting an example array using heap sort (2)	359
10.7	Merging two sorted arrays of 4 elements	363
10.8	Mergesort of an example array of 8 elements	364
10.9	An example application of the function partition	366
10.10	Bucket sort of an eight element array <i>A</i>	369
10.11	Radix sort on an array of 7 elements	370

LIST OF FIGURES

Algorithm Analysis

We will study the most known data structures in this book. Although the definition changes from one programming language to another, the main idea behind a data structure remains the same. Data structures are programming constructs to process information as fast as possible. For example, linked list can bring solution to the the shift problem, which occurs when we insert an element to an array or remove an element from an array.

In this chapter, we will study how we can analyze the time complexity of a program segment (or an algorithm). First we will give the definitions of \mathcal{O} , Ω , and Θ functions, which are used to represent the complexity of a program, then we will give the analysis of recursive and non-recursive algorithms in two separate sections.

1.1 Big-O Notation

In order to analyze the time complexity of a program, one can determine the number of steps required to finish the program. For example, the code segment in Table 1.1 will execute

- 1 time `largest = array[0]` assignment statement
- 1 time `i = 1` assignment statement
- N times `i < N` comparison statement

1.1. BIG-O NOTATION

Table 1.1: The algorithm that finds the largest element of an array

```
1 int arrayLargest(int[] array){  
2   int i, largest;  
3   largest = array[0];  
4   for (i = 1; i < array.length; i++){  
5     if (array[i] > largest)  
6       largest = array[i];  
7   }  
8   return largest;  
9 }
```

- $N - 1$ times `i++` assignment statement
- $N - 1$ times `if (array[i] > largest)` comparison statement
- $N - 1$ times `largest = array[i]` assignment statement

for a sorted array in increasing order. If we assume that the time complexity of assignment statements and the comparison statements are the same, the total number of statements will be $4N - 1$. This assumption will save us from the details of (i) how the comparison and assignment statements are expressed in the machine language (ii) the speed of the processor that the program runs, and (iii) the memory complexity of the program and the speed of this memory etc. Using the attributes dependent on the computer that the program runs, will produce an undesired analysis result which is totally dependent on that computer.

Second simplification used in algorithm analysis is discarding the small degree terms while representing the time complexity of a program. For example, if a program executes $3N^3 + 5N + 2$ number of steps, while representing the time complexity of this program, one discards the small degree terms such as $5N$ and 2 , the coefficient 3 before the term N^3 is removed since the computers can be as 3 times fast in a short time. As a result, the time complexity of this program segment is said to be $\mathcal{O}(N^3)$. Similarly the time complexity of the program given in Table 1.1 is not $4N - 1$, but $\mathcal{O}(N)$. Here $\mathcal{O}(N)$, provides an upper bound for the time complexity of the program. If we express mathematically:

CHAPTER 1. ALGORITHM ANALYSIS

Definition 1. Let $f(n)$ and $g(n)$ be two functions defined from positive integers to real numbers. If there exists c and n_0 such that for each integer $n > n_0$, $f(n) < cg(n)$, then $f = \mathcal{O}(g)$ and f provides an upper bound for g .

We can also define a lower bound on the time complexity of the program. For example, the code segment in Table 1.1 will execute

- 1 time `largest = array[0]` assignment statement
- 1 time `i = 1` assignment statement
- N times `i < N` comparison statement
- $N - 1$ times `i++` assignment statement
- $N - 1$ times `if (array[i] > largest)` comparison statement

resulting in a total of $3N$ statements for a sorted array in decreasing order. So, $\Omega(N)$ provides a lower bound for the time complexity of the program. If we express mathematically again:

Definition 2. Let $f(n)$ and $g(n)$ be two functions defined from positive integers to real numbers. If there exists c and n_0 such that for each integer $n > n_0$, $f(n) > cg(n)$, then $f = \Omega(g)$ and f provides a lower bound for g .

If the lower and upper bounds of the time complexity of a program are the same, one can use Θ function to represent the time complexity of that program. For example, since the lower and upper bounds are the same for the code segment in Table 1.1, $\Omega(N)$ provides both a lower and an upper bound for the time complexity of that code segment.

Definition 3. Let $f(n)$ and $g(n)$ be two functions defined from positive integers to real numbers. If there exists c_1 , c_2 , and n_0 such that for each integer $n > n_0$, $c_1g(n) > f(n) > c_2g(n)$, then $f = \Theta(g)$ and f provides both a lower and an upper bound for g .

These definitions provide a relative comparison between functions. Given two functions f_1 and f_2 , in most cases there will be points for which one function is smaller than the other function. Therefore, a claim such as $f_1(n) < f_2(n)$ will not be truthful. On the other hand, the growth of these two functions can be compared relatively. Although, for small n , $f_1(n) = 10,000n$, will be larger than $f_2(n) = n^2$, $f_2(n)$ will grow faster

1.1. BIG-O NOTATION

and in the long run, (after $n = 10,000$) will pass $f_1(n)$. When we apply Definition 1, $f_1 = \mathcal{O}(f_2)$ and with $n_0 = 10,000$, $c = 1$. Similarly, if we apply Definition 2, $f_2 = \Omega(f_1)$ and again with $n_0 = 10,000$, $c = 1$.

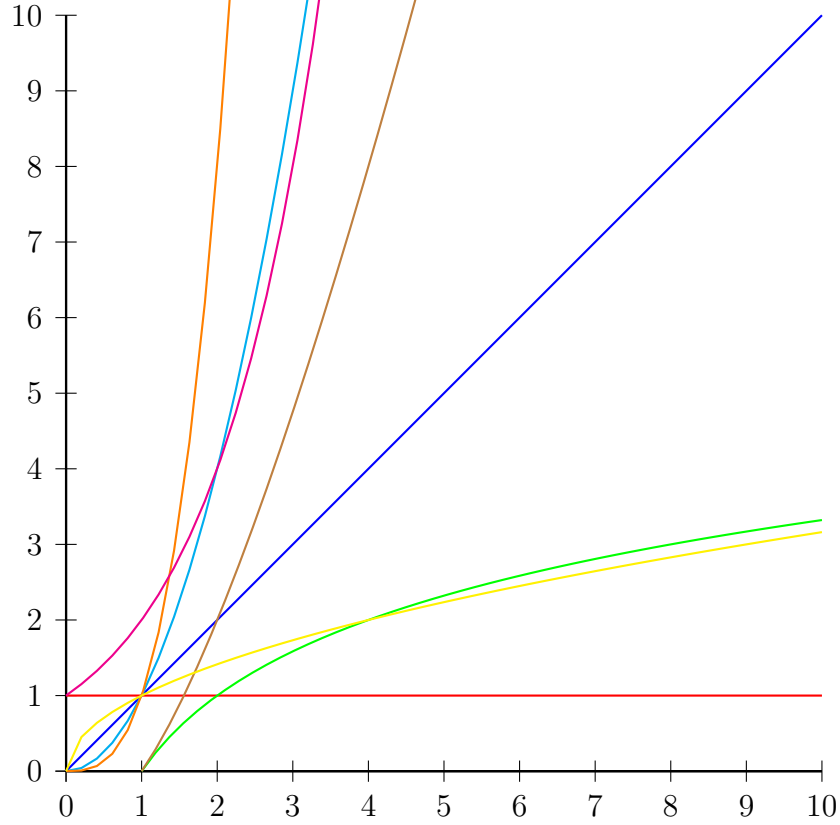


Figure 1.1: The growth of the 8 common functions in algorithm analysis (simple, logarithm, linear, log-linear, quadratic, cubic, exponential, square root)

Figure 1.1 shows the growth of the 8 common functions in algorithm analysis (simple, logarithm, linear, log-linear, quadratic, cubic, exponential, square root). As can be seen, these functions can be sorted as $1 < \log(n) < \sqrt{n} < n < n \log(n) < n^2 < n^3 < 2^n$ with respect to their order of growth.

Figure 1.2 shows example $f(n)$ and $g(n)$ functions for three representations and constants n_0 specified for these representations. In Figure 1.2(a),

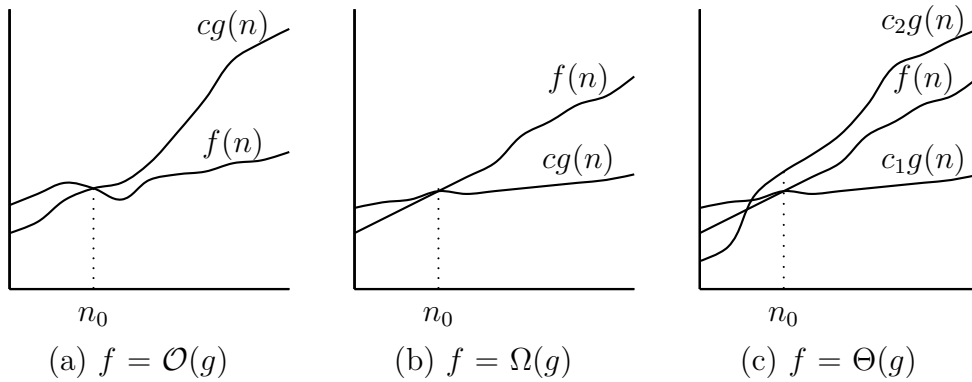


Figure 1.2: Example cases for three representations

f function takes larger values than $cg(n)$ until n_0 , then after n_0 , function g limits function f from above. In Figure 1.2(b), function f takes smaller values than $cg(n)$ until n_0 , then after n_0 , function g limits function f from below. As the last case, in Figure 1.2(c), function g limits function f both from below and above using coefficients c_1 and c_2 respectively.

1.2 Analysis of Non-recursive Algorithms

While doing the complexity analysis of a program (or function), the most important point is to check if the program is recursive or not. If the program (or function) to be analyzed calls itself, then this program (or function) is called a recursive program (or function), otherwise it is called a non-recursive program (or function). The simplifying points that can be used in a non-recursive program analysis is given as below.

The time complexity of a for loop is calculated by multiplying the time complexity of the code segment inside the for loop and the number of iterations of that for loop. For example, to calculate the time complexity of the function

```

1 int sumSquares(int N){
2   int i, sum = 0;
3   for (i = 1; i <= N; i++)
4     sum += i * i;
5   return sum;

```

1.2. ANALYSIS OF NON-RECURSIVE ALGORITHMS

6 }

one needs only calculate the time complexity of the code segment inside the for loop. Let $T(N)$ show the time complexity of the function, then

$$\begin{aligned} T(N) &= \sum_{i=1}^N 1 \\ &= N \in \mathcal{O}(N) \end{aligned}$$

If there are more than one loop and they are within each other (nested loops), then the time complexity of the function is the product of the time complexities of all loops. For example, the time complexity of the code segment

```
1 sum = 0;
2 for (i = 0; i < N; i++)
3     for (j = 0; j < N; j++)
4         sum++;
```

is

$$\begin{aligned} T(N) &= \sum_{i=0}^{N-1} \underbrace{\sum_{j=0}^{N-1} 1}_N \\ &= \sum_{i=0}^{N-1} N \\ &= N \underbrace{\sum_{i=0}^{N-1} 1}_N \\ &= N^2 \in \mathcal{O}(N^2) \end{aligned}$$

If the code segments are one after another, then the time complexity of the slowest code segment is accepted as the time complexity of the program. For example, the time complexity of the code segment

```
1 sum = 0;
2 for (i = 1; i <= N; i++)
3     sum += i * i;
```

CHAPTER 1. ALGORITHM ANALYSIS

```
4 for (i = 0; i < N; i++)
5     for (j = 0; j < N; j++)
6         sum++;
```

is $\mathcal{O}(N) + \mathcal{O}(N^2) \in \mathcal{O}(N^2)$.

The time complexity of the if/else condition statement is the maximum of the time complexities of code segments that are executed when the condition is true or not.

```
1 sum = 0;
2 if (N % 2 == 0)
3     for (i = 1; i <= N; i++)
4         sum += i * i;
5 else
6     for (i = 0; i < N; i++)
7         for (j = 0; j < N; j++)
8             sum++;
```

The time complexity of the program above is, the largest of the time complexity of the executed code segment when the condition `N % 2 == 0` is true $\mathcal{O}(N)$ and the time complexity of the executed code segment when that condition is false $\mathcal{O}(N^2)$, $\mathcal{O}(N^2)$.

When the loop variable does not increase/decrease one by one, identifying the values of the loop variable while executing the loop will help us to determine the time complexity of the loop. For example, in the function

```
1 int digitCount(int N){
2     int i, digits = 1;
3     while (N > 1){
4         digits++;
5         N = N / 2;
6     }
7     return digits ;
8 }
```

the loop variable N is divided by 2 in each iteration of the loop. After first iteration its value will be $\frac{N}{2}$, after second iteration its value will be $\frac{N}{4} = \frac{N}{2^2}$, after third iteration its value will be $\frac{N}{8} = \frac{N}{2^3}$, ..., and after k . iteration its value will be $\frac{N}{2^k}$. When we exit the loop, the value of N will be 1, and the

1.3. ANALYSIS OF RECURSIVE ALGORITHMS

number of iterations of the loop will be k . Then

$$\begin{aligned}\frac{N}{2^k} &= 1 \\ N &= 2^k \\ k &= \log_2 N\end{aligned}$$

the number of iterations will be $\log_2 N$, and the time complexity of the function will be $\mathcal{O}(\log N)$.

1.3 Analysis of Recursive Algorithms

The analysis of recursive algorithms can not be done as the analysis of the non-recursive algorithms. The main reason is, the time complexity of a recursive function does not depend only the statements in the function declaration but also depends on a function of the time complexity of the function itself. For example, the time complexity of a well known factorial function for input N

```
1 int factorial (int N){  
2   if (N <= 1)  
3     return 1;  
4   else  
5     return N * factorial (N - 1);  
6 }
```

depends again on the time complexity of the factorial function for input $N - 1$. If we express mathematically, let $f(N)$ show the time complexity of factorial function, then

$$\begin{aligned}f(1) &= 1 \\ f(N) &= f(N - 1) + 1\end{aligned}$$

Here the first equation shows the number of statements the function executes when $N = 1$, in other words the return operation in line `return 1;`, $+1$ in the second equation shows the multiplication operation in line `return N * factorial (N - 1);`. In order to determine the time complexity of the factorial function, we need to solve this recurrence equation system. By assigning $N - 1, N - 2, \dots, 2$

CHAPTER 1. ALGORITHM ANALYSIS

to N in the second equation, we get

$$\begin{aligned}f(N) &= f(N-1) + 1 \\f(N-1) &= f(N-2) + 1 \\f(N-2) &= f(N-3) + 1 \\&\dots \\f(2) &= f(1) + 1\end{aligned}$$

When we sum up the left and right side of the equalities, $f(N-1)$, $f(N-2)$, \dots , $f(2)$ cancel and

$$\begin{aligned}f(N) &= f(1) + N - 1 \\f(N) &= N \in \mathcal{O}(N)\end{aligned}$$

remains.

Our second example function is the recursive version of the function `digitCount` we have written in Section 1.2.

```
1 int digitCount(int N){
2   if (N == 1)
3     return 1;
4   else
5     return 1 + digitCount(N / 2);
6 }
```

The time complexity of this function (for input N) depends again on the time complexity of the function itself (for input $N / 2$). If we express mathematically, let $b(N)$ show the time complexity of function `digitCount`, then

$$\begin{aligned}b(1) &= 1 \\b(N) &= b(N/2) + 1\end{aligned}$$

Here the first equation shows the number of statements the function executes when $N = 1$, namely the return operation in line `return 1;`, +1 in the second equation shows the addition operation in line `return 1 + digitCount(N / 2);`. In order to determine the time complexity of function `digitCount` we need to solve this recurrence equation system. Like the previous equation system, assigning $N - 1$, $N - 2$, \dots , 2 to N will not work in this equation system. Instead, by assuming $N = 2^k$ we assign $N/2 = 2^{k-1}$, $N/2^2 = 2^{k-2}$, \dots , $N/2^{k-1} = 2$ to N , we get

1.3. ANALYSIS OF RECURSIVE ALGORITHMS

$$\begin{aligned}b(2^k) &= b(2^{k-1}) + 1 \\b(2^{k-1}) &= b(2^{k-2}) + 1 \\b(2^{k-2}) &= b(2^{k-3}) + 1 \\&\dots \\b(2^1) &= b(1) + 1\end{aligned}$$

When we sum up the left and right side of the equalities, $b(2^{k-1})$, $b(2^{k-2})$, \dots , $b(2)$ cancel and

$$\begin{aligned}b(2^k) &= b(1) + k \\b(2^k) &= k + 1 \\b(N) &= \log_2 N + 1 \in \mathcal{O}(\log N)\end{aligned}$$

remains.

We can also solve the same equation system using the master theorem in section 1.4. When we compare two equations

$$\begin{aligned}b(N) &= b(N/2) + 1 \\T(N) &= aT(N/b) + \mathcal{O}(N^d)\end{aligned}$$

we will see $a = 1$, $b = 2$, $d = 0$. Since $\log_b a = \log_2 1 = 0 = d$, the solution of the equation system is $b(N) = \mathcal{O}(N^d \log N) = \mathcal{O}(\log N)$.

As a last example, let's do the time complexity analysis of the following function.

```
1 void hanoi(int N, int tower1, int tower2){
2     int tower3 = 6 - tower1 - tower2;
3     if (N == 1)
4         movePeg(tower1, tower2);
5     else{
6         hanoi(N - 1, tower1, tower3);
7         movePeg(tower1, tower2);
8         hanoi(N - 1, tower3, tower2);
9     }
10 }
```

The time complexity of hanoi function depends again (for input N) the time complexity of the function itself (for input $N - 1$). If we express mathemat-

CHAPTER 1. ALGORITHM ANALYSIS

ically, let $h(N)$ show the time complexity of hanoi function, then

$$\begin{aligned}h(1) &= 1 \\h(N) &= 2h(N-1) + 1\end{aligned}$$

Here the first equation shows the number of statements the function executes when $N = 1$, namely the operation in line `movePeg(tower1, tower3);`; $2h(N-1)$ in the second equation shows the line `hanoi(N-1, ...)` where two recursive calls were made; $+1$ shows the operation in line `movePeg(tower1, tower3);`. In order to determine the time complexity of function hanoi we need to solve this recurrence equation system. By assigning $N-1, N-2, \dots, 2$ to N in the second equation, we get

$$\begin{aligned}h(N) &= 2h(N-1) + 1 \\h(N-1) &= 2h(N-2) + 1 \\h(N-2) &= 2h(N-3) + 1 \\&\dots \\h(2) &= 2h(1) + 1\end{aligned}$$

Here adding left and right side of the equations does not work. First we need to equalize the coefficients of h functions on the left side with the coefficients of h functions on the right side with the same parameters. For example, the coefficient of $h(N-1)$ on the left side of the equation must be the same as the coefficient of $h(N-1)$ on the right side of the equation. In order to do that, we need to multiply second equation with 2, third equation with 4 = $2^2, \dots$. After these multiplications, the recurrence system will look like

$$\begin{aligned}h(N) &= 2h(N-1) + 1 \\2h(N-1) &= 2^2h(N-2) + 2 \\2^2h(N-2) &= 2^3h(N-3) + 2^2 \\&\dots \\2^{N-1}h(2) &= 2^N h(1) + 2^{N-1}\end{aligned}$$

When we sum up the left and right side of the equalities, $h(N-1), h(N-2),$

..., $h(2)$ cancel and

$$\begin{aligned} h(N) &= 2^N f(1) + 2^{N-1} + 2^{N-2} + \dots + 1 \\ h(N) &= \sum_{i=0}^N 2^i \\ h(N) &= 2^{N+1} - 1 \in \mathcal{O}(2^N) \end{aligned}$$

remains.

1.4 Master Theorem

The equation systems we encountered frequently in the analysis of recursive functions are called recurrence equations in the literature. We encounter these type of equations especially when we try to solve divide-and-conquer type of problems. For example, in order to solve a problem of size N recursively, let say we first solve a subproblems of size N/b and combine the solutions of these subproblems in $\mathcal{O}(N^d)$ time. Then the total time complexity to solve the problem will be defined by the equation

$$T(N) = aT(N/b) + \mathcal{O}(N^d)$$

Theorem 1 (Master Theorem). *Let a , b , and d be real numbers satisfying conditions $a > 0$, $b > 1$, $d \geq 0$, the solution of the equation*

$$T(N) = aT(N/b) + \mathcal{O}(N^d)$$

is given as

$$T(N) = \begin{cases} \mathcal{O}(N^d) & \text{if } d > \log_b a \\ \mathcal{O}(N^d \log N) & \text{if } d = \log_b a \\ \mathcal{O}(N^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

In order to prove this theorem, we will assume that N is a power of b .

As can be seen in Figure 1.3, the problem of size N is first divided into subproblems of size $\frac{N}{b}$, then into subproblems of size $\frac{N}{b^2}$, then into subproblems of size $\frac{N}{b^3}$, ..., after $\log_b N$ levels into subproblems of size 1. We also see in Figure 1.3 that, in the first level a subproblems, in the second level

CHAPTER 1. ALGORITHM ANALYSIS

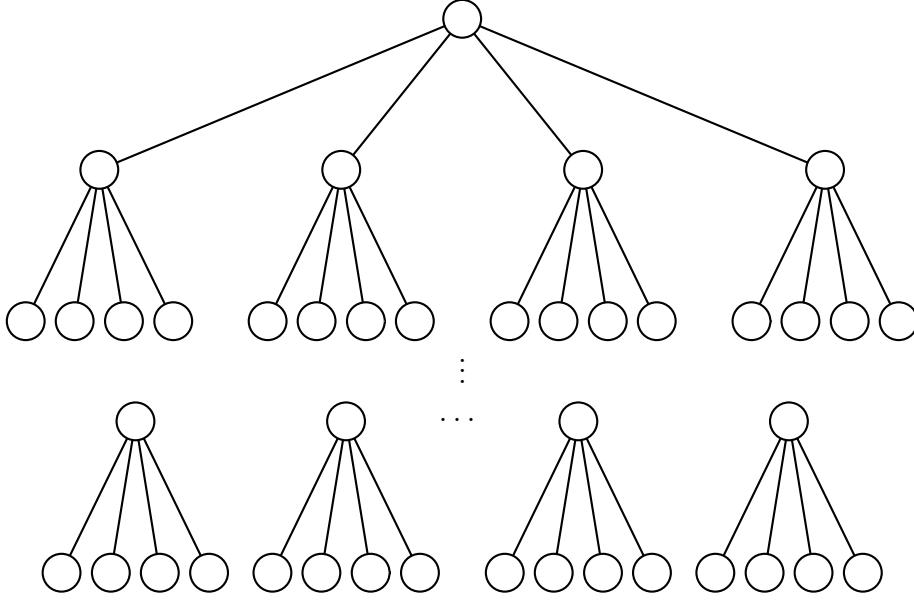


Figure 1.3: Division of a problem into subproblems in the master theorem

a^2 subproblems, \dots , and lastly in the k 'th level a^k subproblems need to be solved.

Since in the k 'th level a^k problems of size $\frac{N}{b^k}$ need to be solved, the total number of operations in level k will be

$$a^k \times \mathcal{O}\left(\frac{N}{b^k}\right)^d = \mathcal{O}(N^d) \times \left(\frac{a}{b^d}\right)^k$$

Then total number of operations in all levels will be (time complexity of the program)

$$\begin{aligned} T(N) &= \sum_{k=0}^{\log_b N} \mathcal{O}(N^d) \times \left(\frac{a}{b^d}\right)^k \\ &= \mathcal{O}(N^d) \sum_{k=0}^{\log_b N} \left(\frac{a}{b^d}\right)^k \end{aligned}$$

According to the ratio of $\frac{a}{b^d}$

- If $\frac{a}{b^d} < 1$, the series will converge, and $\sum_{k=0}^{\log_b N} \left(\frac{a}{b^d}\right)^k$ will be constant. The sum of series will be $\mathcal{O}(N^d)$.

- If $\frac{a}{b^d} = 1$, all elements of the series will be the same and 1. The sum of series will be $\mathcal{O}(N^d \log N)$.
- If $\frac{a}{b^d} > 1$, the series will diverge, and only the last element will be important in the sum. The sum of series will be $\mathcal{O}\left(\frac{a}{b^d}\right)^{\log_b N} = \mathcal{O}(N^{\log_b a})$.

For example, in the binary search algorithm, in order to find a specific number in an array of size N , depending on the middle number and the number searched, we search left or right subarrays of size $N / 2$. Therefore, since in each case we are doing search in one subarray $a = 1$, since in each case we search only half of the array $b = 2$ and since in each case we only compare the middle number with the number searched $d = 0$. Since $d = \log_b a$ second case is active and the time complexity of binary search will be $\mathcal{O}(N^d \log N) = \mathcal{O}(\log N)$.

1.5 Notes

The well known book in algorithm analysis is the Knuth's 3 volume "The Art of Computer Programming" series [10], [11], [12]. Big-O, big-omega and big-theta representations are also proposed by Knuth [9].

1.6 Solved Exercises

1. What is the time complexity of the following code fragment?

```

1 sum = 0;
2 for (i = 0; i < N; i++)
3     for (j = 0; j < i; j++)
4         sum++;

```

In this non-recursive code segment, the most repeated statement is the statement in line four, namely `sum++`. The number of executions of

CHAPTER 1. ALGORITHM ANALYSIS

this statement

$$\begin{aligned} T(N) &= \sum_{i=0}^{N-1} \underbrace{\sum_{j=0}^{i-1} 1}_i \\ &= \sum_{i=0}^{N-1} i \\ &= \frac{(N-1)N}{2} \\ &= 0.5N^2 - 0.5N \in \mathcal{O}(N^2) \end{aligned}$$

will give us the time complexity of this code segment.

2. What is the time complexity of the following code fragment?

```
1 sum = 0;
2 for (i = 0; i < N; i++)
3     for (j = 1; j < N; j *= 2)
4         sum++;
```

In this non-recursive code segment, the inner and outer loops are independent of each other. The time complexity of the outer loop is

$$\begin{aligned} T(N) &= \sum_{i=0}^{N-1} 1 \\ &= N \in \mathcal{O}(N) \end{aligned}$$

whereas the time complexity of the inner loop is $\mathcal{O}(\log N)$.

If there are nested loops and if these loops are independent from each other then the time complexity of the code segment will be the product of the time complexities of all loops. Therefore the time complexity of this program will be $\mathcal{O}(N) \times \mathcal{O}(\log N) = \mathcal{O}(N \log N)$.

3. What is the time complexity of the following recursive function?

```
1 int multiplyAll (int [] A, int n){
2     if (n == 0)
3         return A[0];
4     else
5         return A[n] * multiplyAll (A, n-1);
6 }
```

1.6. SOLVED EXERCISES

In this recursive function, for input N the time complexity is again dependent on the time complexity of `multiplyAll` function for input $N - 1$. If we express mathematically, let $h(N)$ show the time complexity of function `multiplyAll`, then

$$\begin{aligned} h(0) &= 1 \\ h(N) &= h(N - 1) + 1 \end{aligned}$$

Here the first equation shows the number of statements the function executes when $N = 0$, namely the return statement in line `return A[0];`, +1 in the second equation shows the multiplication operation in line `return A[n] * multiplyAll(A, n - 1);`. In order to determine the time complexity of function `multiplyAll` we need to solve this recurrence equation system. By assigning $N - 1, N - 2, \dots, 1$ to N in the second equation, we get

$$\begin{aligned} h(N) &= h(N - 1) + 1 \\ h(N - 1) &= h(N - 2) + 1 \\ h(N - 2) &= h(N - 3) + 1 \\ &\dots \\ h(1) &= h(0) + 1 \end{aligned}$$

When we sum up the left and right side of the equalities, $h(N - 1), h(N - 2), \dots, h(1)$ cancel and

$$\begin{aligned} h(N) &= h(0) + N \\ h(N) &= N + 1 \in \mathcal{O}(N) \end{aligned}$$

remains.

4. There are three algorithms for the same problem: Algorithm A, divides the original problem of size N into four subproblems of half the size, recursively solves each subproblem and combines solutions in linear time. Algorithm B divides the original problem into two subproblems of size $N/4$, recursively solves each subproblem and combines solutions again in linear time. Last algorithm C, divides the original problem into nine subproblems of size $N / 9$, recursively solves each subproblem and combines the solutions in $\mathcal{O}(N^2)$ time. Which algorithm is the fastest one?

CHAPTER 1. ALGORITHM ANALYSIS

If we express the time complexities of three algorithms as recurrence equations, we get

$$\begin{aligned}A(N) &= 4A(N/2) + \mathcal{O}(N) \\B(N) &= 2B(N/4) + \mathcal{O}(N) \\C(N) &= 9C(N/3) + \mathcal{O}(N^2)\end{aligned}$$

If we try to solve these equations with master theorem, for algorithm A we see that

$$\begin{aligned}A(N) &= 5A(N/2) + \mathcal{O}(N) \\lstlisting T(N) &= aT(N/b) + \mathcal{O}(N^d)\end{aligned}$$

$a = 5, b = 2, d = 1$. Since $\log_b a = \log_2 5 = 2.32 > d$, time complexity of the algorithm A will be $A(N) = \mathcal{O}(N^{\log_b a}) = \mathcal{O}(N^{2.32})$.

For algorithm B, we see that

$$\begin{aligned}B(N) &= 2B(N/4) + \mathcal{O}(N) \\T(N) &= aT(N/b) + \mathcal{O}(N^d)\end{aligned}$$

$a = 2, b = 4, d = 1$. Since $\log_b a = \log_4 2 = 0.5 < d$, time complexity of the algorithm B will be $B(N) = \mathcal{O}(N^d) = \mathcal{O}(N)$.

For algorithm C, we see that

$$\begin{aligned}C(N) &= 9C(N/3) + \mathcal{O}(N^2) \\T(N) &= aT(N/b) + \mathcal{O}(N^d)\end{aligned}$$

$a = 9, b = 3, d = 2$. Since $\log_b a = \log_3 9 = 2 = d$, time complexity of the algorithm C will be $C(N) = \mathcal{O}(N^d \log N) = \mathcal{O}(N^2 \log N)$ and the fastest algorithm will be B.

1.7 Exercises

1. What is the time complexity of the following code fragment?

```
1 sum = 0;
2 for (i = 0; i < N; i++)
3     for (j = 0; j < N * N; j++)
4         sum++;
```

1.7. EXERCISES

2. What is the time complexity of the following code fragment?

```
1 sum = 0;
2 for (i = 1; i < N; i *= 2)
3     sum++;
```

3. What is the time complexity of the following code fragment?

```
1 sum = 0;
2 for (i = 0; i < N; i++)
3     for (j = 0; j < i * i; j++)
4         if (j % i == 0)
5             sum++;
```

4. What is the time complexity of the following code fragment?

```
1 sum = 0;
2 for (i = 0; i < N; i++)
3     k = Math.pow(2, N);
4     for (j = 1; j < k; j *= 2)
5         sum++;
```

5. What is the time complexity of the following code fragment?

```
1 sum = 0;
2 for (i = 0; i < N; i++)
3     for (j = 0; j < i; j++)
4         for (k = 0; k < j; k++)
5             sum++;
```

6. What is the time complexity of the following code fragment?

```
1 sum = 0;
2 for (i = 0; i < N; i++)
3     for (j = 0; j < i * i; j++)
4         if (j % i == 0)
5             for (k = 0; k < j; k++)
6                 sum++;
```

7. What is the time complexity of the following code fragment?

CHAPTER 1. ALGORITHM ANALYSIS

```
1 sum = 0;
2 for (i = 0; i < N; i++)
3     for (j = i; j < N; j++)
4         sum++;
5 for (i = 0; i < N; i++)
6     for (j = 1; j < N; j*=2)
7         sum++;
```

8. What is the time complexity of the following function?

```
1 int magic(int N){
2     sum = 0;
3     for (i = 0; i < N; i++)
4         sum++;
5     for (i = 1; i < N; i++)
6         for (j = 1; j < N; j++)
7             if (i % j == 0)
8                 sum++;
9     for (i = 0; i < N; i++)
10        for (j = 1; j < N; j *= 2)
11            sum++;
12    return sum;
13 }
```

9. What does the function `algorithm1` return in terms of N ?

```
1 int algorithm1(int N){
2     int i, j, k, sum = 0;
3     for (i = 1; i <= N; i++)
4         for (j = i; j <= N; j++)
5             for (k = 1; k <= j - i; k++)
6                 sum++;
7     return sum;
8 }
```

10. What is the time complexity of the following recursive function?

```
1 void f(int n){
2     if (n > 1){
3         System.out.println(" Still -going-on ...");
4         f(n / 2);
5         f(n / 2);
6     }
}
```

7 }
 8 }

11. What is the time complexity of the following recursive function?

```

1  int fastExponentiation(int x, int n){
2      if (n == 1)
3          return x;
4      if (even(n))
5          return fastExponentiation(x * x, n / 2);
6      else
7          return fastExponentiation(x * x, n / 2) * x;
8  }
```

12. What is the time complexity of the following recursive function?

```

1  int magic(int N){
2      sum = 0;
3      for (i = 0; i < N; i++)
4          sum++;
5      return magic(N - 1) + sum;
6  }
```

13. What is the time complexity of the following recursive function?

```

1  int magic2(int N){
2      sum = 0;
3      for (i = 0; i < N; i++)
4          sum++;
5      return magic2(N / 2) + sum;
6  }
```

14. What does the function algorithm2 return in terms of N?

```

1  int algorithm2(int N){
2      if (N == 0)
3          return 0;
4      sum = 0;
5      for (i = 0; i < N; i++)
6          sum++;
7      return algorithm2(N - 1) + sum;
8  }
```

15. What is the time complexity of the following recursive function?

CHAPTER 1. ALGORITHM ANALYSIS

```
1  int algorithmB(int n){  
2      int number1, number2, number3;  
3      if (n > 0){  
4          number1 = algorithmB (n - 1);  
5          number2 = algorithmB (n - 1);  
6          number3 = algorithmB (n - 1);  
7          return number1 + number2 + number3;  
8      } else  
9          return 0;  
10 }
```

16. Given the following function

```
1  int magic(int N){  
2      if (N == 0)  
3          return 0;  
4      int sum = 0;  
5      for (i = 0; i < N; i++)  
6          sum++;  
7      return algorithm(N / 2) + sum;  
8  }
```

What is the time complexity of magic?

17. Given the following function

```
1  int algorithm1(int N, int k){  
2      if (N == 0)  
3          return 0;  
4      int sum = 0;  
5      for (int i = 0; i < N; i++)  
6          sum++;  
7      for (int i = 0; i < N; i++)  
8          for (int j = i; j < N; j++)  
9              sum++;  
10     for (int i = 0; i < 4; i++)  
11         sum += algorithm(N / 2);  
12     return sum;  
13 }
```

What is the time complexity of algorithm1?

18. Given the following function

```

1  int algorithm2(int N){
2      int sum = 0;
3      for (int i = 1; i <= N; i++){
4          for (int j = i; j <= N; j++){
5              sum++;
6          }
7      }
8  }

```

What is the value of sum after the execution of the algorithm2?

19. Given the following function

```

1  int algorithm(int N){
2      int sum = 0;
3      if (N == 0)
4          return 0;
5      if (N % 2 == 0){
6          for (i = 0; i < N; i++)
7              for (int j = 0; j < N; j++)
8                  sum++;
9      } else {
10         for (int j = 0; j < N; j++)
11             sum++;
12     }
13     return algorithm(N - 1) + sum;
14 }

```

What is the time complexity of algorithm? Extract the recurrence equation for the time complexity. Solve the recurrence equation.

20. Given the following function

```

1  int magic(int N){
2      if (N == 0)
3          return 0;
4      int sum = 0;
5      for (i = 0; i < N; i++)
6          sum++;
7      return magic(N / 2) + sum;
8  }

```

What is the time complexity of magic? Extract the recurrence equation for the time complexity. Solve the recurrence equation without the master theorem.

1.8 Problems

1. Given the following function

```

1  int algorithm1(int N, int k){
2      if (N == 0)
3          return 0;
4      int sum = 0;
5      for (int i = 0; i < N; i++)
6          sum++;
7      for (int i = 0; i < N; i++)
8          for (int j = i; j < N; j++)
9              sum++;
10     for (int i = 0; i < 4; i++)
11         sum += algorithm1(N / 2);
12     return sum;
13 }
```

What is the time complexity of algorithm1? Explain your result. Use the master theorem.

2. Given the following function

```

1  int algorithm1(int N){
2      int i, j, k, sum = 0;
3      for (i = 1; i <= N; i++)
4          for (j = i; j <= N; j++)
5              for (k = 1; k <= j - i; k++)
6                  sum++;
7      return sum;
8  }
```

What does the function algorithm1 return in terms of N?

3. Given the following function

```

1  int algorithm1(int N){
2      if (N == 0)
```

```

3     return 0;
4     int sum = 0;
5     for (int i = 0; i < N; i++)
6         sum++;
7     for (int i = 0; i < N; i++)
8         for (int j = 0; j < N; j++)
9             sum++;
10    for (int i = 0; i < 9; i++)
11        sum += algorithm1(N / 3);
12    return sum;
13 }

```

What is the time complexity of algorithm1? Explain your result. Use the master theorem.

4. Given the following function

```

1  int algorithm1(int N, int k){
2      if (N == 0)
3          return 0;
4      int sum = 0;
5      for (int i = 0; i < N; i++)
6          sum++;
7      for (int i = 0; i < N; i++)
8          for (int j = 0; j < sqrt(N); j++)
9              sum++;
10     if (sum % 2 == 0)
11         sum += algorithm1(N / 2);
12     else
13         sum -= algorithm1(N / 2);
14     return sum;
15 }

```

What is the time complexity of algorithm1? Explain your result. Use the master theorem.

5. Given the following function

```

1  int algorithm2(int N){
2      int i, j, k, sum = 0;
3      for (i = 1; i <= N; i++)
4          sum++;
5      for (i = 1; i <= N; i++)
6          for (j = N; j >= i; j--)
7              sum++;

```


CHAPTER 1. ALGORITHM ANALYSIS

```
8   for (i = 1; i <= N; i++)
9       for (j = 1; j <= N; j++)
10          for (k = j; k <= N - j; k++)
11              sum++;
12   return sum;
13 }
```

What does the function `algorithm2` return in terms of N ?

6. What does the function `algorithm1` return in terms of N ?

```
1  int algorithm1(int N){
2      int i, j, k, sum = 0;
3      for (i = 1; i <= N; i++)
4          for (j = i; j <= N - i; j++)
5              for (k = 1; k <= j - i; k++)
6                  sum++;
7      return sum;
8  }
```

Assume that N is even.

```
7.
1  int algorithm1(int N){
2      if (N == 0)
3          return 0;
4      int sum = 0;
5      for (int i = 0; i < N; i++)
6          for (int j = 0; j < N; j++)
7              sum++;
8      for (int i = N; i > 0; i--)
9          for (int j = 0; j < N; j++)
10             for (int k = 0; k < N; k++)
11                 sum++;
12     for (int i = 0; i < 4; i++)
13         for (int j = 0; j < 2; j++)
14             sum += algorithm1(N / 2);
15     return sum;
16 }
```

What is the time complexity of `algorithm1`? You **must** construct the recurrence equation and solve it with the master theorem.

```
8.
1  int algorithm1(int N){
2      if (N == 0)
```

```

3      return 0;
4      int sum = 0;
5      for (int i = N; i > 0; i--)
6          for (int j = 0; j < N; j++)
7              sum++;
8      for (int i = 0; i < 4; i++)
9          for (int j = 0; j < 4; j++)
10             sum += algorithm1(N / 4);
11      return sum;
12  }

```

What is the time complexity of algorithm1? Solve the recurrence equation without the master theorem.

9.

```

1  int algorithm2(int N){
2      int i, j, k, sum = 0;
3      for (i = 1; i <= N; i++)
4          sum++;
5      for (i = 1; i <= N; i++)
6          for (j = 1; j <= 2 * i; j++)
7              for (k = 1; k <= N; k++)
8                  sum++;
9      for (i = 0; i <= N; i++)
10         for (j = i + 1; j <= N - i; j++)
11             sum++;
12      return sum;
13  }

```

What does the function algorithm2 return in terms of N ? Assume that N is even.

10.

```

1  int algorithm1(int N){
2      if (N == 0)
3          return 0;
4      int sum = 0;
5      for (int i = 0; i < N; i++)
6          for (int j = 0; j < N; j++)
7              for (int k = 0; k < N; k++)
8                  sum++;
9      for (int i = 0; i < 4; i++)
10         for (int j = 0; j < 4; j++)
11             if ((i + j) % 2 == 0)
12                 sum += algorithm1(N / 2);
13         else

```

CHAPTER 1. ALGORITHM ANALYSIS

```
14         sum++;
15     return sum;
16 }
```

What is the time complexity of algorithm1? Solve the recurrence equation without the master theorem.

11.

```
1  int algorithm2(int N){
2      int i, j, k, sum = 0;
3      for (i = 1; i <= N; i++)
4          sum++;
5      for (i = 1; i <= N; i++)
6          for (j = N - i; j > i; j--)
7              for (k = 1; k <= j - i; k++)
8                  sum++;
9      for (i = 0; i <= N; i++)
10         for (j = i + 1; j <= N - i; j++)
11             sum++;
12     return sum;
13 }
```

What does the function algorithm2 return in terms of N ? Assume that N is even.

12.

```
1  int algorithm1(int N){
2      if (N == 0)
3          return 0;
4      int sum = 0;
5      for (int i = 0; i < N; i++)
6          for (int j = 0; j < N; j++)
7              for (int k = 0; k < N; k++)
8                  sum++;
9      for (int i = 0; i < 2; i++)
10         for (int j = 0; j < 2; j++)
11             for (int k = 0; k < 2; k++)
12                 sum += algorithm1(N / 2);
13     return sum;
14 }
```

What is the time complexity of algorithm1? Solve the recurrence equation without the master theorem.

13.

```
1  int algorithm1(int N){
```

1.8. PROBLEMS

```

2  if (N == 0)
3      return 0;
4  int sum = 0;
5  for (int i = 0; i < N; i++)
6      for (int j = 0; j < N; j++)
7          sum++;
8  for (int i = 0; i < 2; i++)
9      for (int j = 0; j < 2; j++)
10         for (int k = 0; k < 2; k++)
11             if ((i + j + k) % 3 == 1)
12                 sum += algorithm1(N / 2) + algorithm1(N / 2);
13             else
14                 if ((i + j + k) % 3 == 0)
15                     sum += algorithm1(N / 2);
16                 else
17                     sum++;
18  return sum;
19 }

```

What is the time complexity of algorithm1? Solve the recurrence equation without the master theorem.

14.

```

1  int algorithm2(int N){
2      int i, j, k, sum = 0;
3      for (i = 1; i <= N - i; i++)
4          sum++;
5      for (i = 1; i <= N; i++)
6          for (j = 1; j <= 2 * i; j++)
7              for (k = 1; k <= j - i; k++)
8                  sum++;
9      for (i = 0; i <= N; i++)
10         for (j = i + 1; j <= N; j++)
11             sum++;
12  return sum;
13 }

```

What does the function algorithm2 return in terms of N ? Assume that N is even.

15.

```

1  int algorithm1(int N){
2      if (N == 0)
3          return 0;
4      int sum = 0;
5      for (int i = 0; i < N; i++)

```

CHAPTER 1. ALGORITHM ANALYSIS

```
6   for (int j = 0; j < N; j++)
7       for (int k = 0; k < N; k++)
8           sum++;
9   for (int i = 0; i < 2; i++)
10      for (int j = 0; j < 2; j++)
11          sum += algorithm1(N / 2) + algorithm1(N / 2);
12   return sum;
13 }
```

What is the time complexity of algorithm1? Solve the recurrence equation without the master theorem.

16.

```
1  int algorithm1(int N){
2      int i , j , k, sum = 0;
3      for ( i = 1; i <= N; i++)
4          for ( j = i ; j <= N; j++)
5              for (k = 1; k <= j - i; k++)
6                  sum++;
7      return sum;
8  }
```

What does the function algorithm1 return in terms of N?

17.

```
1  int algorithm2(int N){
2      if (N == 0)
3          return 0;
4      int sum = 0;
5      for (int i = 0; i < N; i++)
6          for (int j = 0; j < N; j++)
7              sum++;
8      for (int i = N; i > 0; i--)
9          for (int j = 0; j < N; j++)
10             for (int k = 0; k < N; k++)
11                 sum++;
12     for (int i = 0; i < 4; i++)
13         for (int j = 0; j < 2; j++)
14             sum += algorithm2(N / 2);
15     return sum;
16 }
```

What is the time complexity of algorithm2? You must construct the recurrence equation and solve it without using the master theorem.

18.

```

1  int algorithm(int N){
2      if (N == 0)
3          return 0;
4      int sum = 0;
5      for (int i = 0; i < N; i++)
6          for (int j = 0; j < N; j++)
7              for (int k = 0; k < N; k++)
8                  sum++;
9      for (int i = 0; i < 4; i++)
10         for (int j = 0; j < 4; j++)
11             if ((i + j) % 2 == 0)
12                 sum += algorithm1(N / 2);
13             else
14                 sum++;
15     return sum;
16 }
```

What is the time complexity of algorithm? You **must** construct the recurrence equation and solve it with the master theorem.

Linked List

Linked list is a data structure, where the elements are stored linearly. Linked list is similar to array in that, but is dissimilar with respect to getting the elements inside of it. Though one can get the elements of an array using indexes, the elements of a linked list can only be reached using pointers.

Insertion and deletion operations can be done faster with linked lists than arrays. While inserting an element in an array or deleting an element from an array requires shifting the elements of the array, since the elements are stored dynamically in a linked list, only changing a few connections is required to do these operations in the linked list.

Table 2.1: Deleting the k 'th element from an array of size N

1	<code>void deleteKth(int* array, int k, int N)</code>	<code>void deleteKth(int[] array, int k){</code>
2	<code>int i;</code>	<code>int i;</code>
3	<code>for (i = k; i < N - 1; i++){</code>	<code>for (i = k; i < array.length - 1; i++){</code>
4	<code>array[i] = array[i + 1];</code>	<code>array[i] = array[i + 1];</code>
5	<code>}</code>	<code>}</code>
6	<code>}</code>	<code>}</code>

Table 2.1 shows the code fragment that deletes the k 'th element from an array of size N . In order to remove the gap occurred after deleting the k 'th element, we need to shift left each element after the k 'th element (Line 4), which makes a total of $N - k$ shifts. On the other hand, as we will see in this chapter, when we delete an element from a doubly linked list, the only thing we need to do is to modify two links. So, the time complexity of deleting the

k 'th element from an array is $\mathcal{O}(N)$, whereas the time complexity of deleting the k 'th element from a doubly linked list is $\mathcal{O}(1)$.

Table 2.2: Inserting a new element at position k into an array of size N

```

1 void insertKth(int* array, int k, int newItem, int N){
2     int i;
3     for (i = N - 2; i >= k; i--){
4         array[i + 1] = array[i];
5     }
6     array[k] = newItem;
7 }

```

```

1 void insertKth(int[] array, int k, int newItem){
2     int i;
3     for (i = array.length - 2; i >= k; i--){
4         array[i + 1] = array[i];
5     }
6     array[k] = newItem;
7 }

```

The same situation also appears when we insert a new element. For example, Table 2.2 shows the code fragment that inserts a new element at position k into an array of size N . In order to obtain the gap to insert the new element, we need to shift left each element after the k 'th element (Line 4), which makes a total of $N - k$ shifts. On the other hand, as we will see in this chapter, when we insert a new element to a doubly linked list, the only thing we need to do is to modify two links. So, the time complexity of inserting a new element at position k to an array is $\mathcal{O}(N)$, whereas the time complexity of inserting a new element to a doubly linked list is $\mathcal{O}(1)$.

For searching a number in a sorted array, the situation is reverse. The time complexity of searching a number in a sorted array is $\mathcal{O}(\log N)$, whereas the time complexity of searching a specific element in a doubly linked list is $\mathcal{O}(N)$. Table 2.3 shows the code fragment that searches number k in a sorted array of size N .

Table 2.3: Searching number k in a sorted array of size N

<pre> 1 void search(int* array, int k, int N){ 2 int left, right, middle; 3 left = 0; 4 right = N - 1; 5 middle = (left + right) / 2; 6 while (left <= right){ 7 if (k < array[middle]) 8 right = middle - 1; 9 else 10 if (k > array[middle]) 11 left = middle + 1; 12 else 13 return middle; 14 middle = (left + right) / 2; 15 } 16 return -1; 17 }</pre>	<pre> void search(int[] array, int k){ int left, right, middle; left = 0; right = array.length - 1; middle = (left + right) / 2; while (left <= right){ if (k < array[middle]) right = middle - 1; else if (k > array[middle]) left = middle + 1; else return middle; middle = (left + right) / 2; } return -1; }</pre>
---	--

2.1 Definition

Every node of a linked list is defined by a data structure. Each node in a linked list has (i) a data field showing the contents of the node, and (ii) a link (pointer) showing the next node. Figure 2.1 shows an example node of a linked list. This element contains 9 as data and a link pointing to the next node.

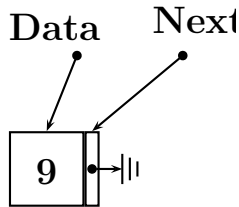


Figure 2.1: An example node containing integer data

Table 2.4 shows the definition of the data structure **node** for this purpose.

Table 2.4: Definition of Node

1	class Node{	1	public class Node{
2	private :	2	int data;
3	int data;	3	Node next;
4	Node* next;	4	public Node(int data){
5	public :	5	this .data = data;
6	Node(int data);	6	next = null ;
7	}	7	}
8		8	}
9	Node::Node(int data){		
10	this —>data = data;		
11	next = nullptr ;		
12	}		

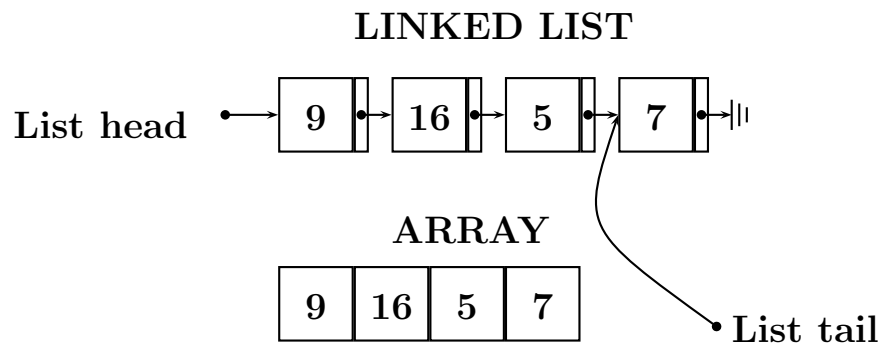


Figure 2.2: An example linked list containing four items.

The **data** field shows the data stored in the node, **next** field contains the link pointing to the next node in the linked list.

An example linked list and its corresponding array can be seen in Figure 2.2. In the array data structure, we can easily access any element using the indexes. On the other hand, in the linked list data structure, we need pointers to access the elements. The head link shows the first node of the linked list. If the head node is **NULL**, the linked list is empty and does not contain any elements. The tail link shows the last node of the linked list and

Table 2.5: Definition of LinkedList

1	class LinkedList {	1	public class LinkedList{
2	private :	2	Node head;
3	Node* head;	3	Node tail;
4	Node* tail;	4	public LinkedList(){
5	public :	5	head = null ;
6	LinkedList ();	6	tail = null ;
7	~LinkedList ();	7	}
8	}	8	}
9			
10	LinkedList :: LinkedList(){		
11	head = nullptr ;		
12	tail = nullptr ;		
13	}		
14	LinkedList ::~ LinkedList(){		
15	Node* tmp = head;		
16	while (tmp != nullptr){		
17	Node* next = tmp->next;		
18	delete tmp;		
19	tmp = next;		
20	}		
21	}		

mainly used for adding elements at the end of the list. If the tail node is NULL, again the linked list is empty and we understand that the linked list has no elements.

Table 2.5 shows the definition of a linked list data structure. The data structure basically contains two fields: **head** field shows the head node of the linked list, **tail** field shows the tail node of the linked list. In the constructor, we set **head** and **tail** fields to NULL to initialize the linked list as empty.

2.2 Singly Linked List Operations

2.2.1 Insertion

Insertion into a linked list can be done in three ways: Inserting at the beginning of the list, inserting at the end of the list, and inserting at the middle of the list.

2.2. SINGLY LINKED LIST OPERATIONS

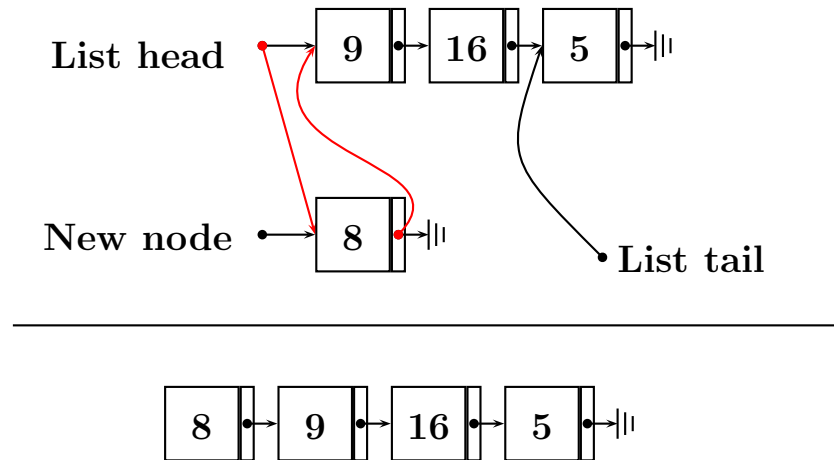


Figure 2.3: Inserting node 8 at the beginning of the linked list

Insert First

Table 2.6 shows the code fragment that adds a **newNode** at the beginning of the linked list **l**. When we insert a new node at the head of the list, **head** field of the linked list shows the new node to be added (Line 5). We also modify the **next** field of the new node to point the original head of the link list (Line 4). If the linked list was empty (Line 2), we also change the tail pointer of the link list and set the **tail** field to the new node (Line 3).

Table 2.6: Inserting at the beginning of the singly linked list

<pre> 1 void LinkedList :: insertFirst (Node* newNode){ 2 if (tail == nullptr) 3 tail = newNode; 4 newNode->next = head; 5 head = newNode; 6 }</pre>	<pre> void insertFirst(Node newNode){ if (tail == null) tail = newNode; newNode.next = head; head = newNode; }</pre>
--	---

In Figure 2.3, we see an example case, where we add a new node 8 at the beginning of a linked list containing originally three nodes. After inserting, field **head** points the added node instead of 9. Also the **next** field of the new node shows 9.

CHAPTER 2. LINKED LIST

Insert Last

The code fragment given in Table 2.7 implements the algorithm that adds a **newNode** at the end of the linked list **l**. First we modify the **next** field of the **tail** node of the link list to show the new node (Line 5). Second, we modify the **tail** link of the linked list to show the new node (Line 6). If the original list was empty (Line 2), we also change the **head** pointer of the link list and set the **head** field to the new node (Line 3).

Table 2.7: Inserting at the end of the singly linked list

<pre>1 void LinkedList :: insertLast (Node* newNode){ 2 if (head == nullptr) 3 head = newNode; 4 else 5 tail->next = newNode; 6 tail = newNode; 7 }</pre>	<pre>void insertLast(Node newNode){ if (head == null) head = newNode; else tail.next = newNode; tail = newNode; }</pre>
--	---

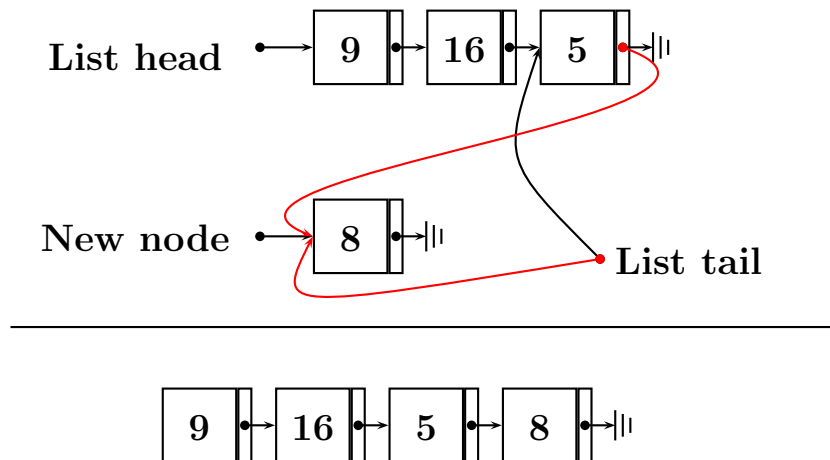


Figure 2.4: Inserting node 8 at the end of the linked list

In Figure 2.4, we see an example case, where we add a new node 8 at the end of a linked list containing originally three nodes. After inserting, field **tail** points to the added node instead of 5. Also the **next** field of the node 5 shows the new node 8.

2.2. SINGLY LINKED LIST OPERATIONS

Table 2.8: Inserting at the middle of a singly linked list

```
1 void LinkedList :: insertMiddle(Node* newNode, Node* previous){  
2     newNode->next = previous->next;  
3     previous->next = newNode;  
4 }
```

```
1 void insertMiddle(Node newNode, Node previous){  
2     newNode.next = previous.next;  
3     previous.next = newNode;  
4 }
```

Insert Middle

In order to insert a new node into the middle (except head or tail) of a linked list, we need to know the node coming before the new node. The previous node's next field will point to the new node.

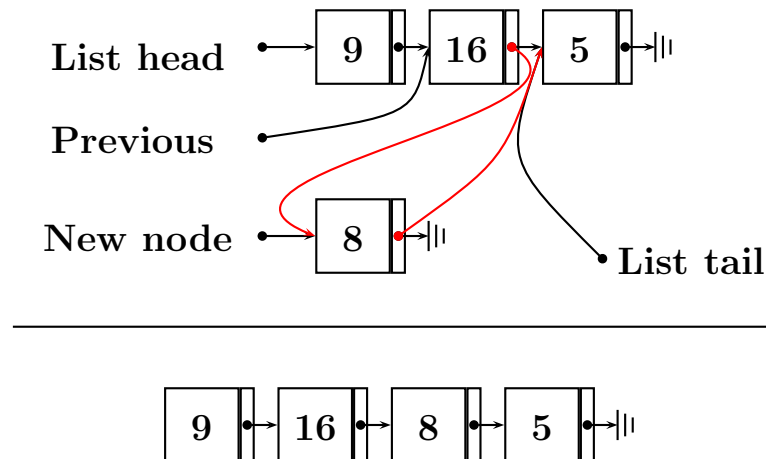


Figure 2.5: Inserting node 8 at the middle of the linked list

Table 2.8 shows the code fragment that adds a **newNode** after the **previous** node. The **next** link of the new node will now point to the next node after the previous node (Line 2). The **next** link of the previous node will point to the new node (Line 3).

In Figure 2.5, we see an example case, where we add a new node 8 after

CHAPTER 2. LINKED LIST

the node 16. The links that change:

- next link of 16 will point 8 instead of 5.
- next link of the new node 8 will point 5.

Singly Linked List Operations

- Insert First: $\mathcal{O}(1)$
- Insert Last: $\mathcal{O}(1)$
- Insert Middle: $\mathcal{O}(1)$

2.2.2 Search

Another basic operation is to search the linked list for specific values. Table 2.9 shows the code fragment that searches the node with a given **value** in the linked list **l**. The search begins with the head node (Line 3) and using the **next** links we traverse whole list (Line 7). At each iteration, we compare the given value with the current node's data (Line 5), and if they are equal we return the current node (Line 6). If the value does not exist in the list, the function returns **NULL** (Line 9).

Table 2.9: Searching a specific value in the linked list

<pre>1 Node* LinkedList::search(int value){ 2 Node* tmp; 3 tmp = head; 4 while (tmp != nullptr){ 5 if (tmp->data == value) 6 return tmp; 7 tmp = tmp->next; 8 } 9 return nullptr; 10 }</pre>	<pre>Node search(int value){ Node tmp; tmp = head; while (tmp != null){ if (tmp.data == value) return tmp; tmp = tmp.next; } return null; }</pre>
--	---

2.2. SINGLY LINKED LIST OPERATIONS

2.2.3 Getting i 'th Node

Table 2.10 shows the algorithm that returns i 'th node in a given linked list l . The main disadvantage of linked lists is the linear complexity of getting the elements by their indexes. In the usual array data structure, it is enough to give the index such as $a[i]$ to get the i 'th element of array a . On the other hand, in the linked list data structure we need to traverse the list until we reach the i 'th element.

Table 2.10: Getting i 'th node of a linked list

<pre>1 Node* LinkedList::nodeIth(int i){ 2 Node* tmp = head; 3 int j = 0; 4 while (tmp != nullptr && j < i){ 5 j++; 6 tmp = tmp->next; 7 } 8 return tmp; 9 }</pre>	<pre>Node nodeIth(int i){ Node tmp = head; int j = 0; while (tmp != null && j < i){ j++; tmp = tmp.next; } return tmp; }</pre>
--	---

Each time we jump from one node to another node in the linked list (Line 6), we increment the counter j by one (Line 5). The traversal can end in two possible ways either the list contain less than i elements (The first condition in Line 4 is false) or we find the i 'th node (When $i = j$, the second condition in Line 4 is false). In the first case, there is no i 'th element, the function returns NULL. In the second case, the function returns **tmp**, namely the i 'th element.

Singly Linked List Operations

- Search: $\mathcal{O}(N)$
- Getting i 'th node: $\mathcal{O}(N)$

2.2.4 Deletion

Deletion from a linked list can be done in three ways: The node that will be deleted can be (i) at the beginning of the list, (ii) at the end of the list, or

CHAPTER 2. LINKED LIST

(ii) at the middle of the list.

Delete First

Table 2.11 shows the code fragment that removes the head node of the linked list l. When we delete the first node, **head** field shows the second element (Line 2). If there was only one node in the beginning, when the first node is deleted, the list will be empty (Line 3), and we need to set the **tail** field to **NULL** (Line 4).

Table 2.11: Deleting from the beginning of the singly linked list

```
1 void LinkedList :: deleteFirst (){\n2   Node* tmp = head;\n3   head = head->next;\n4   if (head == nullptr)\n5       tail = nullptr;\n6   delete tmp;\n7 }
```

```
void deleteFirst(){\n    head = head.next;\n    if (head == null)\n        tail = null;\n}
```

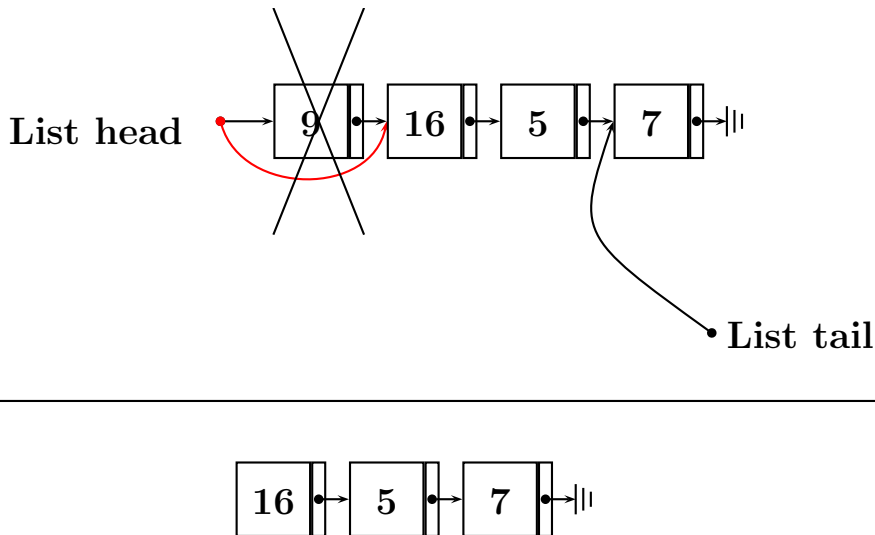


Figure 2.6: Deleting head node 9 from the linked list

2.2. SINGLY LINKED LIST OPERATIONS

In Figure 2.6, we see an example case, where we delete the head node 9 from the linked list. After deleting, the **head** field does not point to 9 anymore, but points to the next node after 9, namely 16.

Delete Last

When we delete the last node from the linked list, we need to identify the new last node, namely the node before the last node. We can only go forward using the links in the singly linked list, so we can not access it directly using links. Therefore, first we need to traverse the linked list and go to the node before the last node, then we update links such that the tail link will show that node.

Table 2.12: Deleting the last node from the singly linked list

<pre>1 void LinkedList :: deleteLast(){ 2 Node *tmp, *previous, *deleted; 3 deleted = tail; 4 tmp = head; 5 previous = nullptr; 6 while (tmp != tail){ 7 previous = tmp; 8 tmp = tmp->next; 9 } 10 if (previous == nullptr) 11 head = nullptr; 12 else 13 previous->next = nullptr; 14 tail = previous; 15 delete deleted; 16 }</pre>	<pre>void deleteLast(){ Node tmp, previous; tmp = head; previous = nullptr; while (tmp != tail){ previous = tmp; tmp = tmp.next; } if (previous == nullptr) head = nullptr; else previous.next = nullptr; tail = previous; }</pre>
---	--

The code fragment given in Table 2.12 implements the algorithm that deletes the last node from the linked list *l*. First, we find the node before the last node (Lines 3 - 8) by traversing the list. While traversing, we need to store both current node (**tmp**) and the node before the current node (**previous**). When we arrive at the end of the list, **previous** node will show the node before the last node.

If the original linked list contains just one node (Line 9), when the last node is deleted, the linked list will be empty and we need to modify the **head**

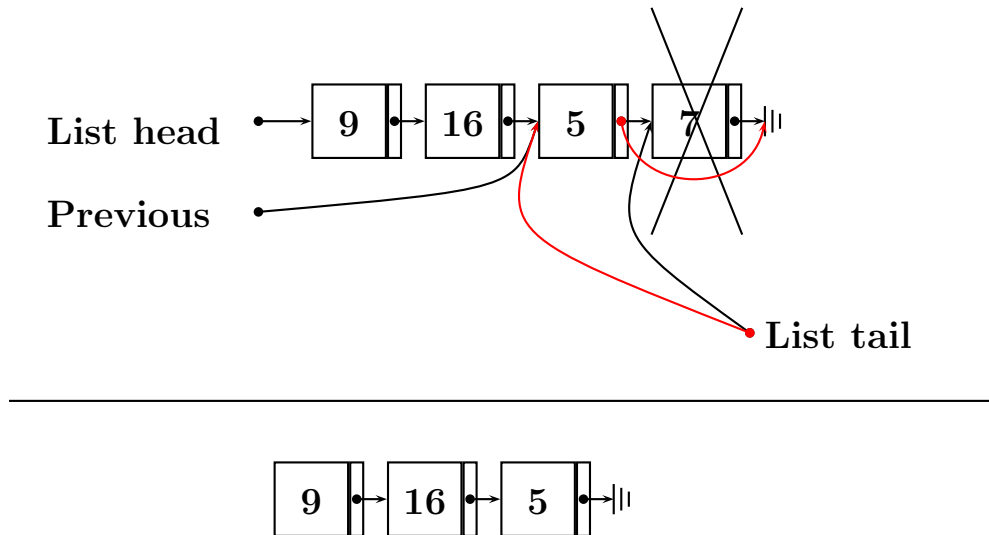


Figure 2.7: Deleting last node 7 from the linked list

link to show NULL (Line 10). If the original list contains more than one node (Line 11), the new **tail** node will be **previous** (Line 13) and the **next** link of the new last node will point to NULL (Line 12).

In Figure 2.7, we see an example case, where we delete the last node 7 from the linked list. After deleting, the **tail** link does not point 7, but 5, namely the node before 7.

Delete Middle

If we are given the data, or the address of the node that contains the data to be deleted, we can easily find and delete the node. If we know the data itself, we can find the node containing that data using the search function in Section 2.2.2.

When we delete a middle node from a singly linked list, the **next** field of the previous node must show the node coming after the deleted node. Since we can only traverse the list from left to right, we can not access the previous node from the current node. Therefore, first we need to search the list, find the previous node and update the links accordingly.

Table 2.13 shows the code fragment that deletes node **s** from a singly linked list **l**. First we find the node before the node **s** (Lines 3-8) by travers-

2.2. SINGLY LINKED LIST OPERATIONS

Table 2.13: Deleting from the middle of a singly linked list

<pre> 1 void LinkedList :: deleteMiddle(Node* s){ 2 Node *tmp, *previous; 3 tmp = head; 4 previous = nullptr; 5 while (tmp != s){ 6 previous = tmp; 7 tmp = tmp->next; 8 } 9 previous->next = s->next; 10 delete s; 11 }</pre>	<pre> void deleteMiddle(Node s){ Node tmp, previous; tmp = head; previous = nullptr; while (tmp != s){ previous = tmp; tmp = tmp.next; } previous.next = s.next; }</pre>
--	--

ing the list. While traversing, we store both current node (**tmp**) and the node before the current node (**previous**). When **tmp** shows the node to be deleted, **previous** node will show the node before the deleted node, and we stop searching. After finding the (to be deleted) node, **next** field of the **previous** node will show the node after the deleted node (Line 9).

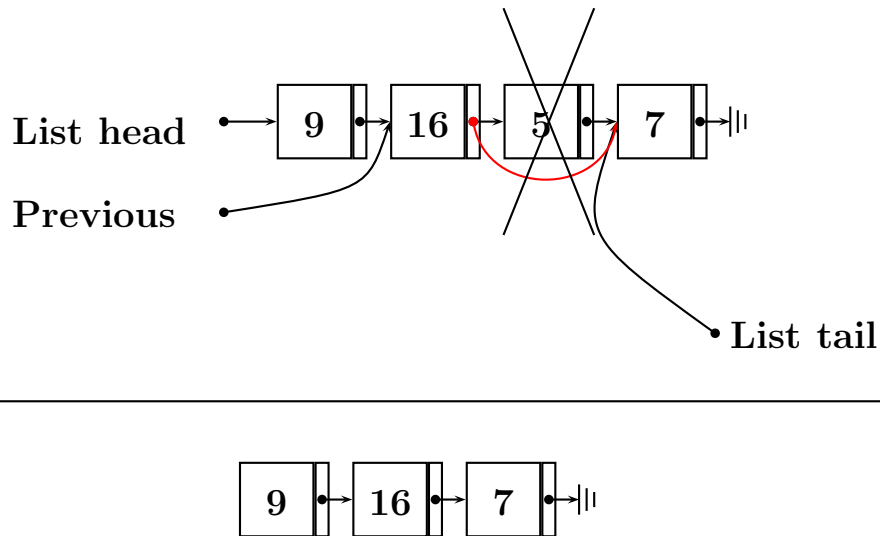





Figure 2.8: Deleting node 5 from the singly linked list

In Figure2.8, we see an example case, where we delete the node 5 from

CHAPTER 2. LINKED LIST

the linked list. After deleting, the next link of 16 does not point to 5, but 7, namely the node after 5.

Singly Linked List Operations

-  Delete First: $\mathcal{O}(1)$
-  Delete Last: $\mathcal{O}(N)$
-  Delete Middle: $\mathcal{O}(N)$

2.2.5 Number of Nodes in a Singly Linked List

Another problem we encounter in linked lists is determining the number of elements in the list. Table 2.14 shows the function that determines the number of elements in a singly linked list. The function takes the linked list as a parameter and returns the number of elements in that linked list. In order to find the number of elements in the linked list, we need to traverse the linked list starting from its head (Line 4) till the end of the list (Line 5) by using pointers. While traversing, we increase the counter by one at each element of the list (Line 7). The value of the counter will give us the number of elements (Line 9).

Table 2.14: Finding number of nodes in a linked list

```
1 int LinkedList::nodeCount(){
2     int count = 0;
3     Node* tmp;
4     tmp = head;
5     while (tmp != nullptr){
6         tmp = tmp->next;
7         count++;
8     }
9     return count;
10 }
```

```
int nodeCount(){
    int count = 0;
    Node tmp;
    tmp = head;
    while (tmp != null){
        tmp = tmp.next;
        count++;
    }
    return count;
}
```

2.3 Merging Two Singly Linked Lists

Yet another problem we encounter is merging two linked lists. The elements of the new linked list will be the elements of the original two lists or a carbon copy of those elements since the elements of these two lists will be used again. In this section we will cover the first case. In this case there is no need to produce copies of the elements of the two linked lists. Merging can be done easily by putting the first element of the second list to the end of the first list.

Table 2.15: Merging two linked lists

```

1  LinkedList LinkedList :: merge(LinkedList l1, LinkedList l2){
2      LinkedList newList;
3      if (l1->head == nullptr)
4          return l2;
5      if (l2->head == nullptr)
6          return l1;
7      newList = LinkedList();
8      newList->head = l1->head;
9      newList->tail = l2->tail;
10     l1->tail->next = l2->head;
11     return newList;
12 }
```

```

1  static LinkedList merge(LinkedList l1, LinkedList l2){
2      LinkedList newList;
3      if (l1.head == null)
4          return l2;
5      if (l2.head == null)
6          return l1;
7      newList = new LinkedList();
8      newList.head = l1.head;
9      newList.tail = l2.tail;
10     l1.tail.next = l2.head;
11     return newList;
12 }
```

Table 2.15 shows the algorithm that merges two linked lists and returns the merged result list. Since the new list will be created in the function, we

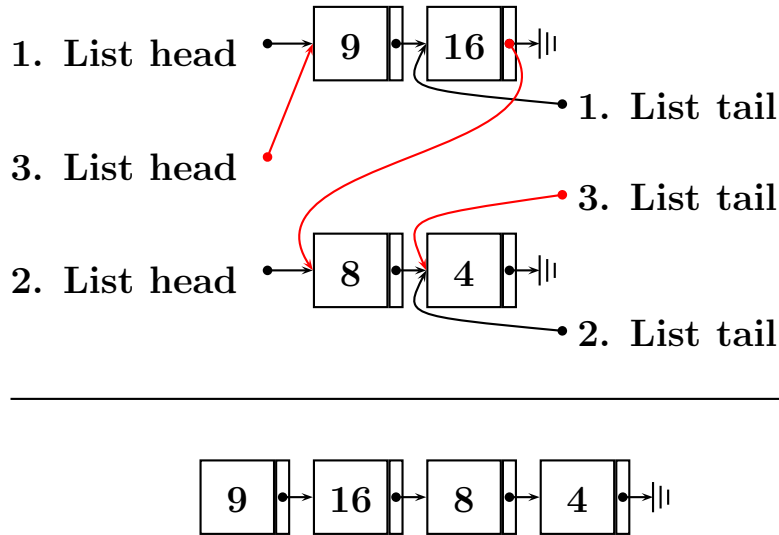


Figure 2.9: Merging linked lists

need to allocate memory for it.

The function first checks if the two lists are empty or not (Line 3, 5), if one of them is empty returns the other list (Line 4, 6). If both of the lists contain at least one element, the result list is created (Line 7), the first element of the result list points to the first element of the first list (Line 8), the last element of the result list points to the last element of the second list (Line 9) and the pointer that shows the next element after the last element of the first list is changed from NULL to the first element of the second list (Line 10).

Figure 2.9 shows an example of merging two linked lists. The first list contains 9 and 16, the second list contains 8 and 4. The first element of the result list is the first element of the first list, namely 9, the last element of the result list is the last element of the second list, namely 4.

2.4 Doubly Linked List

Although singly linked list has the important advantage of constant time insertion and deletion compared to the arrays, it also includes an important

2.4. DOUBLY LINKED LIST

disadvantage: we can traverse the singly linked list only in forward manner. This disadvantage shows itself in the following basic operations:

- Inserting in the middle of a singly linked list (Table 2.8) is only possible if we have a pointer to the node (named **before**) which is before the node where the insertion is to be made. If we do not have the pointer to the node **before**, we need to traverse the list starting from the head node until node **before**.
- If we delete the last element of a singly linked list (Table 2.12), we need to traverse the list starting from the head node until the tail node in order to determine the new tail node of the list.
- Deleting from the middle of a singly linked list (Table 2.13) is only possible if we have a pointer to the node (named **before**) which is before the node to be deleted. If we do not have the pointer to the node **before**, we need to traverse the list starting from the head node until the node **before**.

All these disadvantages can be removed by adding to each node a new pointer, which shows the node before the current node. This linked list structure, where each node contains two pointers instead of one is called doubly linked list. In a doubly linked list, each element has a link **next** showing the next node, and a link **previous** showing the node before the current node.

- In order to insert in the middle of a doubly linked list (Table 2.20) we only need a link to the node before which a new node will be inserted. The node before the current node can be found by using the link **previous** of the current node.
- When we delete the last element of a doubly linked list (Table 2.22), the new last element of the doubly linked list will be the node that is shown by the link **previous** of the old last node of the list.
- In order to delete from the middle of a doubly linked list (Table 2.23) we only need a link to the node to be deleted. We can access the node before the deleted node by using the link **previous** of that deleted node.

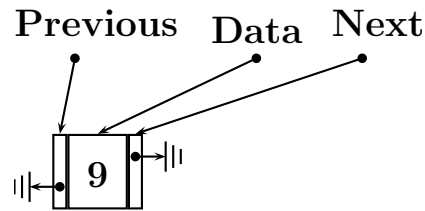


Figure 2.10: Node data structure which contains an integer (For doubly linked list)

Table 2.16: Definition of a node (For doubly linked list)

1	class DoubleNode{	1	public class DoubleNode{
2	private :	2	int data;
3	int data;	3	DoubleNode next;
4	DoubleNode* next;	4	DoubleNode previous;
5	DoubleNode* previous;	5	public DoubleNode(int data){
6	public :	6	this .data = data;
7	DoubleNode(int data);	7	next = null ;
8	}	8	previous = null ;
9		9	}
10	DoubleNode::DoubleNode(int data){	10	}
11	this —>data = data;		
12	next = nullptr ;		
13	previous = nullptr ;		
14	}		

An example node of a doubly linked list can be seen in Figure 2.10. Here, this node contains the data 9, a link **next** pointing to the next node and a link **previous** pointing to the node before.

Table 2.16 shows the definition of a node in a doubly linked list. Each node of the doubly linked list **struct doublenode**. **data** field shows the data in the node, **next** field shows the link pointing to the next node in the doubly linked list, and **previous** field shows the link pointing to the node before.

An example doubly linked list and its corresponding array is shown in Figure 2.11. According to this figure, the head node of the linked list contains data 9, a **next** link pointing to the next node 16 and a **previous** link **NULL** since there is no node before the head node. The tail link showing the last

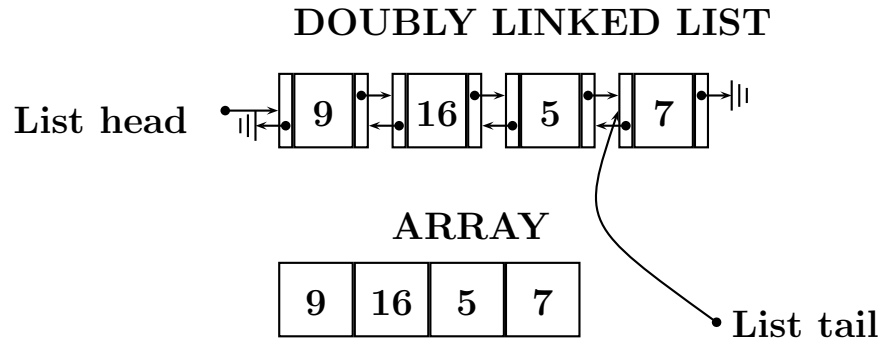


Figure 2.11: An example doubly linked list and its corresponding array

node of a doubly linked list is usually used when inserting a new node at the end of the list. The **next** link of the tail node of a doubly linked list points to NULL. The names of these two pointers, namely **head** and **tail** are the same as in the singly linked list.

Table 2.17 shows the definition of doubly linked list containing integers. The list itself is a **struct doublylinkedlist** composed of an head and a tail node. **head** field shows the head node of the doubly linked list, **tail** field shows the tail node of the doubly linked list. When a new doubly linked list is created, we set **head** and **tail** fields to NULL.

2.5 Doubly Linked List Operations

2.5.1 Insertion

Insertion into a doubly linked list can be done in three ways: Inserting at the beginning of the list, inserting at the end of the list, and inserting at the middle of the list.

Insert First

Table 2.17: Definition of DoublyLinkedList

1	class DoublyLinkedList{	1	public class DoublyLinkedList{
2	private :	2	DoubleNode head;
3	DoubleNode* head;	3	DoubleNode tail;
4	DoubleNode* tail;	4	public DoublyLinkedList(){
5	public :	5	head = null ;
6	DoublyLinkedList();	6	tail = null ;
7	~DoublyLinkedList();	7	}
8	}	8	}
9			
10	DoublyLinkedList::DoublyLinkedList(){		
11	head = nullptr ;		
12	tail = nullptr ;		
13	}		
14	DoublyLinkedList::~~DoublyLinkedList(){		
15	DoubleNode* tmp = head;		
16	while (tmp != nullptr){		
17	DoubleNode* next = tmp->next;		
18	delete tmp;		
19	tmp = next;		
20	}		
21	}		

Table 2.18 shows the algorithm that inserts a **newNode** at the beginning of the doubly linked list *l*. When we insert a new node at the head of the doubly linked list, **head** link of the doubly linked list will point to the **newNode** (Line 7). Also we need to put the old head of the doubly linked list as the next node of the **newNode** (Line 6). If there is at least one node before insertion, we need to update the **previous** link of the head node to point to the **newNode** (Line 4-5). As a last step if the list is empty, the link pointing to the tail node of the doubly linked list will point to the **newNode** (Line 2-3).

In Figure 2.12, a new node with data 8 is inserted at the beginning of a doubly linked list originally containing three nodes. The changes made can be summarized as:

- **head** link does not point to the node with data 9 but the new node.
- The **previous** link of the node with data 9 does not point **NULL** but the new node.
- The **next** link of the new node points to 9.

2.5. DOUBLY LINKED LIST OPERATIONS

Table 2.18: Inserting at the beginning of the doubly linked list

```

1 void DoublyLinkedList:: insertFirst (DoubleNode* newNode){
2     if ( tail == nullptr)
3         tail = newNode;
4     else
5         head->previous = newNode;
6     newNode->next = head;
7     head = newNode;
8 }

```

```

1 void insertFirst(DoubleNode newNode){
2     if (tail == null)
3         tail = newNode;
4     else
5         head.previous = newNode;
6     newNode.next = head;
7     head = newNode;
8 }

```

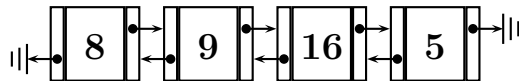
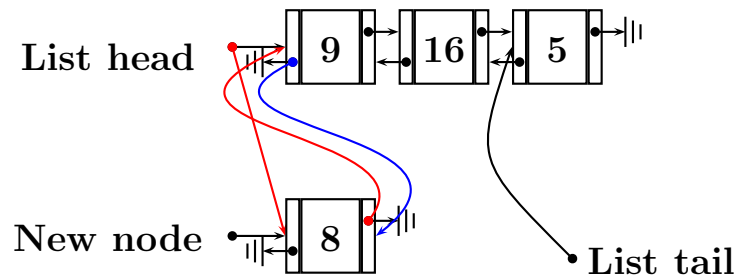


Figure 2.12: Inserting at the beginning of the doubly linked list

Insert Last

In order to insert a new node at the end of the doubly linked list, it is enough to add the new node after the last node.

CHAPTER 2. LINKED LIST

Table 2.19: Inserting a new node at the end of the doubly linked list

```
1 void DoublyLinkedList::insertLast (DoubleNode* newNode){  
2     if (head == nullptr)  
3         head = newNode;  
4     else  
5         tail->next = newNode;  
6     newNode->previous = tail;  
7     tail = newNode;  
8 }
```

```
1 void insertLast(DoubleNode newNode){  
2     if (head == null)  
3         head = newNode;  
4     else  
5         tail.next = newNode;  
6     newNode.previous = tail;  
7     tail = newNode;  
8 }
```

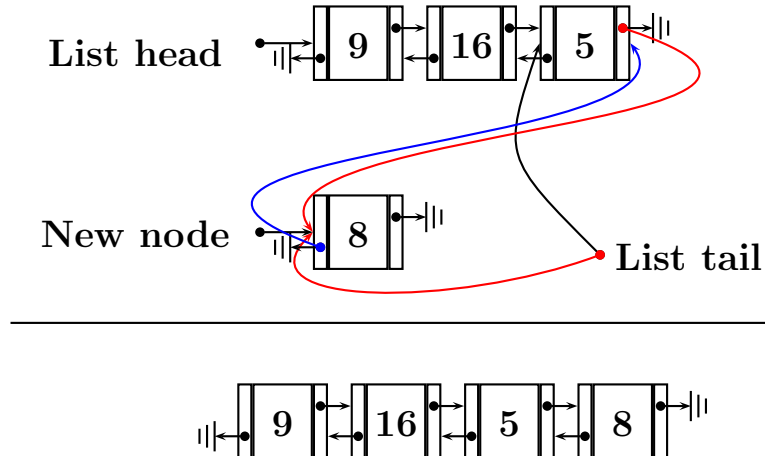


Figure 2.13: Inserting at the end of the doubly linked list

Table 2.19 shows the algorithm that adds the node `newNode` at the end of the doubly linked list `l`. If the list is empty, the head node of the list will be the new node (Line 2-3). If the list contains at least one node, the `previous`

2.5. DOUBLY LINKED LIST OPERATIONS

Table 2.20: The algorithm that inserts a new node into the middle of the doubly linked list

```
1 void DoublyLinkedList::insertMiddle(DoubleNode* newNode, DoubleNode* previous){
2     newNode->next = previous->next;
3     newNode->previous = previous;
4     previous->next->previous = newNode;
5     previous->next = newNode;
6 }
```

```
1 void insertMiddle(DoubleNode newNode, DoubleNode previous){
2     newNode.next = previous.next;
3     newNode.previous = previous;
4     previous.next.previous = newNode;
5     previous.next = newNode;
6 }
```

link of the new node points the last node of the list (Line 6). The **next** link of the last node of the list will be changed to point to the new node (Line 5).

Figure 2.13 shows an example case, where we add node with data 8 at the end of the doubly linked list containing 3 elements. The changes to the links can be summarized as follows:

- **tail** field does not point to the node with data 5 but the new node inserted.
- The **next** link of the node with data 5 does not point to NULL but the new node inserted.
- The **previous** link of the new node points to 5.

Insert Middle

In order to insert in the middle of a doubly linked list we need to find the address of the node that will come before the new node. The **next** link of the node that is before the new node will point to the new node, the **previous** link of the new node will point to the node that is before the new node.

In the algorithm in Table 2.20, **newNode** shows the added node, **previous** shows the node before the new node. 4 links need to be updated.

CHAPTER 2. LINKED LIST

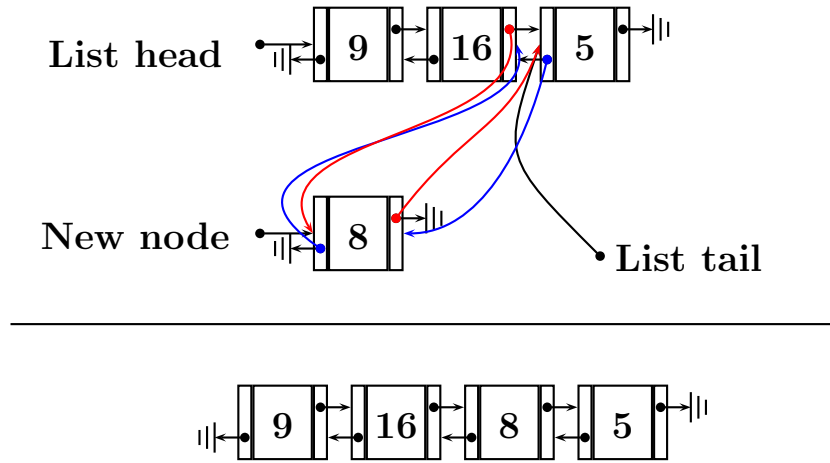


Figure 2.14: Inserting in the middle of an example doubly linked list

- The **next** link of the new node points to the node coming after the node **previous** (Line 2).
- The **previous** link of the new node points to node **previous** (Line 3).
- The **previous** link of the node after the node **previous** will point to the new node (Line 4).
- The **next** link of the node **previous** will point to the new node (Line 5).

Figure 2.14 shows a case where the node with data 8 inserted after the node with data 16. Updated links are:

- The **next** link of 8 will point to 5.
- The **previous** link of 8 will point to 16.
- The **previous** link of 5 will point to 8.
- The **next** link of 16 will point to 8.

2.5. DOUBLY LINKED LIST OPERATIONS

Doubly Linked List Operations

- Insert First: $\mathcal{O}(1)$
- Insert Last: $\mathcal{O}(1)$
- Insert Middle: $\mathcal{O}(1)$

2.5.2 Deletion

Like deleting from the singly linked list, in deleting from a doubly linked list there can be 3 different cases. The node that will be deleted can be (i) at the beginning of the list, (ii) at the end of the list, or (ii) at the middle of the list.

Delete First

Table 2.21 shows the algorithm that removes the head node of the doubly linked list l. When the head node of the linked list is deleted, head field will show the node after the head node (second node) (Line 2). The previous link of the new head node will be changed to NULL (Line 6). When the linked list becomes empty after deleting the head node (Line 3), we set the tail node of the linked list to NULL (Line 4).

Table 2.21: Deleting from the beginning of the singly linked list

<pre>1 void DoublyLinkedList:: deleteFirst (){\n2 DoubleNode* tmp;\n3 tmp = head;\n4 head = head->next;\n5 if (head == nullptr)\n6 tail = nullptr;\n7 else\n8 head->previous = nullptr;\n9 delete tmp;\n10 }</pre>	<pre>void deleteFirst(){\n head = head.next;\n if (head == null)\n tail = null;\n else\n head.previous = null;\n}</pre>
--	--

Figure 2.15 shows the deletion of the head node of an example doubly linked list. The head node of the linked list does not show 9 but the node

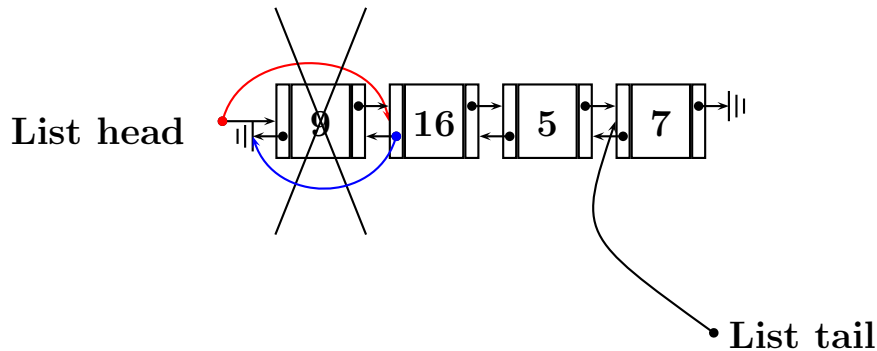


Figure 2.15: Deleting head node from an example doubly linked list

16 pointed by the **next** link of 9. Also the **previous** link of 16 does not point 9 but NULL.

Delete Last

When we delete the last node of a doubly linked list, we need to determine the new last node of the list. Since we can traverse both in forward and backward manner using the links in a doubly linked list, in order to access the node before the last node, we can easily use the **previous** link of the last node.

When we delete the last node from a doubly linked list, **tail** field will point to the node before the last node (Line 2). The **next** link of the new last node will be changed to NULL (Line 6). If there is a single node in the original doubly linked list, when we delete this node, the list becomes empty (Line 3), and the head node of the list will be NULL (Line 4).

Figure 2.16 shows the deletion of the last node from an example doubly linked list. The last node of the list is not 7 but the node 5 that is pointed by **previous** link of 7. Also the **next** link of 5 does not point 7 but NULL.

2.5. DOUBLY LINKED LIST OPERATIONS

Table 2.22: Deleting from the end of the doubly linked list

```

1 void DoublyLinkedList::deleteLast(){
2     DoubleNode* tmp;
3     tmp = tail;
4     tail = tail->previous;
5     if (tail == nullptr)
6         head = nullptr;
7     else
8         tail->next = nullptr;
9     delete tail;
10 }

```

```

void deleteLast(){
    tail = tail.previous;
    if (tail == null)
        head = null;
    else
        tail.next = null;
}

```

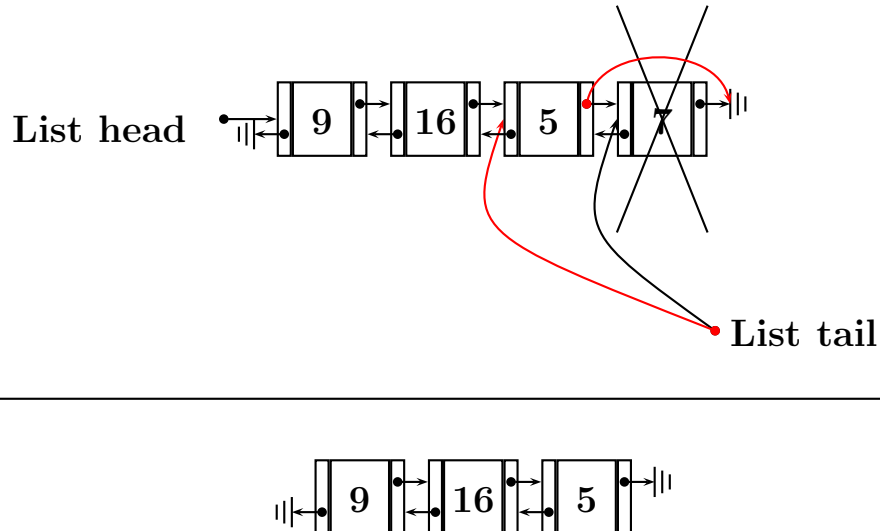


Figure 2.16: Deleting the last node from an example doubly linked list

Delete Middle

Deleting from the middle of a doubly linked list can be done easily if the data of the deleted node or a link to the deleted node is given. If the data of the node is given, we can find a link to the node using the search function in Section 2.2.2.

When we delete the node given its address, the **next** link of the node coming before the deleted node must point to the node coming after the

CHAPTER 2. LINKED LIST

deleted node. In the singly linked list we can only traverse in the forward manner but in doubly linked list using the **previous** links we can also traverse both in forward and backward manner. Therefore, contrary to the singly linked list, in the doubly linked list we do not need to find the node coming from the deleted node. We only need use the **previous** link to determine the previous coming node (Line 3).

Table 2.23: Deleting from the middle of a doubly linked list




```
1 void DoublyLinkedList::deleteMiddle(DoubleNode* s){
2     s->next->previous = s->previous;
3     s->previous->next = s->next;
4     delete s;
5 }
```

```
1 void deleteMiddle(DoubleNode s){
2     s.next.previous = s.previous;
3     s.previous.next = s.next;
4 }
```

Table 2.23 shows the function that deletes a specific element from the middle of a doubly linked list. The **next** link of the node coming before the deleted node will be changed to point to the node coming after the deleted node (Line 3), the **previous** link of the node coming after the deleted node will be changed to point to the node coming before the deleted node (Line 2).

Figure 2.17 shows deletion of the node with data 5. The **next** link of 16 does not point to 5 but the node 7 coming after the deleted node 5. The **previous** link of 7 does not point to 5 but the node 16 coming before the deleted node 5.

Doubly Linked List Operations

-  Delete First: $\mathcal{O}(1)$
-  Delete Last: $\mathcal{O}(1)$
-  Delete Middle: $\mathcal{O}(1)$

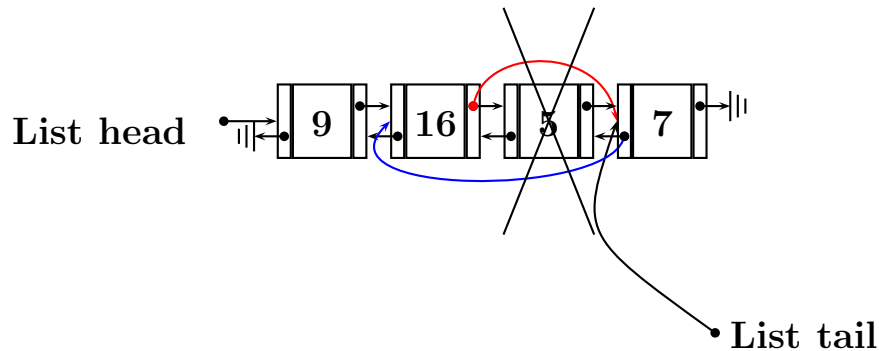


Figure 2.17: Deleting from the middle of a doubly linked list

2.6 Circular Linked List

In some problems, the data may not be stored in an array, where first and last elements are identified, but in a circular structure. The property of a circular structure is, it has no end, in other words, the next element after the last element is simply the first element. In linear lists there are two links pointing to the head and tail of the list, whereas in circular lists there is only one link, where using this link it is possible to add a node both to the start and end of the circular list.

An example circular list and its corresponding array structure is given in Figure 2.18. According to the figure, the elements of the example circular list are 9, 16, 5, and 7 respectively. The **next** link of the last element 7 of the circular list, as we explained above, points to the first element 9 of the circular list, whereas the **previous** link of the first element 9 of the circular list points to the last element 7 of the circular list.

Table 2.24 shows the definition of the circular list containing integers. The list structure itself is a **struct circular list** which contains only a link pointing to a single node. **head** field shows the last element of the list. At the time of

CIRCULAR LINKED LIST

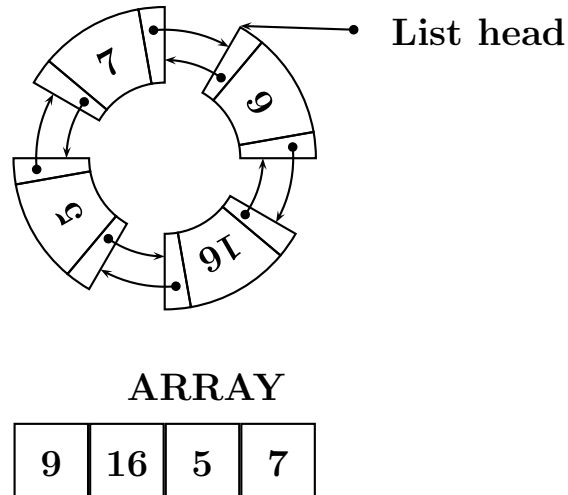


Figure 2.18: An example circular linked list with corresponding array

the construction of the list, it does not contain any elements, therefore **head** is set to value **NULL**.

2.6.1 Insertion

Since inserting at the beginning of the circular list and inserting at the end of the circular list are the same, we only show here inserting at the beginning of the circular list.

Table 2.25 shows the algorithm which add node **newNode** at the circular list **l**. Before adding at the beginning of the circular list

- there are no elements in the original list (Line 2). Because of the circular property of the list, the **next** and the **previous** links of the **newNode** points again the **newNode** itself (Line 3, 4).
- there is at least one element in the original list (Line 5). In this case, the **next** link of the **newNode** points to the original head node of the

2.6. CIRCULAR LINKED LIST

Table 2.24: Definition of circular linked list

1	class CircularList{	1	public class CircularList{
2	private:	2	DoubleNode head;
3	DoubleNode* head;	3	public CircularList(){
4	public	4	head = null ;
5	CircularList ():	5	}
6	~CircularList ():	6	}
7	}		
8			
9	CircularList :: CircularList(){		
10	head = nullptr ;		
11	}		
12	CircularList ::~ CircularList(){		
13	DoubleNode* tmp = head;		
14	while (tmp != nullptr){		
15	DoubleNode* next = tmp->next;		
16	delete tmp;		
17	tmp = next;		
18	}		
19	}		

circular link list (Line 6), whereas the **previous** link of the **newNode** points to the last node of the list (Line 7). Also the **previous** link of the original head node of the circular link list points to the **newNode** (Line 9), whereas the **next** link of the last node of the circular link list points to the **newNode** (Line 8).

As a last step, the **head** link of the list is updated to show the **newNode** (Line 11).

Figure 2.19 shows the case where a node with data 8 is inserted into a circular linked list with four elements. Changes made to links can be summarized as follows:

- The **previous** link of the old head node with data 9 does not point to 7 but the new node added.
- The **next** link of the tail node with data 7 does not point to 9 but the new node added.
- The **next** link of the new node points to the old head node 9.

Table 2.25: Inserting at the beginning of the circular linked list

```

1 void CircularList :: insertFirst (DoubleNode* newNode){
2     if (head == nullptr){
3         newNode->next = newNode;
4         newNode->previous = newNode;
5     }else{
6         newNode->next = head;
7         newNode->previous = head->previous;
8         head->previous->next = newNode;
9         head->previous = newNode;
10    }
11    head = newNode;
12 }

```

```

1 void insertFirst(DoubleNode newNode){
2     if (head == null){
3         newNode.next = newNode;
4         newNode.previous = newNode;
5     }else{
6         newNode.next = head;
7         newNode.previous = head.previous;
8         head.previous.next = newNode;
9         head.previous = newNode;
10    }
11    head = newNode;
12 }

```

- The previous link of the new node points to the tail node with data 7.

2.6.2 Deletion

Since there is little difference between deleting the head node of the circular linked list or tail node of the circular linked list, we will present only deleting the head node of the list here.

Table 2.26 shows the algorithm that deletes the head node from a circular linked list l. Two cases are possible: If there is only one node in the list (Line 2), when we delete this node, there will be no nodes left and therefore the first field of the list is set to NULL (Line 3). If there are more than one node in the list (Line 4)

2.6. CIRCULAR LINKED LIST

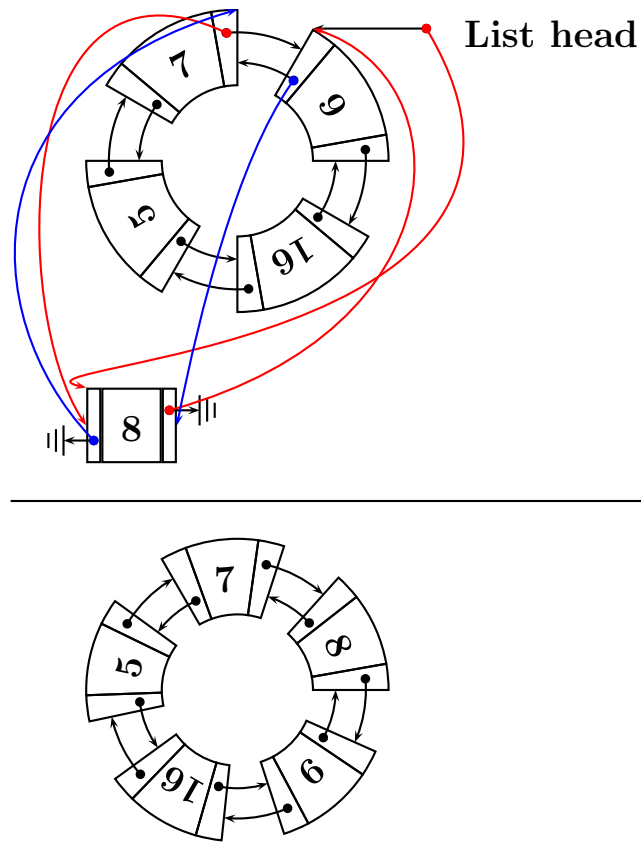


Figure 2.19: Inserting node 8 at the beginning of an example circular linked list

Table 2.26: Deleting head node from the beginning of circular linked list

<pre> 1 void CircularList :: deleteFirst (){ 2 if (head->next == head) 3 head = nullptr; 4 else{ 5 head->previous->next = head->next; 6 head->next->previous = head->previ 7 head = head->next; 8 } 9 }</pre>	<pre> void deleteFirst(){ if (head.next == head) head = null; else{ head.previous.next = head.next; head.next.previous = head.previous; head = head.next; } }</pre>
---	---

CHAPTER 2. LINKED LIST

- The **next** link of the tail node of the list points to the second node of the list (Line 5),
- The **previous** link of the second node of the list points to the tail node of the list (Line 6),
- The new head node of the list is the old second node of the list (Line 7).

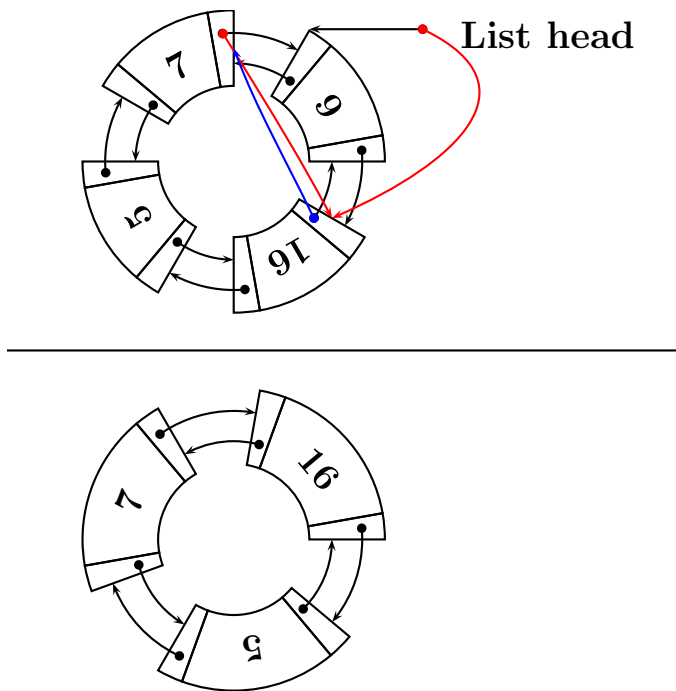


Figure 2.20: Deleting first node 9 from the beginning of an example circular linked list

Figure 2.20 shows deleting the head node of an example circular linked list. The changes made can be summarized as follows:

- The **next** link of 7 does not point to 9, but 16 that is pointed by its next link.
- The **previous** link of 16 does not point to 9, but 7 that is pointer by its previous link.

- The head node of the list is not 9, but 16 that is pointed by its **next** link.

2.7 Application: Polynomial Arithmetic

In this section, we will see an application of linked lists, whereby we will show how addition, subtraction, and multiplication operations on polynomials with a single variable can be done using linked lists. $f(x)$ polynomial with a single variable can be written as

$$f(x) = \sum_{i=0}^N a_i x^i = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$$

Here x represents the variable of the polynomial, $a_0, a_1, a_2, \dots, a_n$ represent coefficients of the 0^{th} , 1^{th} , 2^{nd} , \dots , n^{th} degree terms respectively. If most of these coefficients are not 0, we can store these coefficients in an array. For example, Figure 2.21 shows the array representation of the polynomial $4x^5 + 3x^4 + 6x^2 - 8x - 7$. Since there is no x^3 term in the polynomial, the second element of the array is zero.

4	3	0	6	-8	-7
---	---	---	---	----	----

Figure 2.21: Array representation of the polynomial $4x^5 + 3x^4 + 6x^2 - 8x - 7$

On the other hand, if most of the coefficients are zero, like in the polynomial

$$8x^{74} - 3x^{42} - 12x^{23} + 4$$

where only four terms have non-zero coefficients, whereas the other 71 terms' coefficients are zero. In these cases, representing polynomials with an array can cost extra memory which is not needed.

Yet another alternative representation is linked lists. In the linked list representation, each node corresponds to a non-zero term. Different from the normal linked lists, each node does not contain one data field, but two fields as degree and coefficient of the term. As an example, Figure 2.22 shows the linked list representation of the polynomial $8x^{74} - 3x^{42} - 12x^{23} + 4$. As can

Table 2.27: Definition of a term of polynomial with single variable

1	class Term{	1	public class Term{
2	private :	2	int coefficient ;
3	int coefficient ;	3	int degree;
4	int degree;	4	public Term(int coefficient, int degree){
5	public :	5	this .coefficient = coefficient ;
6	Term(int coefficient , int degree);	6	this .degree = degree;
7	}	7	}
8		8	}
9	Term::Term(int coefficient , int degree){		
10	this ->coefficient = coefficient ;		
11	this ->degree = degree;		
12	}		

be seen, in this representation terms with zero coefficients are not stored and therefore a significant gain from the memory for storing the polynomial is accomplished.

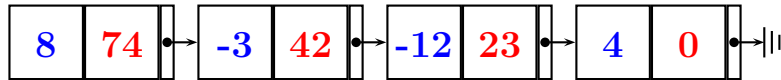


Figure 2.22: Linked list representation of the polynomial $8x^{74} - 3x^{42} - 12x^{23} + 4$

Table 2.27 shows the definition of the Term class which stores the information of a term of a polynomial. Each term of the polynomial is represented with class Term. coefficient field shows the coefficient of the term, degree field shows the degree of the term.

2.7.1 Adding Two Polynomials

When two polynomials are added (i) the coefficients of the terms with the same degree existed in both polynomials are added (ii) the terms with no corresponding degree in one of the polynomials are simply redound on the resulting polynomial. For example, when we add polynomial

$$4x^5 + 3x^2 - 7x + 8$$

2.7. APPLICATION: POLYNOMIAL ARITHMETIC

with the polynomial

$$2x^4 + 6x^2 + 7x$$

the coefficients of the terms with the same degree existed in both polynomials, namely of the terms x^2 , x , and 1, are added, the terms with no corresponding degree in one of the polynomials, namely the terms x^5 and x^4 , are redound on the resulting polynomial. As a result, the sum of these two polynomials will be

$$4x^5 + 2x^4 + (3 + 6)x^2 + (7 - 7)x + 8 = 4x^5 + 2x^4 + 9x^2 + 8$$

Table 2.29 shows the algorithm that finds the sum of two polynomials represented via linked lists. Since the terms of the polynomials are sorted according to their powers, while summing the polynomials we use two links showing current terms processed in each polynomial. *i* link shows the current term processed in the first polynomial (Line 5), whereas *j* link shows the current term processed in the second polynomial (Line 6).

- If both terms have the same degree (Line 9), the degree of the new term that will be added in the resulting polynomial is the common degree (Line 11), the coefficient on the other hand is the sum of those two corresponding terms in the two polynomials (Line 10). Since both terms are used in this case, both links *i* and *j* are advanced (Lines 12-13).
- If the degree of the first term is larger than the second (Line 15), the degree of the new term that will be added in the resulting polynomial is the degree of the first term (Line 17), the coefficient is again the coefficient of the first term (Line 16). Since the first term is used only, link of the first polynomial, namely *i* is advanced (Line 18).
- If the degree of the second term is larger than the first (Line 19), the degree of the new term that will be added in the resulting polynomial is the degree of the second term (Line 21), the coefficient is again the coefficient of the second term (Line 20). Since the second term is used only, link of the second polynomial, namely *j* is advanced (Line 22).

When the terms of one of the two polynomials are processed, **while** loop will end (Line 8). If the first (second) polynomial is over, starting from the link of the first (second) polynomial (Lines 29-32) remaining terms with lower order are added one by one to the resulting polynomial (Lines 33 - 36).

CHAPTER 2. LINKED LIST

Table 2.28: The algorithm that finds the sum of two polynomials (C++)

```
1 LinkedList LinkedList::add(LinkedList p1, LinkedList p2){
2     Node *i, *j, *k, *node;
3     LinkedList result;
4     int coefficient = 0, degree = 0;
5     i = p1.head;
6     j = p2.head;
7     result = LinkedList();
8     while (i != nullptr && j != nullptr){
9         if (i->data.degree == j->data.degree){
10             coefficient = i->data.coefficient + j->data.coefficient;
11             degree = i->data.degree;
12             i = i->next;
13             j = j->next;
14         } else
15             if (i->data.degree > j->data.degree){
16                 coefficient = i->data.coefficient;
17                 degree = i->data.degree;
18                 i = i->next;
19             } else {
20                 coefficient = j->data.coefficient;
21                 degree = j->data.degree;
22                 j = j->next;
23             }
24         if (coefficient != 0){
25             node = new Node(Term(coefficient, degree));
26             result.insertLast(node);
27         }
28     }
29     if (i == nullptr)
30         k = j;
31     else
32         k = i;
33     while (k != nullptr){
34         node = new Node(Term(k->data.coefficient, k->data.degree));
35         result.insertLast(node);
36         k = k->next;
37     }
38     return result;
39 }
```

Figure 2.23 shows adding two polynomials $4x^5 + 3x^2 - 7x + 8$ and $2x^4 +$

2.7. APPLICATION: POLYNOMIAL ARITHMETIC

Table 2.29: The algorithm that finds the sum of two polynomials (Java)

```
1 static LinkedList add(LinkedList p1, LinkedList p2){
2     Node i, j, k, node;
3     LinkedList result;
4     int coefficient = 0, degree = 0;
5     i = p1.head;
6     j = p2.head;
7     result = new LinkedList();
8     while (i != null && j != null){
9         if (i.data.degree == j.data.degree){
10             coefficient = i.data.coefficient + j.data.coefficient;
11             degree = i.data.degree;
12             i = i.next;
13             j = j.next;
14         } else
15             if (i.data.degree > j.data.degree){
16                 coefficient = i.data.coefficient;
17                 degree = i.data.degree;
18                 i = i.next;
19             } else {
20                 coefficient = j.data.coefficient;
21                 degree = j.data.degree;
22                 j = j.next;
23             }
24         if (coefficient != 0){
25             node = new Node(new Term(coefficient, degree));
26             result.insertLast(node);
27         }
28     }
29     if (i == null)
30         k = j;
31     else
32         k = i;
33     while (k != null){
34         node = new Node(new Term(k.data.coefficient, k.data.degree));
35         result.insertLast(node);
36         k = k.next;
37     }
38     return result;
39 }
```

$6x^2 + 7x$ using the algorithm in Table 2.29.

CHAPTER 2. LINKED LIST

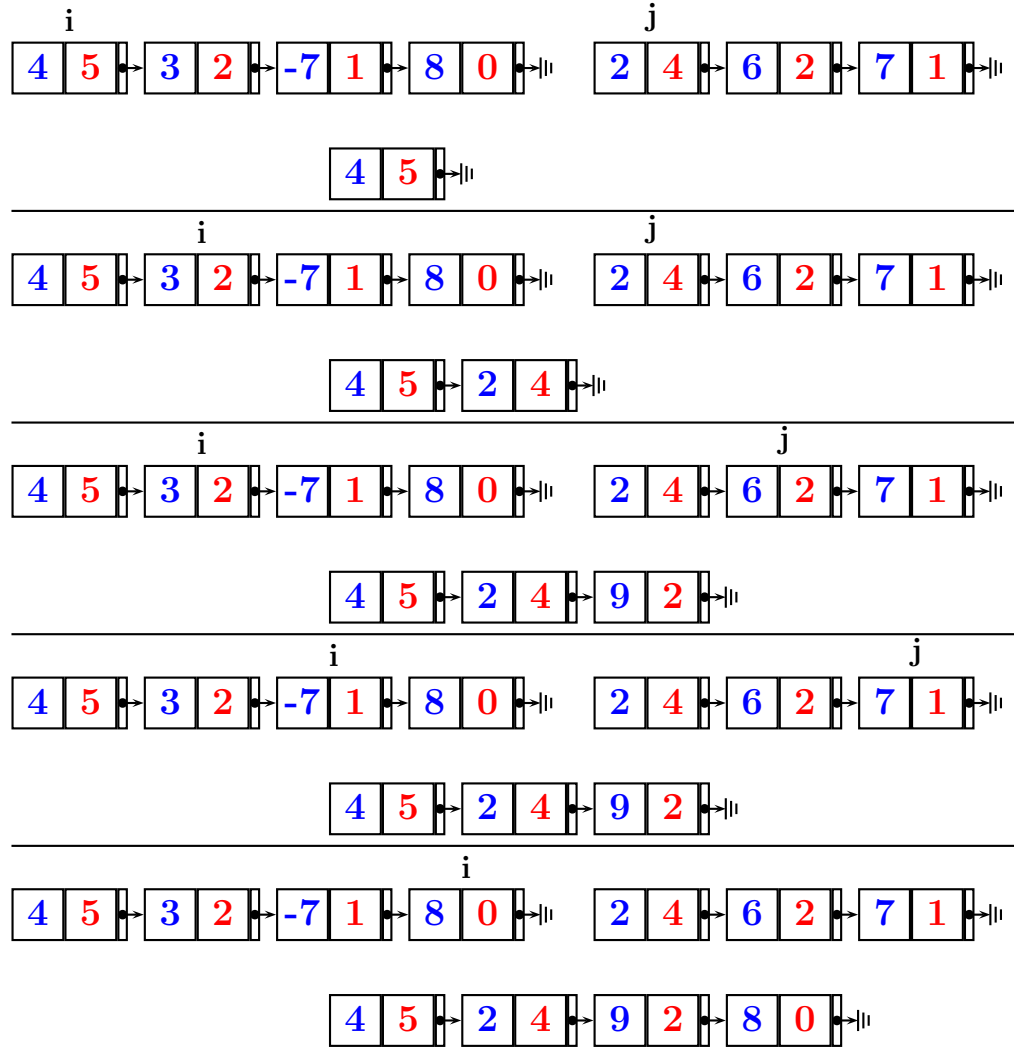


Figure 2.23: Adding polynomials $4x^5 + 3x^2 - 7x + 8$ and $2x^4 + 6x^2 + 7x$

1. The degree of the first term ($4x^5$) is larger than the second term ($2x^4$). The first term is added to the resulting polynomial and link *i* is advanced.
2. The degree of the first term ($3x^2$) is smaller than the second term ($2x^4$). The second term is added to the resulting polynomial and link

2.7. APPLICATION: POLYNOMIAL ARITHMETIC

j is advanced.

3. The powers of the first term ($3x^2$) and second term ($6x^2$) are the same. The sum of the coefficients of the first and second terms, namely 9, is added to the resulting polynomial. Both links i and j are advanced.
4. The powers of the first term ($-7x$) and the second term ($7x$) are the same. Since the sum of the coefficients of the first and second term is 0, the result term is not added to the resulting polynomial. Both links i and j are advanced. j link is now NULL.
5. The remaining part of the first polynomial (8) is added to the resulting polynomial.

2.7.2 Multiplying Two Polynomials

In order to multiply two polynomials, first each term of the first polynomial is multiplied with each term of the second polynomial, then the terms with the same powers are brought together and resulting polynomial is formed. For example, if we multiply the polynomial

$$x^2 + 2x + 1$$

with the polynomial

$$x^2 - 2x$$

- the first term of the first polynomial x^2 is multiplied with $x^2 - 2x$ to get $x^4 - 2x^3$,
- the second term of the first polynomial $2x$ is multiplied with $x^2 - 2x$ to get $2x^3 - 4x^2$,
- the third term of the first polynomial 1 is multiplied with $x^2 - 2x$ to get $x^2 - 2x$

Afterwards when we bring the terms with the same degree together, we get

$$x^4 - 2x^3 + 2x^3 - 4x^2 + x^2 - 2x = x^4 - 3x^2 - 2x$$

Table 2.31 shows the algorithm that finds the product of two polynomials in linked list representation. Since each term of the first polynomial is multiplied with each term of the second polynomial, with two nested loops, all

CHAPTER 2. LINKED LIST

Table 2.30: The algorithm that finds the product of two polynomials (C++)

```
1 LinkedList LinkedList::multiply(LinkedList p1, LinkedList p2){
2     Node *i, *j, *node, *previous;
3     LinkedList result;
4     int coefficient, degree;
5     result = LinkedList();
6     i = p1.head;
7     while (i != nullptr){
8         j = p2.head;
9         while (j != nullptr){
10             coefficient = i->data.coefficient * j->data.coefficient;
11             degree = i->data.degree + j->data.degree;
12             node = new Node(Term(coefficient, degree));
13             result.insertLast(node);
14             j = j->next;
15         }
16         i = i->next;
17     }
18     result.sort();
19     i = result.head;
20     previous = nullptr;
21     while (i != nullptr){
22         j = i->next;
23         while (j != nullptr && j->data.degree == i->data.degree){
24             i->data.coefficient += j->data.coefficient;
25             i->next = j->next;
26             j = j->next;
27         }
28         if (i->data.coefficient == 0)
29             previous->next = j;
30         else
31             previous = i;
32         i = j;
33     }
34     return result;
35 }
```

terms of two polynomials are processed. In the outer loop, *i* link represents the term of the first polynomial currently processed (Lines 6, 16), in the inner loop *j* link represents the term of the second polynomial currently processed (Lines 8, 14). For all *i, j* terms, the product of the coefficients of these terms will be the coefficient (Line 10), the sum of the degrees of these terms will

2.7. APPLICATION: POLYNOMIAL ARITHMETIC

Table 2.31: The algorithm that finds the product of two polynomials (Java)

```
1 static LinkedList multiply(LinkedList p1, LinkedList p2){
2     Node i, j, node, previous;
3     LinkedList result;
4     int coefficient, degree;
5     result = new LinkedList();
6     i = p1.head;
7     while (i != null){
8         j = p2.head;
9         while (j != null){
10             coefficient = i.data.coefficient * j.data.coefficient;
11             degree = i.data.degree + j.data.degree;
12             node = new Node(new Term(coefficient, degree));
13             result.insertLast(node);
14             j = j.next;
15         }
16         i = i.next;
17     }
18     result.sort();
19     i = result.head;
20     previous = null;
21     while (i != null){
22         j = i.next;
23         while (j != null && j.data.degree == i.data.degree){
24             i.data.coefficient += j.data.coefficient;
25             i.next = j.next;
26             j = j.next;
27         }
28         if (i.data.coefficient == 0)
29             previous.next = j;
30         else
31             previous = i;
32         i = j;
33     }
34     return result;
35 }
```

be the degree (Line 11) of the resulting term (Line 12). Each resulting term is added to the resulting linked list (Line 13).

After this step (i) the terms in the resulting polynomial are not sorted according to their degrees (ii) but also there can be more than one term for each degree (iii) and the sum of the coefficients of these common terms can be

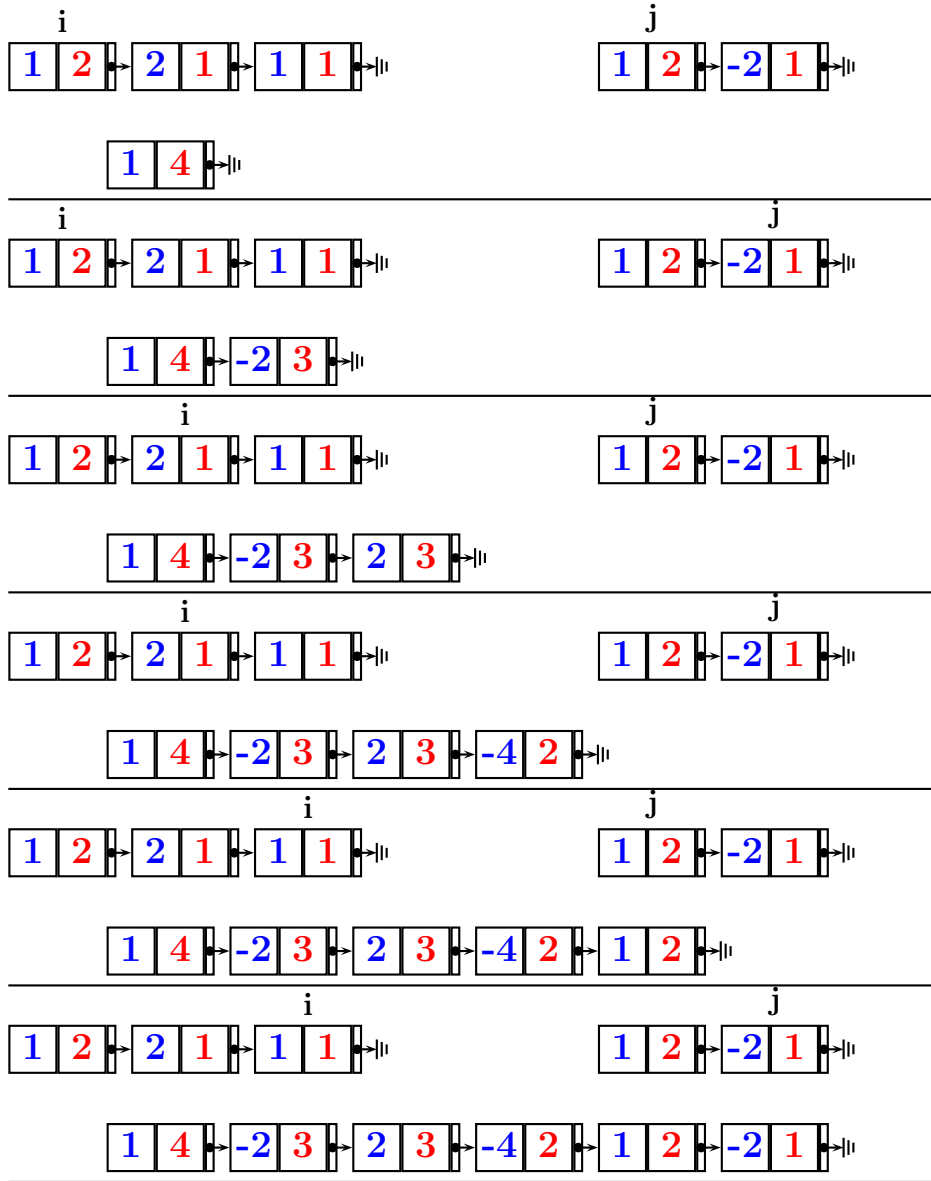


Figure 2.24: Multiplication of polynomials $x^2 + 2x + 1$ and $x^2 - 2x$ (1. Step)

0. In order to solve the first problem, the terms in the resulting polynomial

2.7. APPLICATION: POLYNOMIAL ARITHMETIC

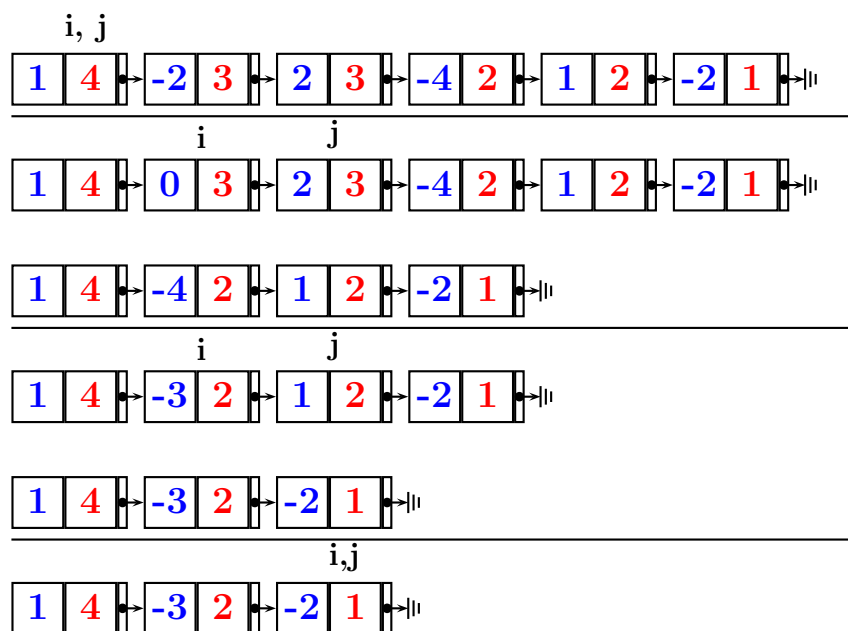


Figure 2.25: Multiplication of two polynomials $x^2 + 2x + 1$ and $x^2 - 2x$ (2. Step)

are sorted in increasing order according to their degrees (Line 18). Then to solve the second problem, the terms in the resulting polynomial are traversed one by one and the coefficients of the terms with the same degree are summed up. Here link i shows the first of the terms with common degree and the sum of coefficients of all terms (j) with that degree are accumulated in it (Line 24). In the last phase, link i is equalized to link j whereby the terms with common degree except i will be deleted (Line 32). Lastly, to solve the third problem, we look the sum of the coefficients with common degree (Line 28), if this sum is zero, term i is deleted from the resulting polynomial (Line 29).

Figures 2.24 and 2.25 show the steps of the multiplication of two polynomials $x^2 + 2x + 1$ and $x^2 - 2x$ using the algorithm in Table 2.31. In the first step (Figure 2.24),

- The first term of the first polynomial (x^2) and the first term of the second polynomial (x^2) is multiplied, the product (x^4) is added to the resulting polynomial.

CHAPTER 2. LINKED LIST

- The first term of the first polynomial (x^2) and the second term of the second polynomial ($-2x$) is multiplied, the product ($-2x^3$) is added to the resulting polynomial.
- The second term of the first polynomial ($2x$) and the first term of the second polynomial (x^2) is multiplied, the product ($2x^3$) is added to the resulting polynomial.
- The second term of the first polynomial ($2x$) and the second term of the second polynomial ($-2x$) is multiplied, the product ($-4x^2$) is added to the resulting polynomial.
- The third term of the first polynomial (1) and the first term of the second polynomial (x^2) is multiplied, the product (x^2) is added to the resulting polynomial.
- The third term of the first polynomial (1) and the second term of the second polynomial ($-2x$) is multiplied, the product ($-2x$) is added to the resulting polynomial.

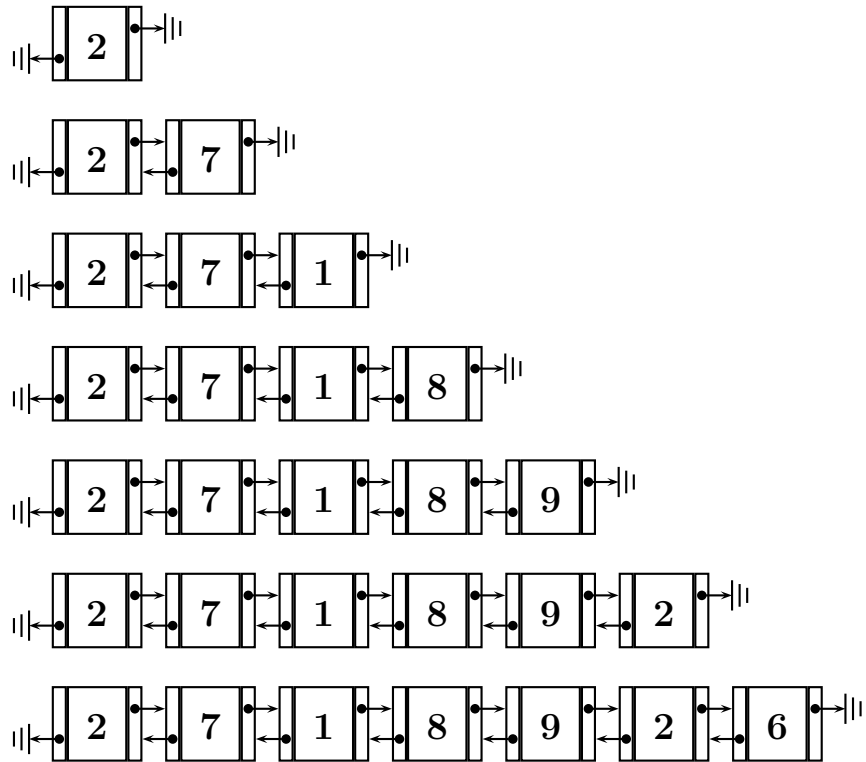
In the second step (Figure 2.25),

- The resulting polynomial has one term of type (x^4). i link is advanced, j link is advanced one after the link i.
- The coefficients of the terms of type (x^3) of the resulting polynomial are added to get $0x^3$ and link j is deleted. Since the sum of coefficients is 0 j link is deleted.
- The coefficients of the terms of type (x^2) of the resulting polynomial are added to get $-3x^2$ and link j is deleted.
- The resulting polynomial has one term of type (x).

2.8 Solved Exercises

1. Show the result of inserting 2, 7, 1, 8, 9, 2, 6 into an initially empty doubly linked list.

The structure after adding numbers 2, 7, 1, 8, 9, 2, 6 into an initially empty doubly linked list is shown below.



2. Write a function that will add a new node after the head of the link list.

```
void insertSecond(Node node)
```

The function that adds a new node after the head of the linked list is given below. If there are no elements in the list (Line 2) the new node is added to the head (Line 3), if there is only one element in the list (Line 5) the new node is added to the tail of the list (Line 6). If there are at least two elements in the list, the **next** link of the new node shows the second element (Line 8), the **next** link of the head node of the list shows the new node added (Line 9).

CHAPTER 2. LINKED LIST

<pre>1 void LinkedList :: insertSecond(Node* node){ 2 if (head == nullptr) 3 insertFirst (node); 4 else{ 5 if (first ->next == nullptr) 6 insertLast (node); 7 else{ 8 node->next = first->next; 9 first ->next = node; 10 } 11 } 12 }</pre>	<pre>void insertSecond(Node node){ if (head == null) insertFirst (node); else{ if (head.next == null) insertLast (node); else{ node.next = head.next; head.next = node; } } }</pre>
---	---

3. Write a function that will copy-and-paste of one part of a doubly linked list into another part of another doubly linked list. Assume that the starting point of copy, the ending point of the copy, and the starting point of the paste are given as parameters.

void copyPaste(DoubleNode first, DoubleNode last, DoubleNode copy)

The function that will copy-and-paste of one part of a doubly linked list into another part of another doubly linked list is given below. Here variables **head** and **tail** represent the starting and ending nodes of the copy, on the other hand variable **copy** represents the position of the paste in the second list. We use one link in each list to consolidate the copy-paste operation. The variable **copied** shows the link in the list to be copied, whereas the variable **pasted** shows the link in the list to be pasted. For each element to be copied (i) new memory is allocated for the new element (Line 5), (ii) the **previous** link of the new element points to the previously pasted element (Line 6), (iii) the **next** link of the previously pasted element points to the new element (Line 7). The links of the last element pasted will be done according to the next element after **copy** before pasted (Lines 11-12).

```
void DoublyLinkedList :: copyPaste(DoubleNode* first, DoubleNode* last, DoubleNode* copy){
  DoubleNode *copied = first, *pasted = copy;
  DoubleNode *next = copied->next, *newNode;
  while (copied != last->next){
    newNode = new DoubleNode(copied->data);
    newNode->previous = pasted;
    pasted->next = newNode;
    copied = copied->next;
  }
```

2.8. SOLVED EXERCISES

```
        pasted = newNode;
    }
    next->previous = pasted;
    pasted->next = next;
}
```

```
void copyPaste(DoubleNode first, DoubleNode last, DoubleNode copy){
    DoubleNode copied = first, pasted = copy;
    DoubleNode next = copied.next, newNode;
    while (copied != tail.next){
        newNode = new DoubleNode(copied.data);
        newNode.previous = pasted;
        pasted.next = newNode;
        copied = copied.next;
        pasted = newNode;
    }
    next.previous = pasted;
    pasted.next = next;
}
```

4. Write a function that will return all prime numbers until N (including N) as a linked list.

```
LinkedList primes(int N)
```

The function that produces all prime numbers between 1 until N and returning them in a singly linked list is given below. The function controls each number i starting from 2 till N for primality (Line 7). An integer is prime if it is not divisible by any integer except 1 and itself. Then what we have to do is to control if i is divisible by any integer from 2 till $N - 1$ (Line 10-13). If it is divisible by any of these numbers, then i can not be prime. Otherwise, i is added to the prime list (Lines 16-17).

CHAPTER 2. LINKED LIST

<pre> LinkedList primes(int N){ int i, j; bool isPrime; LinkedList primeList; Node* node; primeList = LinkedList(); for (i = 2; i <= N; i++){ isPrime = true; for (j = 2; j < N; j++){ if (i % j == 0){ isPrime = false; break; } } if (isPrime){ node = new Node(i); primeList.insertLast (node); } } return primeList; } </pre>	<pre> static LinkedList primes(int N){ int i, j; boolean isPrime; LinkedList primeList; Node node; primeList = new LinkedList(); for (i = 2; i <= N; i++){ isPrime = true; for (j = 2; j < N; j++){ if (i % j == 0){ isPrime = false; break; } } if (isPrime){ node = new Node(i); primeList.insertLast (node); } } return primeList; } </pre>
---	--

5. Given two sorted linked lists L_1 and L_2 , write a function to compute $L_1 \cup L_2$.

```
LinkedList union(LinkedList l1, LinkedList l2)
```

The function that finds the union of two sorted lists are given below. First the result list is created (Line 5). Then we traverse lists l1 and l2 using links tmp1 and tmp2 respectively (Line 6-8). While traversing the lists (i) if the current number of the first list is smaller than the current number of the second list, the link of the first list is advanced and the current number in the first list is added to the resulting list (Lines 10-11), (ii) if the current number of the second list is smaller than the current number of the first list, the link of the second list is advanced and the current number in the second list is added to the resulting list (Lines 14-15), (iii) if both numbers are equal a common element is found, links of both lists are advanced and this common element is added to the resulting list (Lines 17-19).

```

LinkedList union(LinkedList l1, LinkedList l2){
    int data;
    LinkedList result ;

```

2.8. SOLVED EXERCISES

```
Node *tmp1, *tmp2, *node;
result = LinkedList();
tmp1 = l1.head;
tmp2 = l2.head;
while (tmp1 != nullptr && tmp2 != nullptr){
    if (tmp1->data < tmp2->data){
        data = tmp1->data;
        tmp1 = tmp1->next;
    } else
        if (tmp1->data > tmp2->data){
            data = tmp2->data;
            tmp2 = tmp2->next;
        } else{
            data = tmp1->data;
            tmp1 = tmp1->next;
            tmp2 = tmp2->next;
        }
    node = new Node(data);
    result.insertLast(node);
}
return result;
```

```
static LinkedList union(LinkedList l1, LinkedList l2){
    int data;
    LinkedList result;
    Node tmp1, tmp2, node;
    result = new LinkedList();
    tmp1 = l1.head;
    tmp2 = l2.head;
    while (tmp1 != null && tmp2 != null){
        if (tmp1->data < tmp2->data){
            data = tmp1.data;
            tmp1 = tmp1.next;
        } else
            if (tmp1->data > tmp2->data){
                data = tmp2.data;
                tmp2 = tmp2.next;
            } else{
                data = tmp1.data;
                tmp1 = tmp1.next;
                tmp2 = tmp2.next;
            }
        node = new Node(data);
        result.insertLast(node);
    }
}
```

```
}  
    return result;  
}
```

2.9 Exercises

1. Write a function that will find the smallest number in a singly linked list.

`int smallest()`

2. Write a function to delete the second node from a singly linked list.

`void deleteSecond()`

3. Write a function that will add a new node before the last node of a singly linked list.

`void insertBeforeLast (Node newNode)`

4. Given a sorted linked list, write a function to add a new integer without destroying the sortedness property.

`void AddToSortedList(int x)`

5. Write a function to delete k 'th node from a singly linked list.

`void deleteKth(int k)`

6. Given node X , write a function to delete the node before it.

`void deleteBefore (DoubleNode X)`

7. Given node X , write a function to move that node n position forward. Assume that there are at least n nodes after node X .

`void move(Node X, int n)`

8. Write a function to find the middle node of a doubly linked list.

`DoubleNode middle()`

2.9. EXERCISES

9. Write a function to swap two nodes in a doubly linked list. You are not allowed to swap the contents of the nodes.

```
void swap(DoubleNode first, DoubleNode second)
```

10. Given an integer N , write a function which returns the prime factors of N as singly linked list.

```
LinkedList primeFactors(int N)
```

11. Write a function that will delete all nodes whose contents are divisible by N . The function will also return the deleted nodes as a new linked list.

```
LinkedList removeDivisibleByN(int N)
```

12. Write a function that will return the reverse of a singly link list.

```
LinkedList reverse()
```

13. Write a function that determines if a singly link list is palindrome, that is, it is equal its reverse.

```
boolean palindrom()
```

14. Write a function which doubles each node in a doubly linked list, that is, after each node inserts that node again.

```
void doubleList()
```

15. Write a function which removes the node before the last node of a single link list.

```
void removeBeforeLast()
```

2.10 Problems

1. Write a function that returns the Fibonacci numbers between A and B as a linked list. Fibonacci numbers are:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_2 &= 1 \\ &\dots \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

`LinkedList fibonacci(int A, int B)`

2. Suppose you are given a linked list of N integers that are sorted. Write an algorithm to remove duplicate elements from that sorted linked list.

`void removeDuplicates(LinkedList A)`

3. Write a function that will cut-and-paste of one part of a doubly linked list into another part of another doubly linked list. Assume that the starting point of the cut, the ending point of the copy, and the starting point of the paste are given as parameters.

`void cutPaste(DoubleNode first, DoubleNode last, DoubleNode paste)`

4. Given two sorted linked lists L_1 and L_2 , write a function to compute $L_1 \cap L_2$.

`LinkedList intersection (LinkedList l1, LinkedList l2)`

5. Write a function that deletes all nodes having value X from a singly linked list.

`void deleteAll (int X)`

6. Write a function that checks if the original list contains the elements of the second list in the same order.

`boolean subList(LinkedList sub)`

2.10. PROBLEMS

7. Suppose you are given a linked list of N integers that are sorted. Write an algorithm to remove single elements from that sorted linked list.

`void removeSingles(LinkedList A)`

8. Write a function that will delete the odd indexed elements from a singly linked list. The function will also return the deleted nodes as a new linked list.

`LinkedList oddIndexedElements()`

9. Write a function that will delete even indexed elements from a singly linked list.

`void deleteEvenIndexed()`

10. Write the method which prints the contents of the linked list in reverse order. You are not allowed to use any array variable.

`void printReverse()`

11. Write the method which add newnode after each node in the singly linked list.

`void addAfterEachNode(Node newNode)`

12. Write the method which prints the contents of the odd indexed nodes (1, 3, ...) in the linked list.

`void printOddNodes()`

13. Write a function that will delete all **prime** nodes that is their data field is prime such as 2, 3, 5, 7, etc.

`void deletePrimes()`

14. Write a linear time method to delete the nodes indexed between p and q (including p 'th and q 'th items) from a singly linked list.

`void deleteBetween(int p, int q)`

15. Write the algorithm Sieve of Eratosthenes to extract prime numbers using singly linked list. The algorithm works as follows:

- The user enters a number N .
- Put all numbers starting from 2 to N in a linked list.

CHAPTER 2. LINKED LIST

- While the linked list contains numbers
 - Remove the first element p from the linked list. Print it (It is prime).
 - Remove all elements from the linked list which are divisible by p . Do not print them.
- 16. Write a method for doubly linked lists, which returns a new doubly linked list with even indexed nodes (2, 4, ...) from the original list. Your linked list should contain new nodes, not the same nodes in the original linked list. The first node has index 1. You are not allowed to use any linked list methods, just attributes, constructors, getters and setters.

`DoublyLinkedList getEvenOnes()`

17. Write a method which returns true if the singly linked list only contains duplicates, that is, every datum (number) occurs only twice. Important warning, the duplicate elements may not be adjacent. You are not allowed to use any singly linked list methods, just attributes, constructors, getters and setters.

`boolean containsOnlyDuplicates()`

18. Write the algorithm

`LinkedList primeDivisors(int N)`

in the **LinkedList** class which works as follows:

- Creates a temporary linked list primes, which stores the prime numbers until N.
- Using primes returns all prime divisors (with repeating) of N.

Let say $N = 200$, the function will return 2, 2, 2, 5, 5. You are not allowed to use any array in the function.

19. Write the algorithm

`Node* lastOneWins(int k)`

in the **LinkedList** class which works as follows:

- Delete every k 'th element from the list.

- When you get the end of the list, return to the first element, as if the list is circular.
- Return the remaining node.

Let say the list is 1 2 3 4 5 6, and $k = 2$, then 2, 4, 6, 3, 1 will be deleted, 5 remains.

20. Write the following algorithm to sort the elements in the doubly linked list and return a new doubly linked list. The algorithm is as follows:
- Find the largest number N in the linked list.
 - For each number i between 1 and N :
 - Count the number of times linked list has i .
 - Insert that many times i to the new linked list.

The elements before sorting:

5 2 3 4 2 3 4 3 2

The elements after sorting:

2 2 2 3 3 3 4 4 5

`DoublyLinkedList sortElements()`

You are not allowed to use any linked list methods. You are allowed to use attributes, constructors, getters and setters. Write the method in the `DoublyLinkedList` class.

21. Write the method

`boolean evenOddSorted()`

which returns true if the singly linked list odd indexed elements are sorted increasing order and even indexed elements are sorted in decreasing order. The first node has index 1. You are not allowed to use any singly linked list methods. You are allowed to use attributes, constructors, getters and setters. Write the method in the `LinkedList` class.

Sorted:

2 10 3 7 6 4 9 2 13 0 20

22. Write the method

`LinkedList intersec(LinkedList list1 , LinkedList list2)`

CHAPTER 2. LINKED LIST

to find the intersection of the elements in two sorted linked lists and return a new linked list. Implement the following algorithm:

1. At the beginning of the algorithm, let say we have two nodes p1 and p2, showing the head nodes of the first and second lists respectively.
2. Compare the contents of the nodes p1 and p2;
 - If $p1.data < p2.data$, advance p1 pointer to show next node in its list.
 - If $p1.data > p2.data$, advance p2 pointer to show next node in its list.
 - If $p1.data = p2.data$, put a new node with content of p1 and advance both pointers p1 and p2 in their respective lists.
3. Continue with step 2 until one of the p1 or p2 is null.

You are not allowed to use any linked list methods. You are only allowed to use attributes, constructors, getters and setters.

Contents of the first list

1 3 5 7 11 12

Contents of the second list

1 2 6 7 9 11

Contents of the results

1 7 11

23. Write the method

boolean isPalindrom()

which returns true if the doubly linked list is palindrom. Implement the following algorithm:

- At the beginning of the algorithm, we have two pointers p1 and p2, which shows the beginning and the end of the list respectively.
- Compare the contents of the pointers, if they are different, return false, otherwise advance the pointers p1 to next, p2 to previous.
- The algorithm finishes either $p1 = p2$ or $p1.next = p2$, in which case the method returns true.

2.10. PROBLEMS

You are not allowed to use any doubly linked list methods. You are allowed to use attributes, constructors, getters and setters.

24. Write the method

```
void remove(const LinkedList& list2 )
```

which removes the nodes that appear in the list2 from the original list. You are not allowed to use any methods from the LinkedList class. You can assume both the original list and list2 do not contain duplicate elements. Do not modify list2. Your method should run in $\mathcal{O}(N^2)$ time.

25. Write the method

```
void deleteEven()
```

which removes the nodes with even values in the original list. Your method should run in $\mathcal{O}(N)$ time.

26. Write the method

```
LinkedList getIndex(LinkedList list )
```

which returns the elements at positions list[1], list[2], list[3], etc. list[1] is the first element in the list, list[2] is the second element in the list etc. You are not allowed to use any linked list methods. You are only allowed to use attributes, constructors, getters and setters. Assume that list is sorted. Your algorithm should run in $\mathcal{O}(N)$ time. Your linked list should contain new nodes, not the same nodes in the original linked list.

Contents of the original list

3 1 7 5 11 14 2 8 16

Contents of the list

1 4 6 9

Contents of the results

3 5 14 16

27. Write the method

```
void removeKthBeforeLast(int K)
```

which removes the K'th element from the end of the double linked list. If $K = 1$, last element will be deleted. If $K = N$, first element will be

CHAPTER 2. LINKED LIST

deleted. First count the number of elements in the list, i.e. N , then handle special cases $K = 1$, and $K = N$, then do the rest. You are not allowed to use any doubly linked list methods. You are allowed to use attributes, constructors, getters and setters.

28. Write the static method in **LinkedList** class

```
LinkedList difference (LinkedList list1 , LinkedList list2 )
```

to find the difference of the elements in two sorted linked lists and return a new linked list. The resulting list should contain those elements that are in list1 but not in list2. Do not modify linked lists list1 and list2. Your method should run in $\mathcal{O}(N)$ time. Nodes in the resulting list should be new. You can not use any linked list methods except getters and setters.

29. Write a method which returns true if the single linked list only contains triplicates, that is, every datum (number) occurs only three times. Important warning, the triplicate elements may not be adjacent. Your method should have a time complexity of $\mathcal{O}(N^2)$. You are not allowed to use any single linked list methods, just attributes, constructors, getters and setters.

```
boolean containsOnlyTriplicates ()
```

30. Write the method

```
void reverse ()
```

for reversing a doubly linked list. Your method should have a time complexity of $\mathcal{O}(N)$. You are not allowed to use any extra data structures. You are not allowed to use any linked list methods, just attributes, constructors, getters and setters.

2.10. PROBLEMS

Stack is a list data structure consisting of many elements. There are two types of operations defined for the elements of the stack: Adding an element to the stack (push) and removing an element from the stack (pop). In a stack, to be popped element is always the last pushed element. Also when an element is pushed on to the stack, it is placed on top of the stack (at the end of the list). The name stack comes from this idea. When you add objects on the stack, you place on it randomly. Similarly when we want to remove an object from a stack, what we can only do is removing the topmost object. Therefore the definition of stack shows us that the stack data structure is a last in first out data structure.

3.1 Array Implementation

One of the easiest way of representing a stack in a programming language is defining it with an array. The size of the array determines the maximum number of elements that can be stored in the stack. Since both operations defined on the stack use the last element, we need a link pointing to the last element of the array. Given stack S , the last element is represented with the field **top**. Given an array S , $S[0]$ represents the first element of the stack, $S[\text{top}]$ represents the last element of the stack. If top is -1 then the stack is empty, if top is $N - 1$ then the stack is full (where N represents the size of the array).

The definition of an element in an array implemented stack is given in

3.1. ARRAY IMPLEMENTATION

Table 3.1: The definition of an element in an array implemented stack

1	class Element{	1	public class Element{
2	private :	2	int data;
3	int data;	3	public Element(int data){
4	public :	4	this .data = data;
5	Element(int data);	5	}
6	}	6	}
7			
8	Element::Element(int data){		
9	this —>data = data;		
10	}		

Table 3.1. The contents of these elements can be modified by adding new fields to the structure **element**.

Table 3.2 shows the definition of a stack containing integers. The array named **array** is defined inside the structure **stack**, the position of the topmost element in this stack is represented by the field **top**.

The function *isFull* checks whether an array-implemented stack is full or not. The function returns true is the stack is full, false otherwise. Since the stack data structure is defined with an array of size N , if the field **top**, representing the last element of the stack, is $N - 1$ the stack is full and one can not add a new element on top of the stack.

The function *isEmpty* checks whether an array-implemented stack is empty or not. The function returns true if the stack is empty, false otherwise. If there are no elements in the stack, the field **top** is -1 and the stack is empty.

Figure 3.1 shows an example stack of size 8, which contains 6 elements. The elements in the stack from bottom to top are 8, 10, 4, 12, 8, and 6 respectively. As can be seen from the Figure, since there are six elements in the stack, the value of **top** is 5.

3.1.1 Insertion

Table 3.3 shows the algorithm that pushes a new element on top of stack **c**. When we push an element on top of the stack , we only need to increase the field **top** by 1 (Line 3) and place the new element on this new position (Line 4). If the stack is full before this push operation, we can not push. (Line 2).

Figure 3.2 shows the case where we add a new element with data 9 to a

CHAPTER 3. STACK

Table 3.2: Definition of a stack, which contains integers and implemented using array

<pre>1 class Stack{ 2 private: 3 Element* array; 4 int top; 5 int N; 6 public: 7 Stack(int N); 8 ~Stack(); 9 Element top(); 10 bool isFull (); 11 bool isEmpty(); 12 } 13 14 Stack::Stack(int N){ 15 array = new Element[N]; 16 this->N = N; 17 top = -1; 18 } 19 Stack::~~Stack(){ 20 delete [] array; 21 } 22 Element Stack::top(){ 23 return array[top]; 24 } 25 bool Stack::isFull (){ 26 if (top == N - 1) 27 return true; 28 else 29 return false ; 30 } 31 bool Stack::isEmpty(){ 32 if (top == - 1) 33 return true; 34 else 35 return false ; 36 }</pre>	<pre>1 public class Stack{ 2 Element array[]; 3 int top; 4 int N; 5 public Stack(int N){ 6 array = new Element[N]; 7 this.N = N; 8 top = -1; 9 } 10 Element top(){ 11 return array[top]; 12 } 13 boolean isFull(){ 14 if (top == N - 1) 15 return true; 16 else 17 return false; 18 } 19 boolean isEmpty(){ 20 if (top == - 1) 21 return true; 22 else 23 return false; 24 } 25 }</pre>
---	--

stack originally containing 6 elements. From now on the field **top** does not point to 8 but the new pushed element 9.

3.1. ARRAY IMPLEMENTATION

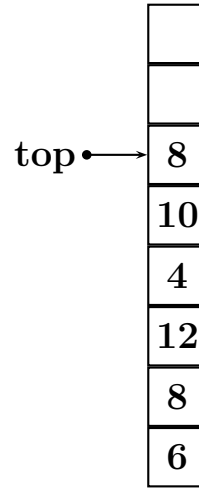


Figure 3.1: The array representation of a stack of size 8, which contains 6 elements

Table 3.3: The algorithm that adds a new element on top of the stack

1	void Stack::push(Element element){	1	void push(Element element){
2	if (! isFull ()) {	2	if (! isFull ()) {
3	top++;	3	top++;
4	array[top] = element;	4	array[top] = element;
5	}	5	}
6	}	6	}

3.1.2 Deletion

Table 3.4 shows the algorithm that removes the topmost element from stack `c`. When we remove an element from the stack (the function also returns that removed element), we need to be careful if the stack was empty or not (Line 2). If the stack is not empty, the topmost element of the stack is returned (Line 4) and the field `top` is decreased by 1 (Line 3). If the stack is empty, the function will return `NULL` (Line 6).

Figure 3.3 shows the case when we remove an element from an example stack originally containing 6 elements. From now on, the field `top` does not point to 8 but the element one below namely 10.

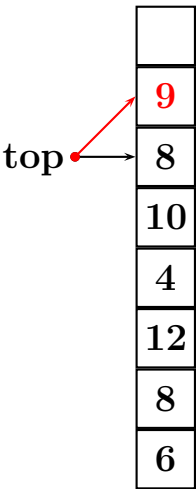




Figure 3.2: Adding an element on top of the stack

Table 3.4: The algorithm that removes an element from the stack and returns it

1	Element Stack::pop(){	1	Element pop(){
2	if (!isEmpty()){	2	if (!isEmpty()){
3	top--;	3	top--;
4	return array[top+1];	4	return array[top+1];
5	}	5	}
6	}	6	return null;
		7	}

Stack Operations (Array)	
	Insertion: $\mathcal{O}(1)$
	Deletion: $\mathcal{O}(1)$

3.2. LINKED LIST IMPLEMENTATION

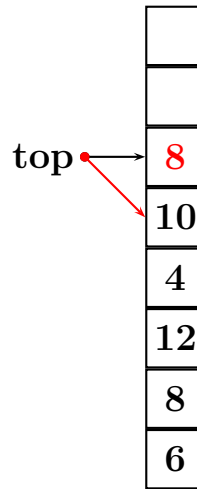


Figure 3.3: Removing an element from the stack

3.2 Linked List Implementation

Yet another possible representation of a stack is defining it with linked lists. Since both operations defined on the stack are done on the last element of the stack, it will be enough to store a link which points to the last element of the stack.

Table 3.5 shows the linked list definition of the stack which contains integers. The contents of the elements in the stack can be modified by adding new fields to the **node** structure. The topmost element of the stack is represented using the field **top**. Contrary to the implementation using arrays, in the linked list implementation there is no upper limit on the number of elements in the stack. As long as there is memory in the computer, one can add new elements onto the stack.

The function *isEmpty* checks if the stack is empty or not. If the stack is empty, the function returns true, otherwise returns false. If there are no elements in the stack, the **top** field is NULL.

Figure 3.4 shows an example linked list implemented stack, which contains 6 elements. As can be seen from the Figure, apart from the array implementation, in the linked list implementation there is no need for extra space. The field **top** points to the element with data 8.

CHAPTER 3. STACK

Table 3.5: Definition of a stack, which contains integers and implemented using linked list

<pre>1 class Stack{ 2 private: 3 Node* top; 4 public: 5 Stack(); 6 ~Stack(); 7 bool isEmpty(); 8 } 9 10 Stack::Stack(){ 11 top = nullptr; 12 } 13 Stack::~~Stack(){ 14 Node* tmp; 15 tmp = top; 16 while (tmp != null){ 17 Node* next = tmp->next; 18 delete tmp; 19 tmp = next; 20 } 21 } 22 boolean Stack::isEmpty(){ 23 if (top == nullptr) 24 return true; 25 else 26 return false; 27 }</pre>	<pre>1 public class Stack{ 2 Node top; 3 public Stack(){ 4 top = null; 5 } 6 boolean isEmpty(){ 7 if (top == null) 8 return true; 9 else 10 return false; 11 } 12 }</pre>
---	--

3.2.1 Insertion

Table 3.6 shows the algorithm that adds a new element onto a given stack *c*. When we add an element onto the stack we need to change the field **top** and modify it to point to the new added element (Line 3). Also the **next** link of the new added element will point to the old **top** field (Line 2).

Figure 3.5 shows the case where a new element with data 9 is added onto a linked list implemented stack with 6 elements. From now on, the **top** field does not point to 8 but the new added element 9.

3.2. LINKED LIST IMPLEMENTATION

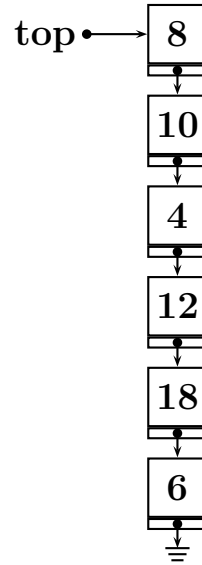


Figure 3.4: The linked list representation of a stack, which contains 6 elements

Table 3.6: The algorithm that adds a new element onto the stack

1	void Stack::push(Node* newNode){	1	void push(Node newNode){
2	newNode->next = top;	2	newNode.next = top;
3	top = newNode;	3	top = newNode;
4	}	4	}

3.2.2 Deletion

Table 3.7 shows the algorithm that removes an element from a given stack *c*. When we remove an element from the stack (the function returns the removed element) we need to look out if the stack is empty or not (Line 3). If the stack is not empty, the last element of the stack is returned (Line 5) and the **top** variable of the stack points to the next element (Line 4).

Figure 3.6 shows the case when an element is removed from a linked list implemented stack of 6 elements. From now on, the **top** field does not point to 8, but one below 8, namely 10.

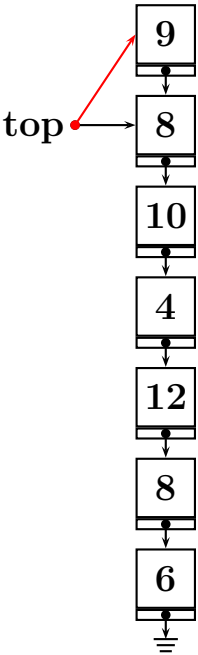




Figure 3.5: Adding an element onto the stack

Table 3.7: The algorithm that removes an element and returns it

1	Node* Stack::pop(){	1	Node pop(){
2	Node* e = top;	2	Node e = top;
3	if (!isEmpty())	3	if (!isEmpty())
4	top = top->next;	4	top = top.next;
5	return e;	5	return e;
6	}	6	}

Stack Operations (Linked List)	
	Insertion: $\mathcal{O}(1)$
	Deletion: $\mathcal{O}(1)$

3.3. APPLICATION: EVALUATING MATHEMATICAL EXPRESSIONS

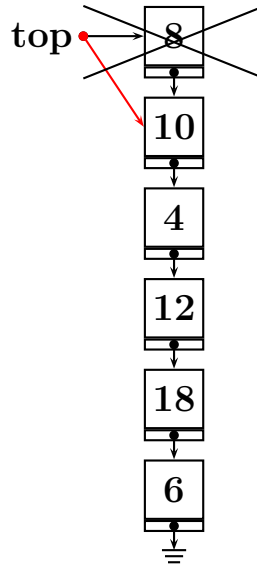


Figure 3.6: Removing an element from the stack

3.3 Application: Evaluating Mathematical Expressions

In high level programming language, the statements are of two types. Assignment statements are the first type where you assign a given mathematical expression to a variable. Conditional statements are the second type where you check the truth value of a conditional mathematical expression and according to the result jump to some part of the program. An example assignment statement is

```
x = a / b - c + d * e - a * c;
```

whereas an example conditional statement is

```
if (3 * x - 5 > 4 * y * y)
```

The common part of both type of statements is that they both contain mathematical expressions. For example, when a compiler compiles the first statement, it will first evaluate the value of the expression $a/b - c + d * e - a * c$,

CHAPTER 3. STACK

and will assign that value to the variable x . For the second statement, the compiler will control truth value of the expression $3 * x - 5 > 4 * y * y$ for the values of the variables x and y at the execution time of the statement. Therefore, compiler and programs that process programs must be able to evaluate of any mathematical expression. In this section, we will discuss how we can use stacks to evaluate mathematical expressions.

3.3.1 Mathematical Expressions

A mathematical expression basically consists of operands and operators. In the mathematical expression $a/b - c + d * e - a * c$, there are five operands, namely a , b , c , d , and e . In a mathematical expression there can be many operands. On the other hand, operators are the connectors of operands. For example, $/$, $-$, $+$, $*$ are the operators in the mathematical expression $a/b - c + d * e - a * c$. The operators can be the basic four operations that we learned in the primary school, or they can also be more complex operations such as $\%$. There are also comparison operators such as $<$, $<=$, $>$, $>=$, $==$. Another class of operators are logical operators such as $\&\&$ (and) and $||$ (or).

Whatever the operators and operand are in a given mathematical expression, in order to evaluate the expression we need to know the order of evaluation of operators. For example, in the previous expression $a/b - c + d * e - a * c$, if there is no precedence relation between the four mathematical operators $+$, $-$, $*$, and $/$, then to evaluate the expression we first divide a to b , then subtract c from the result, then add d to the result etc. But as we learned in the middle school, if $*$ and $/$ have higher precedence than $+$ and $-$, to evaluate the expression first a is divided by b , then d is multiplied with e , then a is multiplied with c and afterwards addition and subtraction operation are done between the results of these expressions. Therefore, in order to evaluate a mathematical expression, we need to know the precedence relationship between the operators existed in that expression. The precedence relations between the operators in C and Java is given below:

1. $-$ (unary minus)
2. $*$ / $\%$
3. $+$ $-$
4. $\&\&$ $||$

3.3. APPLICATION: EVALUATING MATHEMATICAL EXPRESSIONS

Table 3.8: Definition of a token in a mathematical expression

1	class Token{	1	public class Token{
2	private :	2	int type;
3	int type;	3	int operand;
4	int operand;	4	char operator;
5	char operator;	5	int precedence;
6	int precedence;	6	public Token(int operand){
7	public :	7	this .type = 0;
8	Token(int operand);	8	this .operand = operand;
9	Token(char operator);	9	}
10	}	10	public Token(char operator){
11		11	this .type = 1;
12	Token::Token(int operand){	12	this .operator = operator;
13	this —>type = 0;	13	switch (operator){
14	this —>operand = operand;	14	case '(':precedence = 0;
15	}	15	break ;
16	Token::Token(char operator){	16	case '+':
17	this —>type = 1;	17	case '-':precedence = 1;
18	this —>operator = operator;	18	break ;
19	switch (operator){	19	case '*':
20	case '(':precedence = 0;	20	case '/':precedence = 2;
21	break ;	21	break ;
22	case '+':	22	case ')':precedence = 3;
23	case '-':precedence = 1;	23	break ;
24	break ;	24	}
25	case '*':	25	}
26	case '/':precedence = 2;	26	}
27	break ;		
28	case ')':precedence = 3;		
29	break ;		
30	}		
31	}		

In order to express a mathematical expression more easily, we will construct the class **Token**. Table 3.8 shows the **Token** class. **type** represents if the element in the mathematical expression is an operator or an operand. If **type** is 0, the element is an operand, otherwise it is an operator. If the element is an operand, the value is stored in the field **operand**, whereas if the element is an operator, the value is stored in the field **operator**. If the element is an operator the precedence value is also stored in the field **precedence**. Since the precedences of + and - are smallest, their precedences are 1, since the

CHAPTER 3. STACK

precedences of $*$ and $/$ are larger than $+$ and $-$, their precedences are 2. The precedence of opening parenthesis is the smallest and is 0, whereas the closing parenthesis has the largest precedence and is 3.

3.3.2 Postfix Representation

How can we evaluate a mathematical expression? The computer scientists has found the solution by introducing another type of representation of expressions: Postfix representation. The representation we used until now is called infix representation. The meaning of infix is the operators are between (or inside) the operands. But we can present the same expression with another representation, namely postfix representation. In this type of representation, the operators come after (post) the operands. For example, the postfix representation of the infix expression $a * b / c$ is like $ab * c /$. As can be seen, operator $*$ is placed between the operands a and b with which it will multiply. On the other hand, operator $/$ separates the expression $a * b$ from c , therefore in the postfix representation is given after those two operands at the end. So, the postfix representation of the infix expression $a / b - c + d * e - a * c$ is $ab / c - de * + ac * -$.

The main difference between the infix representation and the postfix representation is: If there are no parenthesis in an infix expression, the expression may be ambiguous and the ambiguity is resolved using precedences, but the postfix expression is always unambiguous. For example, in the infix expression $a * b + c$, we need to know which of the addition and multiplication operations will be done before, whereas we do not need such information while evaluating the postfix version of the same expression $(ab * c +)$. What needs to be done is reading expression from left to right and evaluating expression. For this reason the postfix representation does not include parentheses.

3.3.3 Evaluation an Expression Given Its Postfix Form

The evaluation of the postfix expression $ab / c - de * + ac * -$ is show below.

3.3. APPLICATION: EVALUATING MATHEMATICAL EXPRESSIONS

$T_1 = a / b$	$T_1 \text{ c - d e } * + a \text{ c } * -$
$T_2 = T_1 - c$	$T_2 \text{ d e } * + a \text{ c } * -$
$T_3 = d * e$	$T_2 \text{ T}_3 + a \text{ c } * -$
$T_4 = T_2 + T_3$	$T_4 \text{ a c } * -$
$T_5 = a * c$	$T_4 \text{ T}_5 -$
$T_6 = T_4 - T_5$	T_6

The main theme recurring is, two operands and a single operator are taken from the expression and the operation is carried out. Then the result is placed in the expression as if it was an operator. When we scan the whole expression, the result remained in the expression is the result of the overall expression. Therefore, while reading the expression, when we come across an operand we push it onto the stack and when we come across an operator, we pop two operands from the stack, carry out the operation and push the result back onto the stack, we will cover our purpose.

Table 3.10 shows the algorithm that evaluates an expression given its postfix representation. The algorithm takes the expression as a parameter of element array. First the stack that will be used in the algorithm is defined (Line 4). Afterwards each element of the element array is processed one after another (Line 5). If the element currently processed is an operand (Line 7), it is pushed onto the stack (Line 8); if it is an operator (Line 9), two operands are popped from the stack (Lines 10-11) carried out the operations with these operands and the operator (Line 12) and the result of this operation is pushed again onto the stack (Line 14). In order to carry out four arithmetic operations, we use the function `eval` and this function takes two numbers and the character representing this operator as parameters. The function carry out the mathematical operation taken as a parameter on two numbers which are also taken as parameters (Lines 22-31) and returns the result encapsulated in an element object (Line 32).

We also assume that all operators in the expression take two operands. If the operator works with more or less than two operands, one needs to modify the lines `e1 = (Token) c.pop().data; e2 = (Token) c.pop().data` such that the number of lines correspond the number of operands the operator takes. For example, the unary minus operator `-` requires only one operand, therefore it is enough to pop one element from the stack. On the other hand, for the operators such as `+`, `-` which require two operands, we must pop two elements. When we finish to process the element array, there is only one element in the stack

CHAPTER 3. STACK

Table 3.9: The algorithm that evaluates an expression given its postfix representation (C++)

```
1  int evaluate(Element* expr, int length){
2      int i;
3      Token e, e1, e2;
4      Element s;
5      Stack c = Stack(100);
6      for (i = 0; i < length; i++){
7          e = expr[i].data;
8          if (e.type == 0){
9              c.push(expr[i]);
10         }else{
11             e2 = c.pop().data;
12             e1 = c.pop().data;
13             s = Element(eval(e.operator, e1.operand, e2.operand));
14             c.push(s);
15         }
16     }
17     e = c.pop().data;
18     return e.operand;
19 }
20 Token eval(char ch, int e1, int e2){
21     int result = 0;
22     switch (ch){
23         case '+':result = e1 + e2;
24         break;
25         case '-':result = e1 - e2;
26         break;
27         case '*':result = e1 * e2;
28         break;
29         case '/':result = e1 / e2;
30         break;
31     }
32     return Token(result);
33 }
```

and that element represents the value of the expression. In order to get the value of the expression, we pop that element from the stack (Line 17) and the data of that element is returned (Line 18).

Figure 3.7 shows the evaluation of the postfix expression $ab/c - de * + a*$ using a stack. In each of the 11 steps shown in the Figure; the unread part

3.3. APPLICATION: EVALUATING MATHEMATICAL EXPRESSIONS

Table 3.10: The algorithm that evaluates an expression given its postfix representation (Java)

```
1 static int evaluate(Element[] expr){
2     int i;
3     Token e, e1, e2;
4     Element s;
5     Stack c = new Stack(100);
6     for (i = 0; i < expr.length; i++){
7         e = expr[i].data;
8         if (e.type == 0){
9             c.push(expr[i]);
10        }else{
11            e2 = (Token) c.pop().data;
12            e1 = (Token) c.pop().data;
13            s = new Element(eval(e.operator, e1.operand, e2.operand));
14            c.push(s);
15        }
16    }
17    e = (Token) c.pop().data;
18    return e.operand;
19 }
20 static Token eval(char ch, int e1, int e2){
21     int result = 0;
22     switch (ch){
23         case '+':result = e1 + e2;
24             break;
25         case '-':result = e1 - e2;
26             break;
27         case '*':result = e1 * e2;
28             break;
29         case '/':result = e1 / e2;
30             break;
31     }
32     return new Token(result);
33 }
```

of the expression is shown below, current state of the stack is shown in the middle, and if we are currently carrying out an operation, that corresponding operator is shown above.

1. a is an operand. It is pushed onto the stack.

CHAPTER 3. STACK

\boxed{a} $b / c - d e^* + a^*$	$\begin{array}{c} \boxed{b} \\ \boxed{a} \end{array}$ $/ c - d e^* + a^*$	$T_1 = a/b$ $\boxed{T_1}$ $c - d e^* + a^*$	$\begin{array}{c} \boxed{c} \\ \boxed{T_1} \end{array}$ $- d e^* + a^*$
$T_2 = T_1 - c$ $\boxed{T_2}$ $d e^* + a^*$	$\begin{array}{c} \boxed{d} \\ \boxed{T_2} \end{array}$ $e^* + a^*$	$\begin{array}{c} \boxed{e} \\ \boxed{d} \\ \boxed{T_2} \end{array}$ $* + a^*$	$T_3 = d * e$ $\begin{array}{c} \boxed{T_3} \\ \boxed{T_2} \end{array}$ $+ a^*$
$T_4 = T_2 + T_3$ $\boxed{T_4}$ a^*	$\begin{array}{c} \boxed{a} \\ \boxed{T_4} \end{array}$ $*$	$T_5 = a * T_4$ $\boxed{T_5}$	

Figure 3.7: Evaluation of the postfix expression $ab/c - de^* + a^*$ using a stack

2. b is an operand. It is pushed onto the stack.
3. $/$ is an operator. We pop two elements from the stack (a and b), we carry out the operation $/$ using these two elements, and the result (T_1) is pushed onto the stack.
4. c is an operand. It is pushed onto the stack.
5. $-$ is an operator. We pop two elements from the stack (T_1 and c), we carry out the operation $-$ using these two elements, and the result (T_2) is pushed onto the stack.
6. d is an operand. It is pushed onto the stack.
7. e is an operand. It is pushed onto the stack.

3.3. APPLICATION: EVALUATING MATHEMATICAL EXPRESSIONS

8. $*$ is an operator. We pop two elements from the stack (d and e), we carry out the operation $*$ using these two elements, and the result (T_3) is pushed onto the stack.
9. $+$ is an operator. We pop two elements from the stack (T_2 and T_3), we carry out the operation $+$ using these two elements, and the result (T_4) is pushed onto the stack.
10. a is an operand. It is pushed onto the stack.
11. $*$ is an operator. We pop two elements from the stack (a and T_4), we carry out the operation $*$ using these two elements, and the result (T_5) is pushed onto the stack.

3.3.4 Conversion from Infix to Postfix

Until now we have seen how we can evaluate a mathematical expression given in postfix representation. In the usual cases, we are given the infix representation but not postfix representation of the expression. We also know, in the infix representation, the operators are inside the operands. In order to finish the overall problem, that is given an infix representation of an expression how to evaluate it, we need to find the postfix representation of an infix representation. To do that, we will use the similarity between the infix and postfix representations of the expression. In both representations, the order of the operands stay the same, whereas the order of the operators change. Therefore, in this case, we will push the operators onto the stack but not the operands, and when the time comes we will pop and write them.

Table 3.12 shows the algorithm that finds the postfix representation of an infix expression. The algorithm takes an array of elements as the expression for the input. Each element of an expression is an operator or an operand. At a time, let the element currently processed be e (Line 6). If e is an operand e has nothing to do with the stack (Line 7). It is written directly on the screen (Line 8). If e is an operator, it can be one of the two parentheses or another type of operator:

`ch=')` The closing parenthesis has the highest precedence (Line 13). We pop elements from the stack (Line 17) until we see a corresponding opening parenthesis (Line 15) and we write each of them on the screen (Line 16).

CHAPTER 3. STACK

Table 3.11: The algorithm that finds the postfix representation of an infix expression (C++)

```
1 void infixToPostfix(Element* expression, int length){
2     int i;
3     Token e, e1;
4     Stack c = Stack(100);
5     for (i = 0; i < length; i++){
6         e = expression[i].data;
7         if (e.type == 0)
8             cout << e.operand;
9         else
10            if (e.operator == '(')
11                c.push(expression[i]);
12            else
13                if (e.operator == ')'){
14                    e1 = c.pop().data;
15                    while (e1.operator != '('){
16                        cout << e1.operator;
17                        e1 = c.pop().data;
18                    }
19                }
20            else{
21                while (e.precedence <= c.top().precedence){
22                    e1 = c.pop().data;
23                    cout << e1.operator;
24                }
25                c.push(expression[i]);
26            }
27    }
28    while (!c.isEmpty()){
29        e1 = c.pop().data;
30        cout << e1.operator;
31    }
32 }
```

ch='(' The opening parenthesis has the lowest precedence (Line 10). We push it onto the stack (Line 11).

ch = operator If the precedence of the operator ch is larger than the precedence of the topmost operator in the stack, ch is pushed onto the stack (Line 26). Otherwise, until this condition is satisfied, we pop elements from the stack (Line 23) and write them on the screen (Line 24) and when

3.3. APPLICATION: EVALUATING MATHEMATICAL EXPRESSIONS

Table 3.12: The algorithm that finds the postfix representation of an infix expression (Java)

```
1 static void infixToPostfix(Element[] expression){
2     int i;
3     Token e, e1;
4     Stack c = new Stack(100);
5     for (i = 0; i < expression.length; i++){
6         e = (Token) expression[i].data;
7         if (e.type == 0)
8             System.out.print(e.operand);
9         else
10            if (e.operator == '(')
11                c.push(expression[i]);
12            else
13                if (e.operator == ')'){
14                    e1 = (Token) c.pop().data;
15                    while (e1.operator != '('){
16                        System.out.print(e1.operator);
17                        e1 = (Token) c.pop().data;
18                    }
19                }
20            else{
21                while (e.precedence <= c.top().precedence){
22                    e1 = (Token) c.pop().data;
23                    System.out.print(e1.operator);
24                }
25                c.push(expression[i]);
26            }
27    }
28    while (!c.isEmpty()){
29        e1 = (Token) c.pop().data;
30        System.out.print(e1.operator);
31    }
32 }
```

the condition is satisfied `ch` is pushed onto the stack (Line 26).

After all these operations, if there is any element left in the stack (Line 29) those elements are popped one by one from the stack (Line 30) and written onto the stack (Line 31).

Figure 3.8 shows how we find the postfix representation of the infix expression $A + B * (C + D)$ using a stack.

CHAPTER 3. STACK

				<div>(</div> <div>*</div> <div>+</div>
	<div>+</div>	<div>+</div>	<div>*</div> <div>+</div>	<div>(</div> <div>*</div> <div>+</div>
A	+	B	*	(
A	A	AB	AB	AB
<div>(</div> <div>*</div> <div>+</div>	<div>+</div> <div>(</div> <div>*</div> <div>+</div>	<div>+</div> <div>(</div> <div>*</div> <div>+</div>	<div>*</div> <div>+</div>	
C	+	D)	
ABC	ABC	ABCD	ABCD+	ABCD+*+

Figure 3.8: Finding the postfix representation of the infix expression $A + B * (C + D)$ using a stack

1. A is an operand. It is written on the screen.
2. + is an operator. Since there are not elements in the stack, it is pushed onto the stack.
3. B is an operand. It is written on the screen.
4. * is an operator. Since the precedence of * is larger than the precedence of the topmost element +, it is pushed onto the stack.
5. (is an operator. (is directly pushed onto the stack.

6. C is an operand. It is written on the screen.
7. + is an operator. Since the precedence of + is larger than the precedence of the topmost element (, it is pushed onto the stack.
8. D is an operand. It is written on the screen.
9.) is an operator. We pop elements from the stack until we see (and we write each popped element on the screen.
10. The remaining elements in the stack (* and +) are popped from the stack and written on the screen.

3.4 Solved Exercises

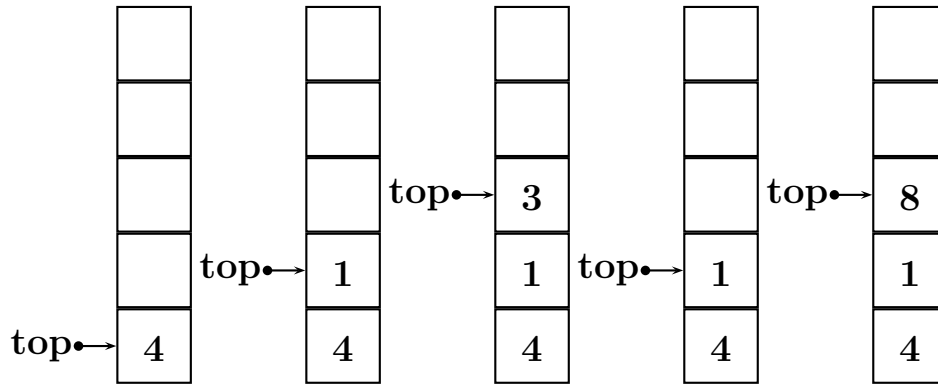
1. What will be the output of the following code fragment?

<pre>Stack c; c = Stack(); c.push(4); c.push(1); c.push(3); cout << c.pop(); c.push(8);</pre>	<pre>Stack c; c = new Stack(); c.push(4); c.push(1); c.push(3); System.out.print(c.pop()); c.push(8);</pre>
---	--

Third line will push 4 onto the stack `c`, fourth line push 1 onto the stack `c`, fifth line push 3 onto the stack `c`. Sixth line pop the topmost element, namely 3 from the stack and write it to the screen. Seventh line push 8 onto the stack `c`. As a result, when the program run the output of the program will be

3

CHAPTER 3. STACK



2. Write a function that prints the binary representation of number N (Hint: Use stack to reverse the digits).

```
void binary(int N)
```

The function that prints the binary representation of number N using a stack is given below. In order to convert from 10 representation to binary representation, we need to divide the number until it is 0 and then write the remainders in these divisions in the reverse order. For example, in order to convert 10 into binary representation;

- 10 is divided by 2. Result is 5 and the remainder is **0**.
- 5 is divided by 2. Result is 2 and the remainder is **1**.
- 2 is divided by 2. Result is 1 and the remainder is **0**.
- 1 is divided by 2. Result is 0 and the remainder is **1**.

When we read the remainders in the division operations, we get 1010. The function below logically follows these rules. The number at hand is divided one ofater another by 2 (Line 11), at each division the remainder (Line 7), since we will need it in the reverse order, is pushed onto the stack (Lines 8-9). When the number pushed onto the stack are popped from the stack (Line 13-14) we will get the remainders in the reverse order.

3.4. SOLVED EXERCISES

```
void binary(int N){
    int digit ;
    Stack c;
    Element e;
    c = Stack(1000);
    while (N > 0){
        digit = N % 2;
        e = Element(digit);
        c.push(e);
        N = N / 2;
    }
    while (c.top != -1){
        e = c.pop();
        cout << e.data;
    }
}
```

```
static void binary(int N){
    int digit;
    Stack c;
    Element e;
    c = new Stack(1000);
    while (N > 0){
        digit = N % 2;
        e = new Element(digit);
        c.push(e);
        N = N / 2;
    }
    while (c.top != -1){
        e = c.pop();
        System.out.print(e.data);
    }
}
```

3. Write a function that returns the maximum element of a stack. You are only allowed to use pop, push, isEmpty functions. The stack must contain the same elements in the same order after the execution of this function.

```
int largest ()
```

The function that returns the maximum element of a stack is given below. The element that is currently processed is stored in the variable **element** (Line 5, 9), the largest element until currently processed element is stored in the variable **largest**. Each element is compared with the **largest** (Line 11), if the current element is larger than the **largest**, the value of the **largest** is changed to the current element (Line 12). In order to put the popped elements back, each popped element is pushed onto a temporary stack (Line 6, 10). After finding the largest element of the stack, the elements in the temporary stack are popped (Line 15) and pushed back onto the original stack (Line 16).

<pre> int largest(){ int largest ; Element element; Stack t = Stack(N); element = pop(); t.push(element); largest = element.data; while (!isEmpty()){ element = pop(); t.push(element); if (element.data > largest) largest = element.data; } while (!t.isEmpty()){ element = t.pop(); push(element); } return largest ; } </pre>	<pre> int largest(){ int largest; Element element; Stack t = new Stack(N); element = pop(); t.push(element); largest = element.data; while (!isEmpty()){ element = pop(); t.push(element); if (element.data > largest) largest = element.data; } while (!t.isEmpty()){ element = t.pop(); push(element); } return largest; } </pre>
--	---

3.5 Exercises

1. `c.push(1);, c.push(2);, c.push(3);, System.out.print(c.pop());, System.out.print(c.pop());, System.out.print(c.pop());`

Without changing the order of lines containing push functions, when the 6 lines in the code fragment are permuted, we can get different permutations of numbers 1, 2, 3. For example, with the following code fragment

```

c.push(1);
c.push(2);
c.push(3);
System.out.print(c.pop());
System.out.print(c.pop());
System.out.print(c.pop());

```

we get the permutation 3, 2, 1,

```
c.push(1);
System.out.print(c.pop());
c.push(2);
System.out.print(c.pop());
c.push(3);
System.out.print(c.pop());
```

we get the permutation 1, 2, 3. Which permutation can not be obtained using this procedure?

2. Write a function that swaps the two topmost elements of a stack. Write the function for both link list and array implementations.

```
void swapTopmost2()
```

3. It is possible to use one array to define two stacks together: First stack starts from zeroth position and expands to increasing positions, whereas second stack starts from the last position ($N - 1$) and expands to decreasing positions. Define such a data structure with corresponding basic functions (pop, push, isEmpty etc.).

3.6 Problems

1. Write a function using stacks that determines if a parenthesis sequence is balanced or not. For example the parenthesis sequence (() ()) is balanced, whereas the parenthesis sequence ((() (is not. You can assume that the character sequence contains just '(' and ')' characters.

```
boolean isBalanced(String s)
```

2. Write a function using stacks that determines if a parenthesis sequence is balanced or not. For example the parenthesis sequence ({ () [{ }] } ()) is balanced, whereas the parenthesis sequence (}]) (is not. You can assume that the character sequence contains just (, {, [,), },] characters.

```
boolean isBalanced(String s)
```

3. For array representation, write a function that enlarges the stack when it is full. The new stack will hold two times more than the original stack.

CHAPTER 3. STACK

void enlarge()

4. Write a function that determines if a character sequence is palindrome, that is, it is equal its reverse. (Hint: Use a stack to reverse the character sequence)

boolean palindrom(String s)

5. Write a function that returns the bottom element of a stack. You are only allowed to use pop, push, isEmpty functions.

Element bottom()

6. Write a function which pops the last two items of a stack and returns the product of those two popped numbers. You are not allowed to use pop, push functions. Write the function for both link list and array implementations.

int multiply()

7. Write a function which removes only the bottom element of the stack. You are only allowed to use pop, push, isEmpty functions (Hint: Use external stack).

void removeBottom()

8. Write a function which returns a copy of the stack as a new stack. You are not allowed to use pop, push functions. Write the function for both link list and array implementations.

Stack copy()

9. Write a function which double each item in the stack, that is, each item appears two times one after another in the stack. You are only allowed to use pop, push, isEmpty functions (Hint: Use external stack).

void doubleStack()

10. Write a function which removes only the middle element of the stack. You are only allowed to use pop, push, isEmpty functions (Hint: Use external stack).

void removeMiddle()

3.6. PROBLEMS

11. Write a function which removes only the K bottom elements of the stack.

void removeBottomK(**int** K)

12. Implement the stack as a doubly linked list. Your implementation must only contain a head node of type `DoubleNode`, and three functions **boolean** `isEmpty()`, `DoubleNode pop()` and **void** `push(DoubleNode newNode)`.

13. Write a method which removes only the odd indexed (1, 3, ...) elements from the stack. The first element at the bottom has index 1. You are only allowed to use `pop`, `push`, `isEmpty` functions (Hint: Use external stack). You are not allowed to use any stack attributes such as N , `top`, `array` etc. You can solve the question with any stack implementation.

void removeOddIndexed()

14. Write a method which pops the k 'th element from the top and returns it. `pop(1)` is equal to the original `pop`, that is, the first element at the top has index 1. You are not allowed to use any stack methods and any external stacks, just attributes, constructors, getters and setters. Write the code for both array and linked list implementations.

`Element pop(int k)`

15. Write a method that compresses the stack by removing the duplicate items. Assume that the elements in the stack are sorted. You are only allowed to use `pop`, `push`, `isEmpty` functions. Write the method for one of the array or linked list implementations. (Hint: Use external stack).

void compress()

Contents of the stack before:

1 1 2 2 2 5 5 5 5 6

Contents of the stack after:

1 2 5 6

16. Write the method

void push(**int** k, **int** data)

which pushes data as the k 'th element from the top. `push(1, data)` is equal to the original `push`, that is, the first element at the top has index 1. You are not allowed to use any stack methods and any external

CHAPTER 3. STACK

stacks. You are allowed to use attributes, constructors, getters and setters. Write the method for both array and linked list implementations.

Contents of the stack before:

1 2 0 4 6 9

Contents of the stack after push(3, 10):

1 2 0 4 10 6 9

Contents of the stack after push(5, 7):

1 2 0 7 4 10 6 9

17. Write a method which removes only the even indexed (2, 4, . . .) elements from the stack. The first element at the bottom has index 1. You are only allowed to use pop, push, isEmpty functions (Hint: Use external stack). You are not allowed to use any stack attributes such as N, top, array etc.

void removeEvenIndexed()

18. Write the method

LinkedList popBottomK(**int** k)

which pops k elements from the bottom and return them as a singly linked list. You are not allowed to use any stack methods and any external stacks. You are allowed to use attributes, constructors, getters and setters. Write the method for linked list implementation.

19. Write a static method using an external stack (only one external stack is allowed) that determines if an integer array is balanced or not. A number k less than 10 is balanced with the number $10+k$. For example, the array 2, 3, 13, 12, 4, 14 is balanced, whereas 5, 15, 4, 3, 14, 13 not. You are not allowed to use any stack attributes such as N, top, array etc. Write the method in array implementation.

boolean isBalanced(**int** [] a)

20. Write the method

void addToStack(Stack s, **int** p, **int** q)

3.6. PROBLEMS

which adds all elements indexed from p to q of the stack s to the top of the original stack. Your stack should contain new nodes, not the nodes in the stack s. The element at the top of the stack has index 1. You are not allowed to use any stack methods and any external stacks. You are allowed to use attributes, constructors, getters and setters. Write the method for linked list implementation.

Contents of the original stack

1 4 5 2

Contents of the stack s

3 1 7 5 11 14 6 8 16

After addToStack(s, 3, 6)

1 4 5 2 6 14 11 5

21. Write a function that inserts a new integer after the largest element of the stack. Write the function for array implementation. You are not allowed to use any stack methods, just attributes, constructors, getters and setters.

```
void insertAfterLargest (int s)
```

Contents of the stack before inserting 6:

1 8 7 3

Contents of the stack after inserting 6:

1 8 6 7 3

22. Write the method

```
void removeOddIndexed()
```

which removes only the odd indexed (1, 3, ...) elements from the stack. The bottom element has index 1. You are only allowed to use pop, push, isEmpty functions. You should use a single external stack. You should use linked list implementation of stack.

Contents of the stack before:

1 2 0 4 6 9

Contents of the stack after:

2 4 9

The queue data structure is very similar to the stack data structure. As the stack data structure, in the queue data structure the elements are stored in a list. Similarly two types of operations are defined for the queue data structure: Adding an element (enqueue) and removing an element (dequeue). From these operations, like in the stack, adding an element is done to the end of the list. On the other hand, when we remove an element, we do not remove the last element, but the first element. This, shows us that the queue data structure is a first-in first-out type data structure. Like the persons waiting in a queue.

When persons try to enter a queue they do directly to the end of the queue. The matters of the persons in the queue are handled starting from the first person in the queue. Therefore, the elements are processed (are deleted) according to their entrance order to the queue. So there are two types of links in the queue data structure: The link pointing to the start of the queue and the link pointing to the end of the queue. When each element is deleted, the link pointing to the first element will point to the second element. When an element is inserted, the link pointing to the last element will changed to point to the added element.

The queue data structure can be implemented in two different ways. Either the queue can be represented using an array or can be defined using a linked list.

4.1. ARRAY IMPLEMENTATION

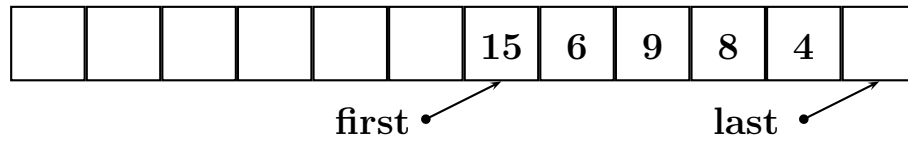


Figure 4.1: The queue data structure that can hold at most 12 elements

4.1 Array Implementation

When we represent the queue structure using an array, the elements are stored in a static array. Table 4.1 shows the definition of the queue structure containing integers. The elements are stored in the field **array**. **first** field points to the first element of the queue. **last** field points to the last element of the queue. When the queue is created, the **first** and **last** fields are equal to each other and take the value 0. Note that, the first element of the queue must not be stored in the zeroth position in the array. The queue may have been started from some middle position of the array and may have been ended elsewhere. The reason of this is that, the enqueue and dequeue operations modify the fields **first** and **last**.

The function *isFull* checks whether a given queue is full or not. If the queue is full, the function will return true, otherwise will return false. Since the queue can hold at most $N - 1$ elements, when the link pointing to the last element is one before the link pointing to the first element, the queue will be full. The function *isEmpty* checks whether a given queue is empty or not. If the queue is empty, the function will return true, otherwise will return false. When both links **first** and **last** points to the position, the queue will be empty.

Figure 4.1 shows the queue data structure that can hold at most 12 elements. The Figure shows the queue after the elements 15, 6, 9, 8, and 4 were added. **first** link points to the first element, namely 15, **last** link points to the position where a new element can be added.

CHAPTER 4. QUEUE

Table 4.1: Definition of a queue, which contains integers and implemented using array

<pre>class Queue{ private: Element array []; int first ; int last ; int N; public: Queue(int N); ~Queue(); bool isFull (); bool isEmpty(); } Queue::Queue(int N) { array = new Element[N]; this.N = N; first = 0; last = 0; } Queue::~Queue{ delete [] array; } bool Queue::isFull (){ if (first == (last + 1) % N) return true; else return false ; } bool Queue::isEmpty(){ if (first == last) return true; else return false ; }</pre>	<pre>public class Queue{ Element array[]; int first; int last; int N; public Queue(int N){ array = new Element[N]; this.N = N; first = 0; last = 0; } boolean isFull(){ if (first == (last + 1) % N) return true; else return false; } boolean isEmpty(){ if (first == last) return true; else return false; } }</pre>
---	--

4.1.1 Insertion

Table 4.2 shows the algorithm that inserts a new element to an array-implemented queue. When we want to insert a new element to the queue , we need to first check if the queue is full or not (Line 2). If the queue is not full, the new

4.1. ARRAY IMPLEMENTATION

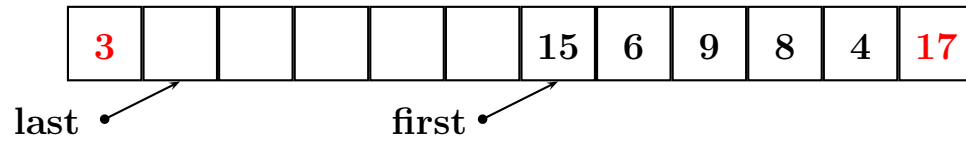


Figure 4.2: Adding elements 17 and 3 to the queue structure given in Figure 4.1

element will be placed to the position pointed by the link **last** of the queue (Line 3). If the link **last** does not point to the end of the array it will be incremented by one, otherwise, since incrementing will put the link out of the array, will be set to point to the first position of the array (Line 4).

Table 4.2: The algorithm that adds a new element to an array implemented queue

<pre> void Queue::enqueue(Element element){ if (! isFull ()) { array[last] = element; last = (last + 1) % N; } } </pre>	<pre> void enqueue(Element element){ if (! isFull ()) { array[last] = element; last = (last + 1) % N; } } </pre>
---	--

Figure 4.2 shows the queue after adding elements 17 and 3 to the queue shown in Figure 4.1. When 17 is added to the queue, the **last** link points to the first position of the array. After adding 3, the **last** link points to the second position of the array.

4.1.2 Deletion

Table 4.3 shows the algorithm that removes an element (returns also that element) from an array implemented queue. When we want remove an element from the queue, we need to first check whether the queue is empty or not (Line 3). If the queue is not empty, we return the first element of the queue (Line 4, 6) and the link **first** pointing to the first element of the queue is incremented by one. But if the link **first** points to the last position of the array, it will point to outside of the array, if it is incremented by one. What

CHAPTER 4. QUEUE

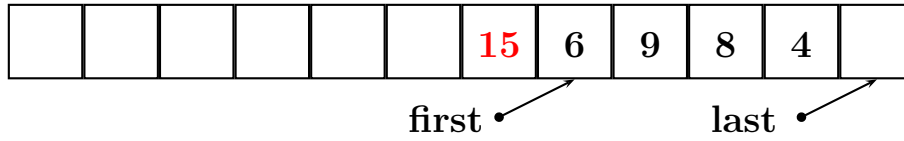




Figure 4.3: Removing element 15 from the queue structure given in Figure 4.1

we should do is the same as inserting an element, this time modifying the link `first` to point to the first position (taking value 0) of the array (Line 5).

Table 4.3: The algorithm that removes an element (and returns that element) from an array implemented queue

<pre> Element Queue::dequeue(){ Element result; if (!isEmpty()){ result = array[first]; first = (first + 1) % N; return result; } return Element(0); } </pre>	<pre> Element dequeue(){ Element result; if (!isEmpty()){ result = array[first]; first = (first + 1) % N; return result; } return null; } </pre>
---	--

Figure 4.3 shows the queue after removing an element from the queue given in Figure 4.1. When the element 15 is removed from the queue, the link `first` will point to 6.

Queue Operations (Array)	
	Insertion: $\mathcal{O}(1)$
	Deletion: $\mathcal{O}(1)$

4.2 Linked List Implementation

When we implement the queue data structure using linked lists, each element is stored in a node of the linked list. The nodes of the linked list are connected to each other using links. Similar to the array implementation we need links pointing to the head and tail of the queue. The links are shown using the **first** and **last** fields in the linked list. Table 4.4 shows the queue data structure, containing integers as data, implemented using linked lists. When the queue is created using linked lists, links **first** and **last** are set to the default value NULL.

The function *isEmpty* checks whether a linked list implemented queue is empty or not. If the queue is empty, the function returns true, otherwise return false. When there is not element in the queue, the linked list is empty and the link **first** is NULL.

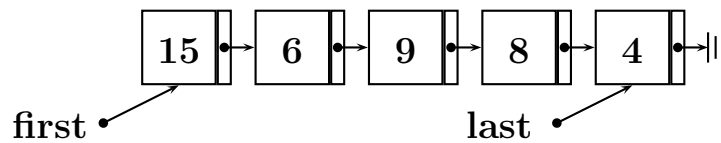


Figure 4.4: The queue structure that contains 5 elements and implemented using linked list

Figure 4.4 shows an example queue structure which is defined using linked list and contains 5 elements. The Figure shows the queue after adding element 15, 6, 9, 8, and 4 one after another. The link **first** points to the first element of the queue, namely 15, the link **last** points to the last element of the queue, namely 4.

4.2.1 Insertion

When linked list is used to implement the queue data structure, there is no possibility that the queue is full. As long as there is memory, one can allocate memory to add new elements .

Table 4.5 shows the algorithm that add a new element to the queue implemented using linked list. The thing to do is to place the new element to

CHAPTER 4. QUEUE

Table 4.4: Definition of a queue, which contains integers and implemented using linked list

<pre>class Queue{ private: Node* first; Node* last; public: Queue(); ~Queue(); bool isEmpty(); } Queue::Queue(){ first = nullptr; last = nullptr; } Queue::~~Queue(){ Node* tmp = first; while (tmp != nullptr){ Node* next = tmp->next; delete tmp; tmp = next; } } Queue::bool isEmpty(){ if (first == nullptr) return true; else return false; }</pre>	<pre>public class Queue{ Node first; Node last; public Queue(){ first = null; last = null; } boolean isEmpty(){ if (first == null) return true; else return false; } }</pre>
--	---

the position pointed by the link **last** of the queue. After that, the link **last** will point to the new element (Line 3). If there is no element in the list before adding a new element, the links **first** and **last** will point to the new added element (Line 5).

Figure 4.5 shows the queue after adding element 17 to the queue given in Figure 4.4. After adding 17, the link **last** will point to 17.

4.2. LINKED LIST IMPLEMENTATION

Table 4.5: Adding a new element to the queue implemented using linked list

<pre> void Queue::enqueue(Node* node){ if (!isEmpty()) last->next = node; else first = node; last = node; } </pre>	<pre> void enqueue(Node node){ if (!isEmpty()) last.next = node; else first = node; last = node; } </pre>
--	--

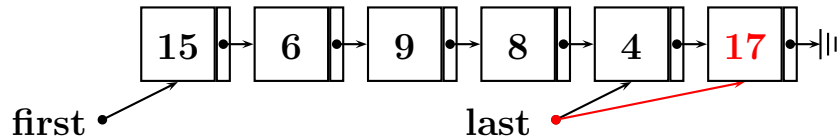


Figure 4.5: Adding element 17 to the queue structure given in Figure 4.4

4.2.2 Deletion

Table 4.6 shows the algorithm that removes an element (and returns that element) from a queue implemented using linked list. In order to remove the first element from a linked list implemented queue structure, we need to advance the link pointing to the first element (Line 5). This is only possible if there is at least one element in the list. Whether there is an element in the list is checked using the function *isEmpty* (Line 4). When the list is empty, the algorithm that removes an element must return NULL. Since the link *first* of the list takes the value NULL when the list is empty, the link *first* can be easily returned (Line 9). If there is a single element in the list, after removal of the first element, the link *first* will be NULL (Line 6), and since in this case the list will be empty, we need to set also the link *last* to NULL (Line 7). The program that uses this function can also understand if the queue is empty or not by checking the return value of this function. If the function has returned NULL, the queue has been emptied, if the function has returned any address, it is not empty.

Figure 4.6 shows the queue after removing 15 from the queue given in Figure 4.4. When we delete 15 from the queue, the link *first* will point to 6.

CHAPTER 4. QUEUE

Table 4.6: The algorithm that removes an element (and returns that element) from the queue implemented using linked list

```
Node* Queue::dequeue(){
    Node* result;
    result = first ;
    if (!isEmpty()){
        first = first->next;
        if ( first == nullptr)
            last = nullptr;
    }
    return result ;
}
```

```
Node dequeue(){
    Node result;
    result = first ;
    if (!isEmpty()){
        first = first .next;
        if ( first == null)
            last = null;
    }
    return result;
}
```

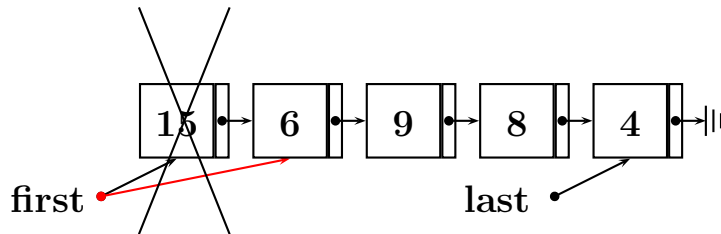


Figure 4.6: Removing element 15 from the queue structure given in Figure 4.4

Queue Operations (Linked List)

Insertion: $\mathcal{O}(1)$

Deletion: $\mathcal{O}(1)$

4.3 Application: Darts

In the game of Darts, the aim is to get a sum of 100 points by shooting darts. One can shoot as many darts as possible, but if the sum of shots is larger than 100, one fails. An example dart board is shown in Figure 4.7. In this

4.3. APPLICATION: DARTS

example, if we shoot 2 times 11, 2 times 21, and 1 time 36, we get the sum $2 \times 11 + 2 \times 21 + 1 \times 36 = 100$. On the other hand, if we shoot 2 times 36 and 1 time 33 ($2 \times 36 + 33 = 105$) we can not get the sum needed.

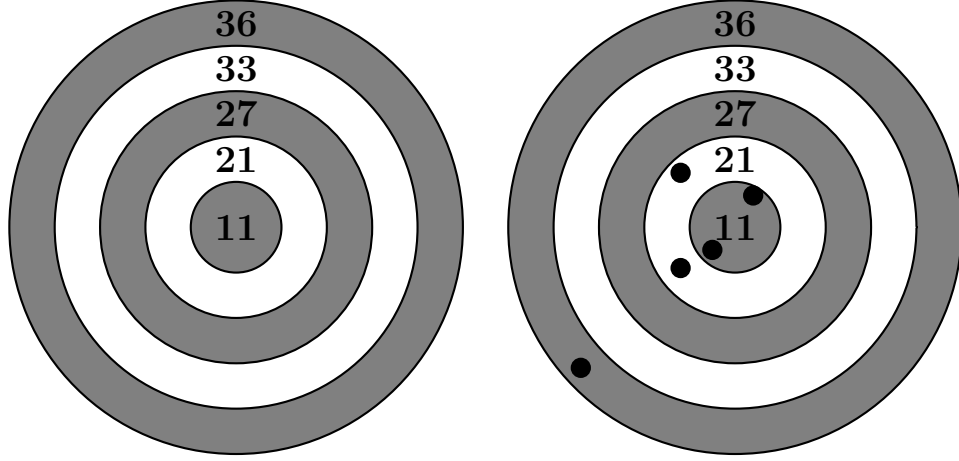


Figure 4.7: An example Dart board and getting 100 in five shots

A lot of problems, like the Dart game problem, can be expressed as a search problem in a state space. In the state space, each state we traverse while solving the problem, is represented as a state. For example, in the Dart game problem, at the beginning, no shots are done and we are in the state of 0; in another case, if 33, 21, and 11 are shot, we are in the state ($33 + 21 + 11 =$) 65.

In the state space, we can move from one state to another by some pre-defined actions. Here an action is defined according to the problem and expressed as a movement from a state to another. For example, in the Dart game problem, we move from one state to another by shooting darts. When we are in the beginning state 0, if we shoot 33, we will move to the state 33. When we are in the state 54 and we shoot 36, we will move to the state ($54 + 36 =$) 90. In the Dart game problem, (The numbers between 0 and 100) there are 101 different states. But some states may not be reachable. For example, in the Dart board given, whatever we do, we can not reach the states like 10, 15, 30, because how we will shoot, the sum of our shots can

CHAPTER 4. QUEUE

not be 10, 15 or 30.

Therefore, our aim in the Dart game problem, is to determine how we will move from the start state (0) to the end state (100). These types of search problems can be solved using the search techniques in the computer science literature. Two types of search techniques are mostly used: breadth first search and depth first search.

4.3.1 Breadth First Search

In breadth first search, we first explore the states that can be accessed directly from the beginning state, then the states that can be accessed with one intermediate state, then the states that can be accessed with two intermediate states, etc. and if one of these states is an end state, we will finish the search. For example, in the Dart game given above, with one shot we can move from the beginning state (0) to the states 11, 21, 27, 33, and 36. On the other hand, with two shots, we can move to the states 22 (11 + 11), 32 (11 + 21), 38 (11 + 27), 42 (21 + 21), 44 (11 + 33), 47 (11 + 36), 48 (21 + 27), 54 (27 + 27 or 21 + 33), 57 (21 + 36), 60 (27 + 33), 63 (27 + 36), 66 (33 + 33), 69 (33 + 36), and 72 (36 + 36). As can be seen, the number of states that can be accessed is increasing exponentially with the number of shots (Figure 4.8).

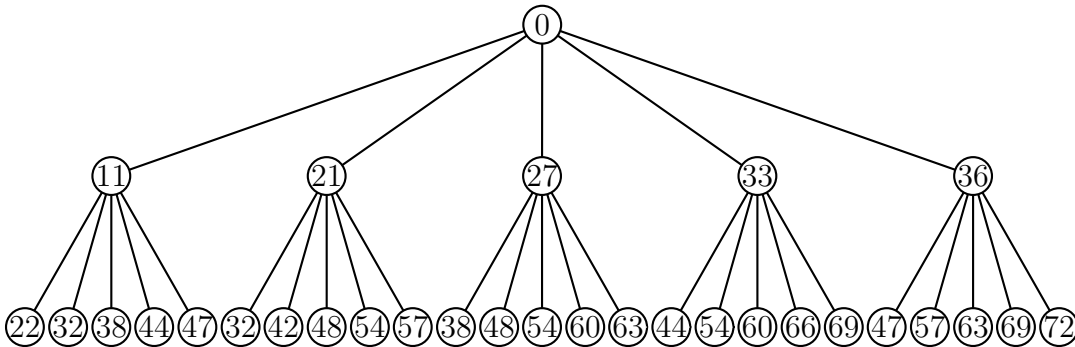


Figure 4.8: Applying two stages of the breadth first search in the Dart game problem

In general, in breadth first search, if we can move to b different states

4.3. APPLICATION: DARTS

from each state, after d stages we can move a total

$$\sum_{i=1}^d b^i = \frac{b^{d+1} - 1}{b - 1} \sim b^d \quad (4.1)$$

states. These states may not be different than each other, in other words, we can encounter same states again and again in breadth first search. For example, in the Dart game above, we can reach state 54 both by shooting 27 twice or by shooting first 21, then 33.

4.3.2 Solving Darts Problem Using Breadth First Search

An easy way of the application of the breadth first search is using a queue data structure. Each element in the queue corresponds to a single state. In the beginning, there is only the beginning state in the queue. At each stage, we remove a state from the queue and the states that are reachable from this state are added to the end of the queue.

Since the states that are reachable from the beginning state are in the first level, the states in the first level are added to the queue. Then the states that are reachable from the states in the first level, namely states in the second level, are added to the queue. This addition continues until we arrive to the end state. Note that, only the states at a level, and the states that are reachable from that level appear in the queue.

Table 4.7: Definition of a state in the Dart game problem

<pre> class State{ private: int total; string darts; public : State(int total, string darts); } State(int total, string darts){ this->total = total; this->darts = darts; } </pre>	<pre> public class State{ int total; String darts; public State(int total, String darts){ this.total = total; this.darts = darts; } } </pre>
---	--

CHAPTER 4. QUEUE

Since we are not only asked if the problem is solvable or not, but also the solution of the problem, that is, which shots have been done to arrive the final state, we need to modify the definition of each element in the queue data structure. Figure 4.7 shows the definition of the new element that also includes this information. Field **total** represents the sum of shots that are done to reach this state, **darts** field represents the shots done to reach this state. For example, if we have shot 21 and 36 from the beginning state to arrive this state, **total** will be 57, and **darts** will be "21 36".

Table 4.8: Solution of the Dart game problem using breadth first search

<pre> static string dartGame(int[] board){ int i, t; string a; Node* e; Queue k; State s; e = new Node(State(0, "")); k = Queue(); k.enqueue(e); while (!k.isEmpty()){ s = k.dequeue()->data; if (s.total == 100) return s.darts; for (i = 0; i < board.length; i++){ if (s.total + board[i] <=100){ t = s.total + board[i]; a = s.darts + "-" + board[i]; e = new Node(State(t, a)); k.enqueue(e); } } } return ""; } </pre>	<pre> static String dartGame(int[] board){ int i, t; String a; Node e; Queue k; State s; e = new Node(new State(0, "")); k = new Queue(); k.enqueue(e); while (!k.isEmpty()){ s = k.dequeue().data; if (s.total == 100) return s.darts; for (i = 0; i < board.length; i++){ if (s.total + board[i] <=100){ t = s.total + board[i]; a = s.darts + "-" + board[i]; e = new Node(new State(t, a)); k.enqueue(e); } } } return null; } </pre>
--	---

The solution to the Dart game problem using breadth first search is given in Table 4.8. As the first step, the beginning state, the empty board state (Line 6), is added as the first element to the queue (Line 8). At each stage, one state is removed from the queue (Line 10), if the sum of the shots of this state is 100 (Line 11), the function returns the shots list of this state (Line

12). For each state retrieved from the queue, the states that can be reached from that state are analyzed (Line 13). If the reachable state is a defined state, that is, the sum of its shots is less than or equal to 100 (Line 14), new state is added to the queue (Lines 15-18). The shots list of the new state is obtained by adding the number (as a string) to the shots list of the previous state (Line 16).

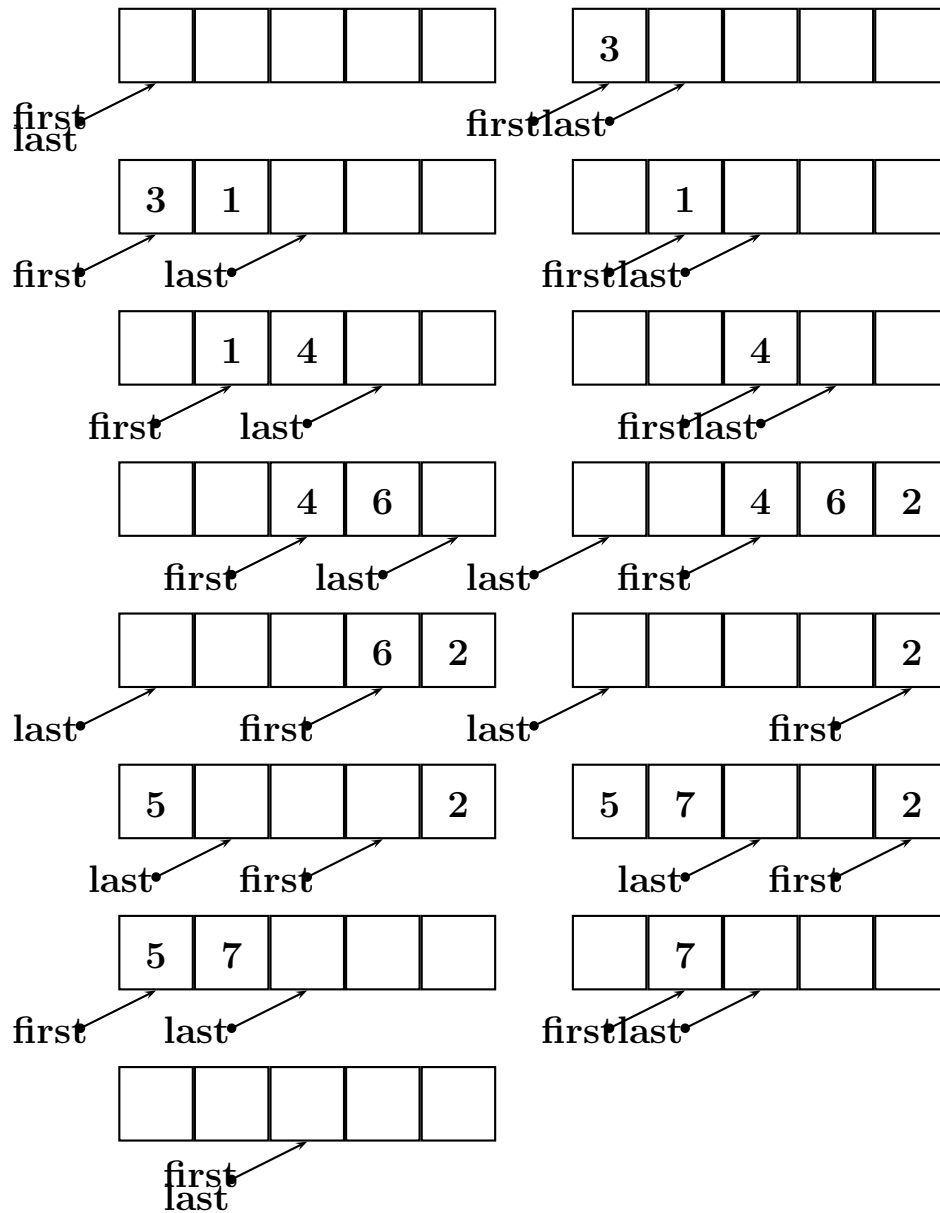
4.4 Solved Exercises

1. Show the resulting queue after each line in the following code fragment. What must be the minimum size of the queue such that none of the following operations will throw an exception?

Queue k = Queue(5);	Queue k = new Queue(5);
k.enqueue(3);	k.enqueue(3);
k.enqueue(1);	k.enqueue(1);
k.dequeue();	k.dequeue();
k.enqueue(4);	k.enqueue(4);
k.dequeue();	k.dequeue();
k.enqueue(6);	k.enqueue(6);
k.enqueue(2);	k.enqueue(2);
k.dequeue();	k.dequeue();
k.dequeue();	k.dequeue();
k.enqueue(5);	k.enqueue(5);
k.enqueue(7);	k.enqueue(7);
k.dequeue();	k.dequeue();
k.dequeue();	k.dequeue();
k.dequeue();	k.dequeue();

The state of the queue after each operation is given in the Figure below. As can be seen, there are 3 elements in the queue at the same time. Therefore, while performing these operations, the minimum size of the queue must be at least $3 + 1 = 4$, so that none of those operations will throw an exception.

CHAPTER 4. QUEUE



- For array representation, write a function that moves the element currently at the rear of the queue to the front of the queue.

void bringFront()

The function that moves the element currently at the rear of the queue to the front of the queue is given below. First, we prepare space in the

4.4. SOLVED EXERCISES

beginning of the queue, in other words, we decrement the link pointing to the first element of the queue (Line 2). We also decrement the link pointing to the last element of the queue (Line 3). As a last step, we move the element at the rear of the queue to the front of the queue (Line 4).

```
void bringFront(){
    first = ( first - 1 ) % N;
    last = (last - 1) % N;
    array[ first ] = array[ last ];
}
```

```
void bringFront(){
    first = ( first - 1 ) % N;
    last = (last - 1) % N;
    array[ first ] = array[ last ];
}
```

3. Describe how to implement the queue data structure using one stack. Modify enqueue and dequeue operations to work on the new data structure.

The definition of the queue, when we change the definition of the queue using a stack, is given below.

```
class Queue{
    private:
        Stack stack;
    public:
        Queue(int N);
}

public Queue::Queue(int N){
    stack = Stack(N);
}
```

```
public class Queue{
    Stack stack;
    public Queue(int N){
        stack = new Stack(N);
    }
}
```

The function that removes one element from the queue, according to the modified queue definition, is given below. The queue is a head out but the stack is a rear out type data structure. In order to remove one element from the new queue definition, we need to remove one element from the stack belonging to the queue. In order to remove one element from the stack, first all elements in the stack must be removed from the stack and moved to a temporary stack (Line 5-9), then again removed from the temporary stack and moved to the original stack back (Line 11-14). The last element that is moved in the first phase is the element we want to delete (Line 10, 15).

<pre> Node* dequeue(){ Node *e, *result ; Stack c; if (!stack.isEmpty()){ c = Stack(N); while (!stack.isEmpty()){ e = stack.pop(); c.push(e); } result = c.pop(); while (!c.isEmpty()){ e = c.pop(); stack.push(e); } return result ; } } </pre>	<pre> Element dequeue(){ Element e, result ; Stack c; if (!stack.isEmpty()){ c = Stack(N); while (!stack.isEmpty()){ e = stack.pop(); c.push(e); } result = c.pop(); while (!c.isEmpty()){ e = c.pop(); stack.push(e); } return result; } } </pre>
--	--

The function that adds a new element to the queue, according to the modified queue definition, is given below. Since both queue and stack data structures are rear-in data structures, in order to add a new element to the modified queue, we only need to add the new element to the stack belonging to the modified queue (Line 3).

<pre> void enqueue(Node* element){ if (!stack.isFull ()) stack.push(element); } </pre>	<pre> void enqueue(Element element){ if (!stack.isFull ()) stack.push(element); } </pre>
--	--

4.5 Exercises

1. Write a function that moves the element currently at the front of the queue to the rear of the queue. Write the function for both array and linked list implementations.

```
void moveToRear()
```

2. For array implementation, write a function that enlarges the queue when it is full. The new queue will hold two times more than the original queue.

3. In our array implementation, the queue can hold at most $N - 1$ elements, when the size of the array is N . Modify the array implementation such that the queue can use all the available space in the array.

4. Write a function that returns the maximum element in a queue. You are only allowed to use enqueue, dequeue, isEmpty functions. The queue must contain the same elements in the same order after the execution of this function.

```
int largest ()
```

5. For linked list implementation, write a function that moves the element currently at the rear of the queue to the front of the queue.

```
void moveToFront()
```

6. Write a function that adds a new element after the front element of the queue. Write the function for both array and linked list implementations.

```
void insertSecond(Element newElement)
```

7. Write a function that shrinks the size of the queue to M . You can assume that the queue contains at most $M - 1$ elements before shrinking.

```
void shrink(int M)
```

8. Modify the queue data structure such that the new implementation uses doubly linked list representation instead of singly linked list representation. Implement enqueue and dequeue operations.
9. A dequeue is a double-ended queue, which is a data structure that supports insertions and deletions from both ends (front and rear). Using a doubly linked list, implement dequeue data structure with its methods.

4.6 Problems

1. Write a function that adds a new element after the K 'th ($K \geq 0$) element of the queue. Write the function for both array and linked list implementations. You can safely assume that, there are at least K elements in the queue.

CHAPTER 4. QUEUE

`void insertAfterKth ()`

2. Write a function that deletes the element in the K 'th ($K \geq 0$) position of the queue. Write the function for array implementation.

`void deleteKth(int K)`

3. Write a method where the method returns the minimum number in a queue. Write the function for both array and linked list implementations. Do not use any class or external methods except isEmpty().

`int minimum()`

4. Write a method which removes and returns the second item from the queue. Write the function for both array and linked list implementations. Your methods should run in $\mathcal{O}(1)$ time. Do not use any class or external methods except isEmpty().

`Element dequeue2nd()`

`Node* dequeue2nd()`

5. Write a function that inserts a new element after the largest element of the queue. Write the function for array implementation. You are not allowed to use any queue methods, just attributes, constructors, getters and setters.

`void insertAfterLargest (int data)`

6. Write a function that creates and returns a new queue by removing even indexed elements from the original queue and inserting into the newly created queue. Write the function for both array and linked list implementations. The first node has index 1. You are not allowed to use any queue or linked list methods, just attributes, constructors, getters and setters.

`Queue divideQueue()`

7. Write the method

`void removeOddIndexed()`

which removes only the odd indexed (1, 3, . . .) elements from the queue. The first element has index 1. You are only allowed to use enqueue, dequeue, isEmpty functions. **You should use external**

queue. You are not allowed to use any queue attributes such as first, last, array etc. You can solve the question with any queue implementation.

8. Write the methods

`Element dequeue(int k), Node dequeue(int k)`

which dequeues data as the k 'th element from the first. `dequeue(1)` is equal to the original `dequeue`, that is, the first element has index 1. You are not allowed to use any queue methods and any external structures (arrays, queues, trees, etc). You are allowed to use attributes, constructors, getters and setters. Write the method for both array and linked list implementations.

9. Write the method for array implementation

`void copyPaste(Queue src, int index)`

which copies all the elements of the src queue and inserts to the queue at position index. You are not allowed to use `enqueue`, `dequeue`, `isEmpty` functions. You can assume the destination queue has enough space for insertion. Your method should run in $\mathcal{O}(N)$ time. Hint: Start by counting the number of positions to shift for opening up space for the elements of src.

10. Write another constructor method

`void Queue(Queue[] list)`

which constructs a new list based queue by concatenating all elements in the list of queues in order. The elements from queues should be recreated (not copied from the queues). You are not allowed to use `enqueue`, `dequeue`, `isEmpty` functions. You should solve the question for list implementation.

11. Write the method for array implementation

`void cutPaste(Queue dest, int p, int q)`

which cuts all the elements between indexes p and q from the original queue and inserts at the end to the dest queue. You are not allowed to use `enqueue`, `dequeue`, `isEmpty` functions. You can assume the destination queue has enough space for insertion. Your method should run in $\mathcal{O}(N)$ time.

CHAPTER 4. QUEUE

12. Write the method

`Queue[] divideQueue(int k)`

which constructs an array of list based queues by dividing the original queue into k equal parts. The first, second, ..., k 'th element of the original queue will be the first element of the first, second, ..., k 'th output queues, etc. The elements of the output queues should be recreated (not copied from the original queue). You are not allowed to use enqueue, dequeue, isEmpty functions. You should solve the question for list implementation.

13. Write another constructor method

`void Queue(Queue[] list)`

which constructs a new array based queue by adding the elements in the *list* of queues one by one. So, the first k elements of the original queue will be constructed with the first elements of the k queues in the list; the second k elements of the original queue will be constructed with the second elements of the k queues in the list etc. The elements from queues should be recreated (not copied from the queues). You are not allowed to use enqueue, dequeue, isEmpty functions. You should solve the question for array implementation.

14. Write the method

`void removeAll(Queue[] list)`

which removes all elements in the queues in the *list* from the original queue. You are not allowed to use enqueue, dequeue, isEmpty functions. You should solve the question for list implementation.

4.6. PROBLEMS

Binary Search Trees

The data structures we have seen in the previous chapters are linear data structures. From this chapter on, we are going to examine non-linear data structures. Tree structure is also a non-linear data structure. Different from the *tree* structure we see in the nature, the tree data structure has its root on top and develops its branches down.

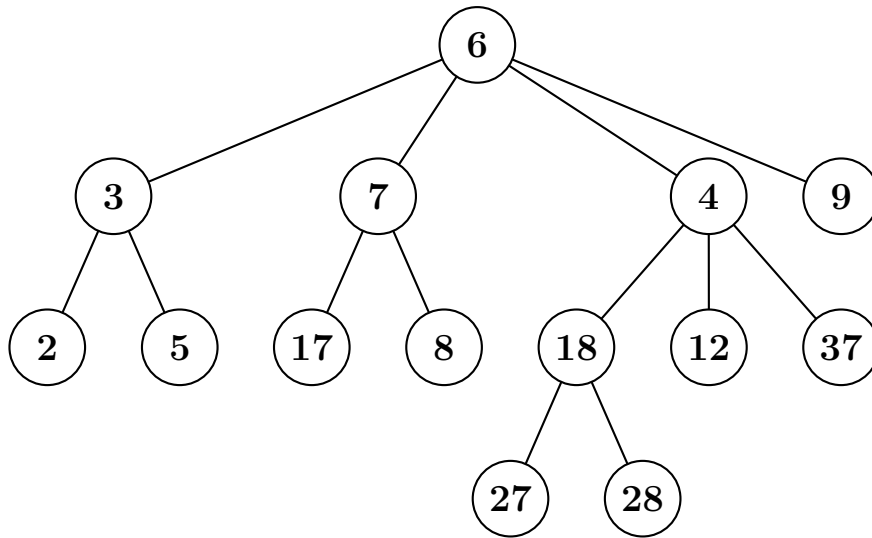


Figure 5.1: An example tree consisting of 14 nodes

Figure 5.1 shows an example tree structure. 6 is the root of this tree. If

5.1. DEFINITION

there is an edge from node x to node y , node x is called the parent of node y , node y is called a child of node x . For example, in the tree shown in Figure 5.1, 6 is parent of 3, and 3 is parent of 2 and 5.

The number of children of a node indicate the degree of that node. For example, in the tree shown in Figure 5.1, the degree of 6 is 4, the degree of 4 is 3, whereas the degree of 7 is 2.

In a tree, starting from node x , if you can reach node y through traversing its children, node y is called a descendant of node x and node x is called ascendant of node y . For example, in the tree shown in Figure 5.1, the descendants of 4 are the nodes 4, 18, 12, 37, 27, and 28. Similarly the ascendants of node 28 are the nodes 6, 4, and 18.

If a node in a tree has not children, that node is called a leaf node. For example, in the tree shown in Figure 5.1, there exists 9 leaf nodes: 2, 5, 17, 8, 27, 28, 12, 37, 9.

The depth of a tree is defined as the maximum number of nodes traversed to reach a leaf node from the root node. For example, the depth of tree shown in Figure 5.1 is 4. This is because, starting from the root node 6 the number of nodes traversed to reach leaf nodes 27 and 28 is 4.

5.1 Definition

If each node in a tree has at most 2 children, then that tree is called a binary tree. In other words, if the degree of each node in a tree is at most 2, then that tree is called a binary tree. For example, the tree shown in Figure 5.1 is not a binary tree. Both 6 and 4 have more than 2 children. On the other hand, the tree shown in Figure 5.2 is a binary tree.

A binary tree is a binary search tree if and only if for each node of the tree, the value of the nodes in its left subtree are smaller than its value and the values of the nodes in its right subtree are larger than its value.

For example, the binary tree shown in Figure 5.2 consisting of 6 nodes is a binary search tree. The root node of the tree is 6 and it is larger than the nodes in its left subtree, namely nodes 2, 3, and 5; also smaller than the nodes in its right subtree, namely nodes 7 and 8. Node 3 is larger than the node in its left subtree, namely 2 and also smaller than the node in its right subtree, namely 5. Node 8 is larger than node 7 and therefore is shown as a right child of it. The depth of this tree is 2.

CHAPTER 5. BINARY SEARCH TREES

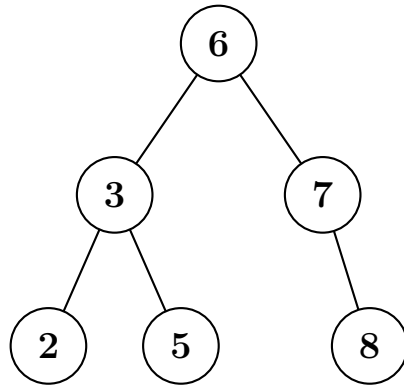


Figure 5.2: A binary search tree consisting of six nodes

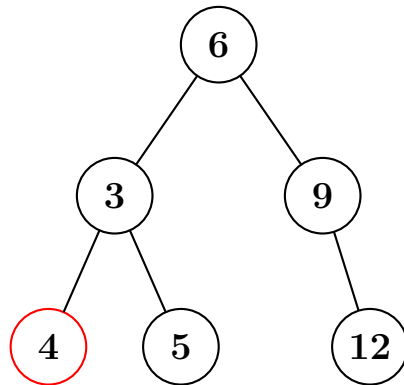


Figure 5.3: The tree consisting of 6 nodes. This tree does not satisfy the binary search tree property. The node on the left side of node 3 is not smaller than 3, although it is required to be smaller.

On the other hand, the tree shown in Figure 5.3 does not satisfy the binary search tree property. The node on the left side of node 3 is 4 and is not smaller than 3, although it is required to be smaller.

The insertion, deletion, and search operations in the linked lists have time complexity of $\mathcal{O}(n)$, where n represents the number of elements in the linked list. Decreasing the time complexity of the basic operations is one of the important existential conditions of the data structures. In the binary search tree, all these operations have the time complexity of $\mathcal{O}(h)$, where h represents the depth of the tree.

5.1. DEFINITION

If a binary search tree of depth h is balanced, the number of nodes is 2^h . Therefore, if we place n nodes in a binary search tree, accessing the nodes in this tree, adding new nodes to this tree and removing nodes from this tree, all have the time complexity of $\mathcal{O}(\log n)$.

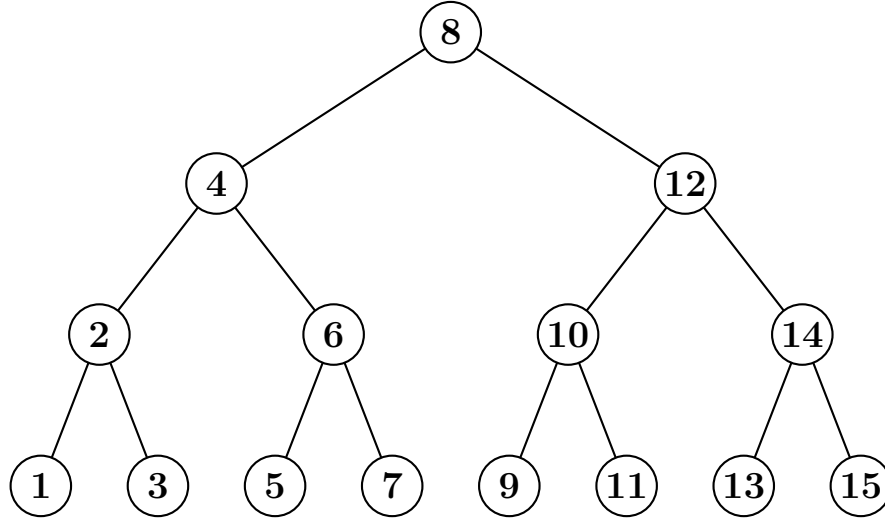


Figure 5.4: A balanced binary search tree of depth 4, which consists of 15 nodes

Figure 5.4 shows an example balanced binary search tree of depth 4 consisting of 15 nodes. In order to reach a leaf node from root node 8, we require to pass exactly $\log(15 + 1) = 4$ nodes. This is the minimum depth tree that can be constructed with that many nodes.

The nodes in the binary search tree shown in Figure 5.2 can be placed differently. Figure 5.5 shows another binary search tree consisting of the same nodes but with a different depth, 4. As can be guessed, this tree is more clumsy than the previous one. For example, in the previous tree, we need to pass 3 nodes to reach the node 5, whereas in this tree we need to pass 5 nodes to reach node 5. The situation is also the same for the node 6, 8, and 7. The binary search tree is degenerate (most unbalanced) when all nodes are on the left or on the right side of the root node. In this case, the binary search tree is turned into a linked list.

Table 5.1 shows the definition of a node in a binary search tree. The node data structure, as the node data structure in the linked list, has a recursive

CHAPTER 5. BINARY SEARCH TREES

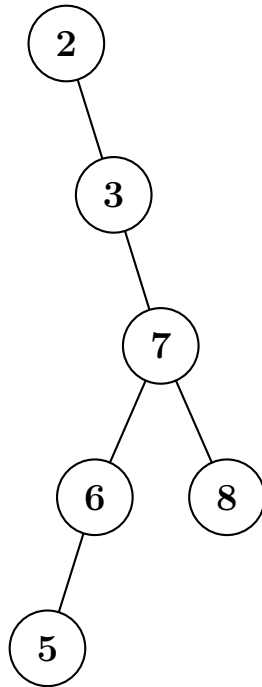


Figure 5.5: A binary search tree of depth 4, which consists of 6 nodes

definition. Each node has a key data field and two links pointing to the left and right child nodes. Since both of these links are also nodes, the definition of node is recursive. If one of the child nodes does not exist, the link pointing to that node has the value of NULL.

Table 5.2 shows the definition of binary search tree using the node definition above. Since the definition of the node data structure is recursive, it can define a tree alone. We only need a link pointing to the first node (root node) of the tree like in the linked list structure. In this case, this is **root** in the definition above. Since in the beginning the tree is empty, **root** is NULL.

5.2. BINARY SEARCH TREE OPERATIONS

Table 5.1: Definition of node of a binary tree

<pre>class TreeNode{ private: int data; TreeNode *left, *right; public: TreeNode(int data); ~TreeNode(); } TreeNode::TreeNode(int data){ this->data = data; left = nullptr; right = nullptr; } TreeNode::~~TreeNode(){ delete left; delete right; }</pre>	<pre>public class TreeNode{ int data; TreeNode left; TreeNode right; public TreeNode(T data){ this.data = data; left = null; right = null; } }</pre>
--	--

Table 5.2: Definition of binary search tree containing integers

<pre>class Tree{ private: TreeNode* root; public: Tree(); ~Tree(); } Tree::Tree(){ root = nullptr; } Tree::~~Tree(){ delete root; }</pre>	<pre>public class Tree{ TreeNode root; public Tree(){ root = null; } }</pre>
---	--

5.2 Binary Search Tree Operations

5.2.1 Search

One of the important properties of binary search tree is that searching a specific value has lower time complexity. For example, searching a specific

CHAPTER 5. BINARY SEARCH TREES

value in a linked list has the time complexity of $\mathcal{O}(n)$, whereas the same operation has the time complexity $\mathcal{O}(\log n)$ in a binary search tree.

Table 5.3: The recursive algorithm that searches a given value in a binary search tree

```
TreeNode* TreeNode::recursiveSearch(int value){
    if (data == value)
        return this;
    else
        if (data > value)
            if (left != nullptr)
                return left->recursiveSearch(value);
            else
                return nullptr;
        else
            if (right != null)
                return right->recursiveSearch(value);
            else
                return nullptr;
}
```

```
TreeNode recursiveSearch(int value){
    if (data == value)
        return this;
    else
        if (data > value)
            if (left != null)
                return left.recursiveSearch(value);
            else
                return null;
        else
            if (right != null)
                return right.recursiveSearch(value);
            else
                return null;
}
```

Table 5.3 shows the recursive algorithm that searches a given value in a binary search tree. Here `d` represents the current node where search is going on, `value` represents the value that is being searched. Searching in a binary search tree is similar to searching in a sorted array. First we compare the searched value with the value in the root node (Line 4). Since the value in

5.2. BINARY SEARCH TREE OPERATIONS

Table 5.4: The iterative algorithm that searches a given value in a binary search tree

<pre> TreeNode* Tree::iterativeSearch (int value){ TreeNode* d; d = root; while (d != nullptr){ if (d.data == value) return d; else if (d.data > value) d = d.left ; else d = d.right ; } return nullptr; } </pre>	<pre> TreeNode iterativeSearch(int value){ TreeNode d; d = root; while (d != null){ if (d.data == value) return d; else if (d.data > value) d = d.left ; else d = d.right ; } return null; } </pre>
---	--

any node in a binary search tree is larger than the values in that node's left subtree and is smaller than the values in that node's right subtree, if the value being searched is smaller than the value in the root node (Line 7) this will show that the value being searched must be in the left subtree of the root node. The recursive algorithm calls itself with the left child of the root node to search the value in the left subtree of the root node (Line 8). Similarly if the value being searched is larger than the value in the root node (Line 9), this will show that the value being searched must be in the right subtree of the root node. In this case, the recursive algorithm calls itself with the right child of the root node to search the value in the right subtree of the root node. The search continues until the searched value is equal to the value in the current search node (Line 4-5). If we can not go any node, that is, we arrived at a leaf node and the searched value is not equal to the value in the leaf node, this will mean that, the searched value does not exist in the tree (Line 2-3).

Table 5.4 shows the non-recursive version of the algorithm that searches a given value in a binary search tree. The search starts from the root node (Line 3) and we represent the node, with which we compare the searched value, with `d`. If the searched value is equal to the value of the current node (Line 5), we have found the node we search for, the function will return that node (Line 6). If the searched value is smaller than the value of the current

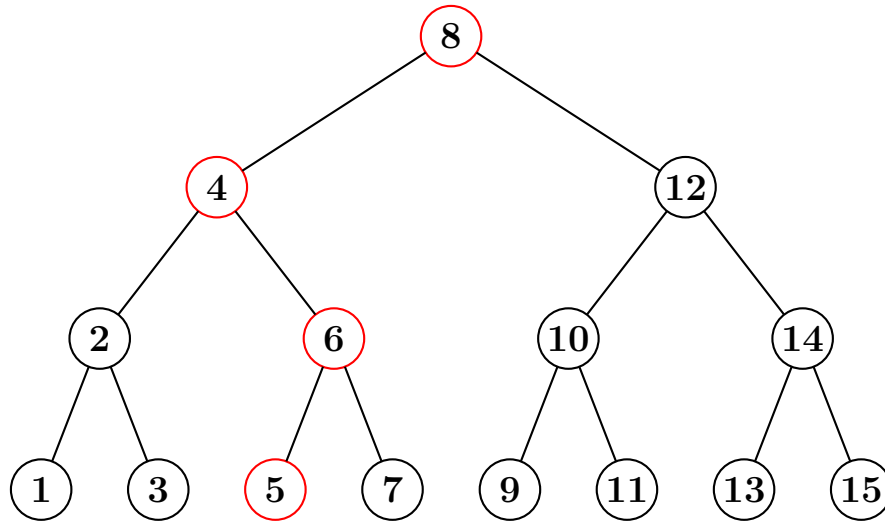


Figure 5.6: Searching a value in a binary search tree

node (Line 8), the number we search for must be on the left subtree of the current node, therefore the new current node must be the left child of the current node (Line 9). As the last case, if the searched value is larger than the value of the current node (Line 10), the number we search for must be on the right subtree of the current node, therefore the new current node must be the right child of the current node (Line 11). If this search continues until the leaf nodes of the tree and we can't find the node we search for, node `d` will be `NULL` and the function will return `NULL` (Line 13).

For example, Figure 5.6 shows the nodes visited when we search 5 in the tree given in Figure Şekil 5.4. First 5 is compared with the root node 8, since 5 is smaller than 8 we continue with the left subtree of 8. In the second step 5 is compared with 4, since 5 is larger than 4 we continue with the right subtree of 4. In the third step we compare 5 with 6, since 5 is smaller than 6 we continue with the left subtree of 6. In the fourth step we compare 5 with 5 and we have found the searched 5. Note that, each step is similar to each step in the binary search, since the number of possible comparable numbers decrease by half.

5.2. BINARY SEARCH TREE OPERATIONS

5.2.2 Minimum and Maximum Elements

Not only searching a specific value but also searching the minimum and maximum elements can be done in logarithmic time (if the tree is balanced) in binary search trees.

Table 5.5: Iterative algorithm that finds the minimum element in a binary search tree

<pre> TreeNode* TreeNode::iterativeMinSearch(){ TreeNode* result = this; while (result . left != nullptr) result = result . left ; return result ; } </pre>	<pre> TreeNode iterativeMinSearch(){ TreeNode result = this; while (result.left != null) result = result . left ; return result; } </pre>
--	---

Table 5.6: Recursive algorithm that finds the minimum element in a binary search tree

<pre> TreeNode* TreeNode::recursiveMinSearch(){ if (left == nullptr) return this; else return left.recursiveMinSearch(); } </pre>	<pre> TreeNode recursiveMinSearch(){ if (left == null) return this; else return left.recursiveMinSearch(); } </pre>
--	--

Tables 5.5 and 5.6 show the algorithms that find the minimum element in a binary search tree non-recursive and recursive respectively. In order to find the minimum element, it is enough to traverse the tree always in the left direction until we see NULL (Lines 3-4). For example, Figure 5.7 shows the search of the minimum element in the tree given in Figure 5.4. Starting from the root node 8, we go left and visit nodes 4, 2, and 1 in that order.

Similarly Table 5.7 and 5.8 show the algorithms that find the maximum element in a binary search tree non-recursive and recursive respectively. In order to find the maximum element, it is enough to traverse the tree always in the right direction until we see NULL (Lines 3-4). For example, Figure 5.8 shows the search of the maximum element in the tree given in Figure 5.4. Starting from the root node 8, we go right and visit nodes 12, 14, and 15 in that order.

CHAPTER 5. BINARY SEARCH TREES

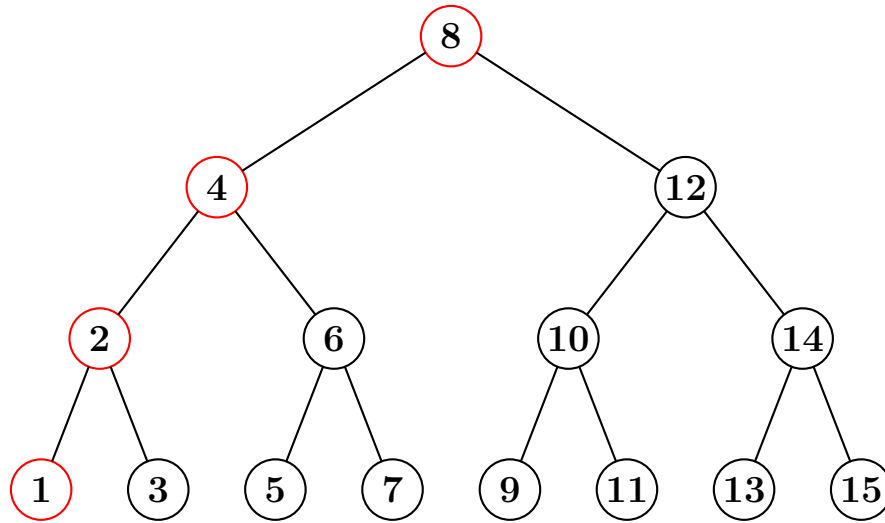


Figure 5.7: Searching for the minimum element in a binary search tree

Table 5.7: Iterative algorithm that finds the maximum element in a binary search tree

<pre> TreeNode* TreeNode::iterativeMaxSearch(){ TreeNode* result = this; while (result . right != nullptr) result = result . right ; return result ; } </pre>	<pre> TreeNode iterativeMaxSearch(){ TreeNode result = this; while (result.right != null) result = result.right ; return result; } </pre>
---	--

Table 5.8: Recursive algorithm that finds the maximum element in a binary search tree

<pre> TreeNode* TreeNode::recursiveMaxSearch(){ if (right == nullptr) return this; else return right .recursiveMaxSearch(); } </pre>	<pre> TreeNode recursiveMaxSearch(){ if (right == null) return this; else return right.recursiveMaxSearch(); } </pre>
---	---

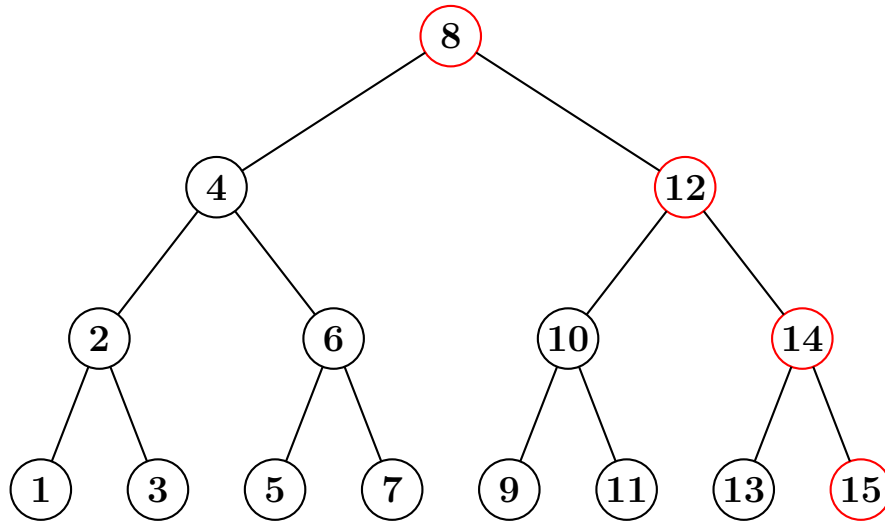


Figure 5.8: Searching for the maximum element in a binary search tree

5.2.3 Insertion

Table 5.9 shows the algorithm that add a new node into a binary search tree. We represent current node with variable x , and the parent node of x with variable y .

In order to add a new node into a binary search tree, we need to first find out the place where we will insert the new node. For this, we start from the root node (Line 3) and traverse the tree down. At each step, we compare the value of the new node with the value of the current node. If the value of the new node is smaller than the value of the current node (Line 6), the new node will be inserted into the left subtree of the current node. To accomplish this, we continue the process with the left child of the current node (Line 7). If the situation is reverse, that is, if the value of the new node is larger than the value of the current node (Line 8), the new node will be inserted into the right subtree of the current node. In this case, we continue the process with the right child of the current node (Line 9). In order to make easy of the arrangement of the links, we also store the parent node of the current node while we are traversing the tree down manner (Line 5).

When we find the place where we will insert the new node, current node will show NULL. But since we already store the parent node, if the value of the new node is smaller than the value of the parent node (Line 14) we insert

CHAPTER 5. BINARY SEARCH TREES

Table 5.9: The algorithm that adds a new node into the binary search tree

<pre> void Tree::insert (TreeNode* node){ TreeNode* y = null; TreeNode* x = root; while (x != nullptr){ y = x; if (node->data < x->data) x = x->left; else x = x->right; } if (y == nullptr) root = node; else if (node->data < y->data) y->left = node; else y->right = node; } </pre>	<pre> void insert(TreeNode node){ TreeNode y = null; TreeNode x = root; while (x != null){ y = x; if (node.data < x.data) x = x.left ; else x = x.right; } if (y == null) root = node; else if (node.data < y.data) y.left = node; else y.right = node; } </pre>
--	---

the new node as the left child of the parent node (Line 15), if the value of the new node is larger than the value of the parent node (Line 16) we insert the new node as the right child of the parent node (Line 17). If there is no parent node (Line 11), which will mean there is no node in the tree yet, we set the new node as the root node of the tree (Line 12).

Figure 5.9 shows the stages while inserting 13 into an example binary search tree consisting of 8 nodes. The nodes that are traversed while inserting 13 are marked with red. First 13 is compared with 12. Since 13 is larger than 12, we move to the right child of 12. Then 18 is compare with 13. Since 13 is smaller than 18, we continue with the left child of 18, namely 15. Since 15 is larger than 13, we move to the left child of 15. Since there is no left child, we place the new node to its place.

5.2.4 Deletion

Similar to other data structures, deletion is the most complex operation in binary search trees. When the value of the to be deleted node is given, we first traverse the tree down manner to find that node.

Table 5.10 shows the algorithm that removes a node from a binary search

5.2. BINARY SEARCH TREE OPERATIONS

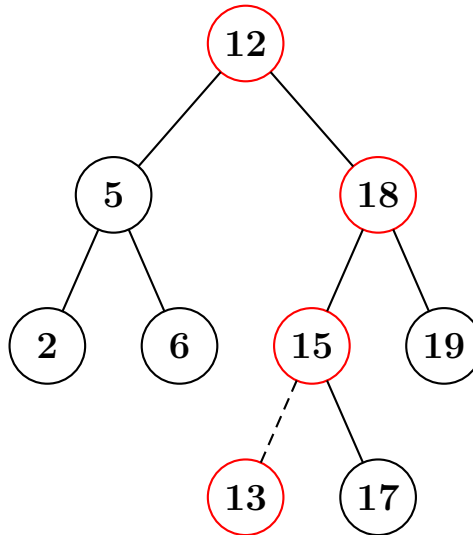


Figure 5.9: Inserting 13 into an example binary search tree. The nodes that are traversed while inserting 13 are marked with red

tree. Note that, the first **while** part of the algorithm is very similar to the part of finding out the place where a new node will be added. Traversing in down manner in deletion is the same as in insertion. We compare the value to be deleted with the value of the current node. If the value to be deleted is smaller than the value of the current node (Line 4) the search operation continues with the left child (Line 5). If the value to be deleted is larger than the value of the current node (Line 6) the search operation continues with the right child (Line 7).

When node to be deleted is found, it must be removed from the tree. If there are no children of the node to be deleted, there is no problem when it is removed. But if it has at least one child, when the remove the link to that node, we can not access the child and the other descendants of the deleted node. What we should do is to put one of its ascendants into its place. But which one? We need to find such a node that, when we replace the deleted node with that node, we will not destroy the binary search tree property of the tree. This is only possible, if we replace the deleted node with the maximum node of the left subtree or the minimum node of the right subtree. But when we replace the deleted node with that node, with which node will replace that replaced node? The same rule also applies here. Again we will

CHAPTER 5. BINARY SEARCH TREES

Table 5.10: The algorithm that deletes a node from the binary search tree

<pre> void Tree:: delete(int value){ TreeNode *y, *x = root, parent; while (x->data != value){ parent = x; if (x->data > value) x = x->left; else x = x->right; } parent = getParent(x); while (true){ if (x->left != nullptr){ y = x->left->recursiveMaxSearch(); parent = getParent(y); } else { if (x->right != nullptr){ y = x->right->recursiveMinSearch(); parent = getParent(y); } else { if (parent == nullptr) root = nullptr; else if (parent->left == x) parent->left = nullptr; else parent->right = nullptr; break; } } x->data = y->data; x = y; } } </pre>	<pre> void delete(int value){ TreeNode y, x = root, parent; while (x.data != value){ parent = x; if (x.data > value){ x = x.left; } else x = x.right; } parent = getParent(x); while (true){ if (x.left != null){ y = x.left.recursiveMaxSearch(); parent = getParent(y); } else { if (x.right != null){ y = x.right.recursiveMinSearch(); parent = getParent(y); } else { if (parent == null) root = null; else if (parent.left == x) parent.left = null; else parent.right = null; break; } } x.data = y.data; x = y; } } </pre>
---	---

look to the left and right subtrees of that node etc. If the replaced node does not have any children, the replacement process will finish. When the second **while** loop iterates, at each loop we find one maximum (Line 10) or one minimum (Line 12) node and replace the deleted node with that minimum or maximum node (Line 15) and set the to be deleted node that replaced node (Line 16).

5.2. BINARY SEARCH TREE OPERATIONS

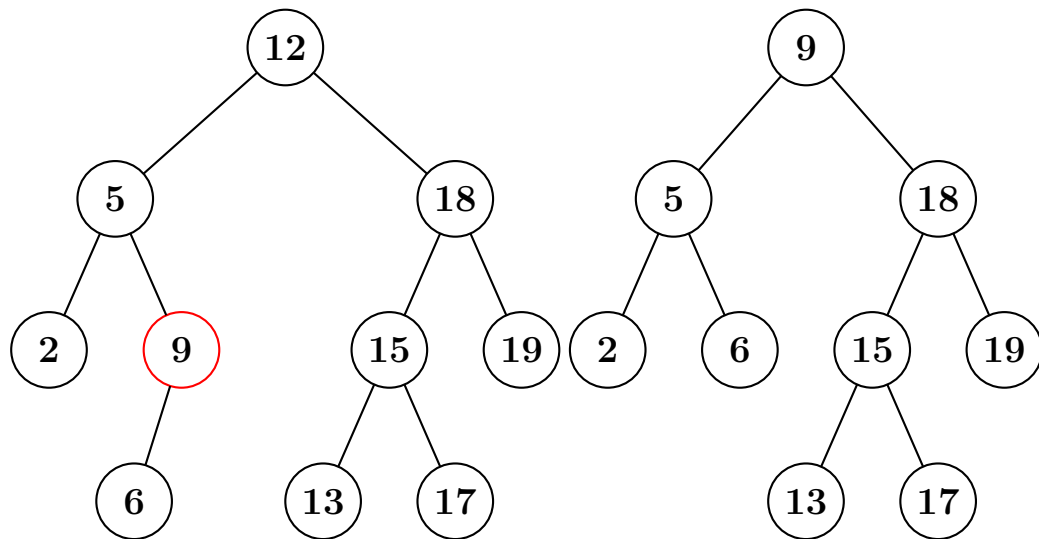


Figure 5.10: Deleting the root node of an example binary search tree

Figure 5.10 shows deleting the root node 12 from a given binary search tree. The element that will replace the root node is the maximum element of the left subtree of the root node, namely 9. But when we delete 9, which element will replace it? The maximum node in the left subtree of 9, namely 6. Since 6 has no children, the replacement process will finish there. The second part of the Figure shows the tree after the deletion operation.

Binary Search Tree Operations

Search: $\mathcal{O}(\log N)$

Insertion: $\mathcal{O}(\log N)$

Deletion: $\mathcal{O}(\log N)$

5.3 Traversals

If we want to generate an array from the elements in the binary search tree or write the elements to the screen, we need to visit the elements in the tree one by one and save these elements while visiting. We can traverse a tree in one of these three ways: Preorder traversal, inorder traversal, and postorder traversal. Visiting a node is composed of three parts: Visiting the node itself, traversing the left subtree of the node and traversing the right subtree of the node. Different orderings of these three parts will give us different traversals. As a rule of thumb, the left subtree is always traversed before the right subtree. In this case, the number of traversals will be 3. In the order of traverse,

- Parent, Left child, Right child
- Left child, Parent, Right child
- Left child, Right child, Parent

will give us all possible traversals. Note that, in the first case, the parent is visited before its children, in the second case the parent is visited in the middle of its children, and in the third case the parent is visited after its children. That visit order of the parent shows where the names of the traversals come from. The first type of traversal is called preorder traversal, second type of traversal is called inorder traversal and the third type of traversal is called postorder traversal.

Table 5.11: Preorder traversal algorithm

<pre>void TreeNode::preorder(){ cout << data; if (left != nullptr) left ->preorder(); if (right != nullptr) right ->preorder(); }</pre>	<pre>void preorder(){ System.out.print(data); if (left != null) left .preorder(); if (right != null) right .preorder(); }</pre>
--	---

The preorder traversal algorithm is given in Table 5.11, the inorder traversal is given in Table 5.12 and the postorder traversal algorithm is given

Table 5.12: Inorder traversal algorithm

<pre> void TreeNode::inorder(){ if (left != nullptr) left ->inorder(); cout << data; if (right != nullptr) right->inorder(); } </pre>	<pre> void inorder(){ if (left != null) left .inorder (); System.out.print(data); if (right != null) right .inorder (); } </pre>
--	---

Table 5.13: Postorder traversal algorithm

<pre> void TreeNode::postorder(){ if (left != nullptr) left ->postorder(); if (right != nullptr) right->postorder(); cout << data; } </pre>	<pre> void postorder(){ if (left != null) left .postorder(); if (right != null) right .postorder(); System.out.print(data); } </pre>
--	---

in Table 5.13. Note that, the place of the write function is the only change in the algorithm. This function represents what we do when we visit that node, namely writing the data of the node to the screen.

Figure 5.11 shows the visit order of the nodes of different traversal algorithm for the binary search tree given in Figure 5.2. The number on the upper left side of each node represents the visit order of that node.

In the preorder traversal, since we visit the parent node first, the root node is visited first, then the nodes on the left subtree of the root node (repeating the same ordering), and then the nodes on the right subtree of the root node (again repeating the same ordering) are visited and we get the numbers 6, 3, 2, 5, 7, 8 in that order.

In the inorder traversal, first the left subtree of the root node, then the root node, then the right subtree of the root node is visited and we get the numbers 2, 3, 5, 6, 7, 8 in that order. Note that, it is possible to sort the numbers in a binary search tree by using the in order traversal.

In the postorder traversal, first the left subtree of the root node, then the right subtree of the root node, then the root node is visited and we get the numbers 2, 5, 3, 8, 7, 6 in that order.

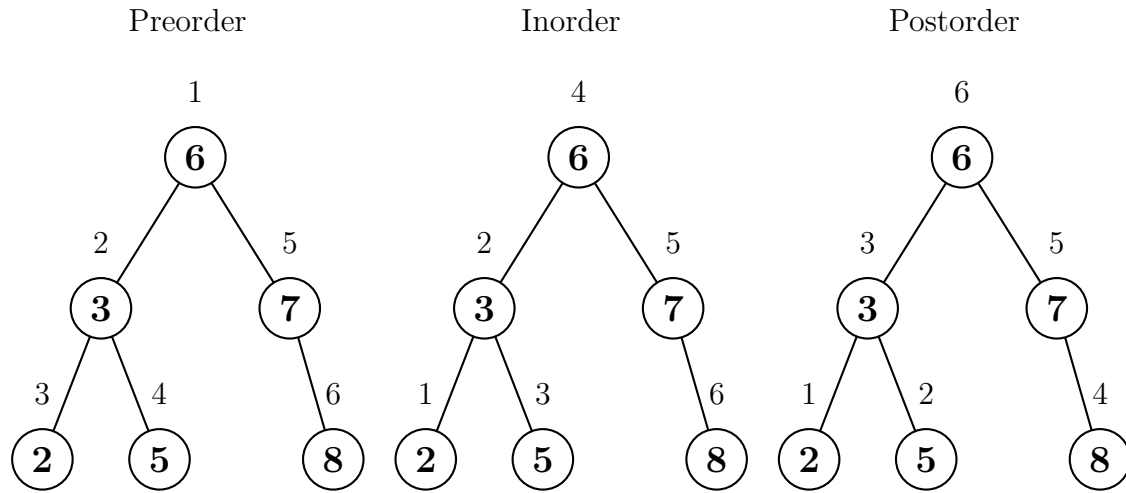


Figure 5.11: The visit order of the nodes of different traversal algorithm in binary search tree

5.4 Non-recursive Traversal

In some cases writing a recursive function may be easy, but in some other cases writing a non-recursive function may be more easier. Also, due to the nature of the problem, it may not be possible to write a recursive function. In these cases, we can use stack or queues data structures to process recursively defined data structures (such as binary search trees). For both cases, the elements of a stack or a queue correspond to a node of the tree.

We modify the element structure so that it contains tree node as the data field. Since given a tree node we can access any descendants of that node, actually the new element structure not only contains a tree node but a subtree rooted at that tree node.

Using this new structure, we can solve many problems in binary search tree non-recursively. For example, Table 5.14 shows the algorithm that finds the number of nodes in a binary search tree using stack data structure. First, we construct the stack that will hold the tree nodes in memory (Line 6). The first element that will be added to the stack is the root node (Line 7). If the tree has a root node (Line 8), this node is encapsulated inside an element (Line 9) and added to the stack (Line 10).

5.4. NON-RECURSIVE TRAVERSAL

Table 5.14: The algorithm that finds the node count in a binary search tree (Stack)

```

int nodeCountWithStack(){
    TreeNode* d;
    Node* e;
    Stack c;
    int count = 0;
    c = Stack();
    d = root;
    if (d != nullptr){
        e = new Node(d);
        c.push(e);
    }
    while (!c.isEmpty()){
        e = c.pop();
        d = e->data;
        count++;
        if (d->left != nullptr){
            e = new Node(d->left);
            c.push(e);
        }
        if (d->right != nullptr){
            e = new Node(d->right);
            c.push(e);
        }
    }
    return count;
}

```

```

int nodeCountWithStack(){
    TreeNode d;
    Node e;
    Stack c;
    int count = 0;
    c = new Stack();
    d = root;
    if (d != null){
        e = new Node(d);
        c.push(e);
    }
    while (!c.isEmpty()){
        e = c.pop();
        d = e.data;
        count++;
        if (d.left != null){
            e = new Node(d.left);
            c.push(e);
        }
        if (d.right != null){
            e = new Node(d.right);
            c.push(e);
        }
    }
    return count;
}

```

As long as the stack is not empty (Line 12), we pop elements from the stack (Line 13). Since each element of the stack corresponds to a tree node (Line 14), the variable **count**, which represents the number of nodes, is increased by one (Line 15). As a last step, the left and right subtrees of the current node, if they exist (Line 16, 20), are encapsulated inside an element (Line 17, 21), and pushed into the stack (Line 18, 22). In this way, all nodes of the binary search tree are pushed into the stack, popped from the stack, and counted once.

We can also solve the same problem using queue data structure instead of stack data structure. For example, Table 5.15 shows the algorithm that finds the number of nodes in a binary search tree using queue data structure.

CHAPTER 5. BINARY SEARCH TREES

Table 5.15: The algorithm that finds the node count in a binary search tree (Queue)

<pre> int nodeCountWithQueue(){ TreeNode* d; Node* e; Queue k; int count = 0; k = Queue(); d = root; if (d != nullptr){ e = new Node(d); k.enqueue(e); } while (!k.isEmpty()){ e = k.dequeue(); d = e->data; count++; if (d->left != nullptr){ e = new Node(d->left); k.enqueue(e); } if (d->right != nullptr){ e = new Node(d->right); k.enqueue(e); } } return count; } </pre>	<pre> int nodeCountWithQueue(){ TreeNode d; Node e; Queue k; int count = 0; k = new Queue(); d = root; if (d != null){ e = new Node(d); k.enqueue(e); } while (!k.isEmpty()){ e = k.dequeue(); d = e.data; count++; if (d.left != null){ e = new Node(d.left); k.enqueue(e); } if (d.right != null){ e = new Node(d.right); k.enqueue(e); } } return count; } </pre>
--	--

First, we construct the empty queue that will hold the tree nodes in memory (Line 6). The first element that will be added to the queue is the root node (Line 7). If the tree has a root node (Line 8), this node is encapsulated inside an element (Line 9) and added to the queue (Line 10).

As long as the queue is not empty (Line 12), we dequeue elements from the queue (Line 13). Since each element of the queue corresponds to a tree node (Line 14), the variable **count**, which represents the number of nodes, is increased by one (Line 15). As a last step, the left and right subtrees of the current node, if they exist (Line 16, 20), are encapsulated inside an element (Line 17, 21), and enqueued into the queue (Line 18, 22). In this way, all nodes of the binary search tree are enqueued into the queue, dequeued from

the queue, and counted once.

5.5 AVL Trees

As we have mentioned in the beginning of this chapter, as long as the binary search tree is balanced, the three basic operations, searching an element in the tree, inserting a new element into the tree, and removing an element from the tree have the same time complexity, $\mathcal{O}(\log N)$. On the other hand, if the tree is unbalanced, this time complexity can increase to $\mathcal{O}(\sqrt{N})$ and sometimes to $\mathcal{O}(N)$.

Figure 5.12 shows three different binary search trees which are constructed by inserting numbers 1 through 7 into the tree in different orders. The binary search tree shown in Figure 5.12(a) is constructed by inserting numbers 4, 2, 6, 1, 3, 5, 7 in that order. As can be seen, the tree is fully balanced. Searching any one of the numbers 1, 3, 5, 7 stored in the leaves requires at most 3 operations. The binary search tree shown in Figure 5.12(b) is constructed by inserting numbers 4, 3, 5, 2, 6, 1, 7 in that order. The depth of this tree is 4 and is more unbalanced than the tree shown in Figure 5.12(a). As the last case, we see the most unbalanced binary search tree in Figure 5.12(c). If we insert the same numbers in the reverse sorted order (7, 6, 5, 4, 3, 2, 1) into an empty binary search tree, we get that situation. The depth of this tree is 7 and any one of its nodes has only left child. Similar unbalanced tree can be generated when we insert also the same numbers in increasing order (1, 2, 3, 4, 5, 6, 7). The nodes in such a tree has only right child. Since we can not know in which order the numbers will be inserted into the tree, in practice we will encounter any one of these trees and naturally we prefer the balanced trees.

In order to solve this problem, one needs to insist on balanced trees, that is, when the tree loses its balance, we need to rebalance it. There are many balanced tree structures in the computer science literature. AVL tree is one of the oldest member of these balanced tree structures.

5.5.1 Definition

AVL (**A**delson-**V**elskii and **L**andis) tree is a balanced binary search tree structure. The balance property is very simple and ensures the depth of the

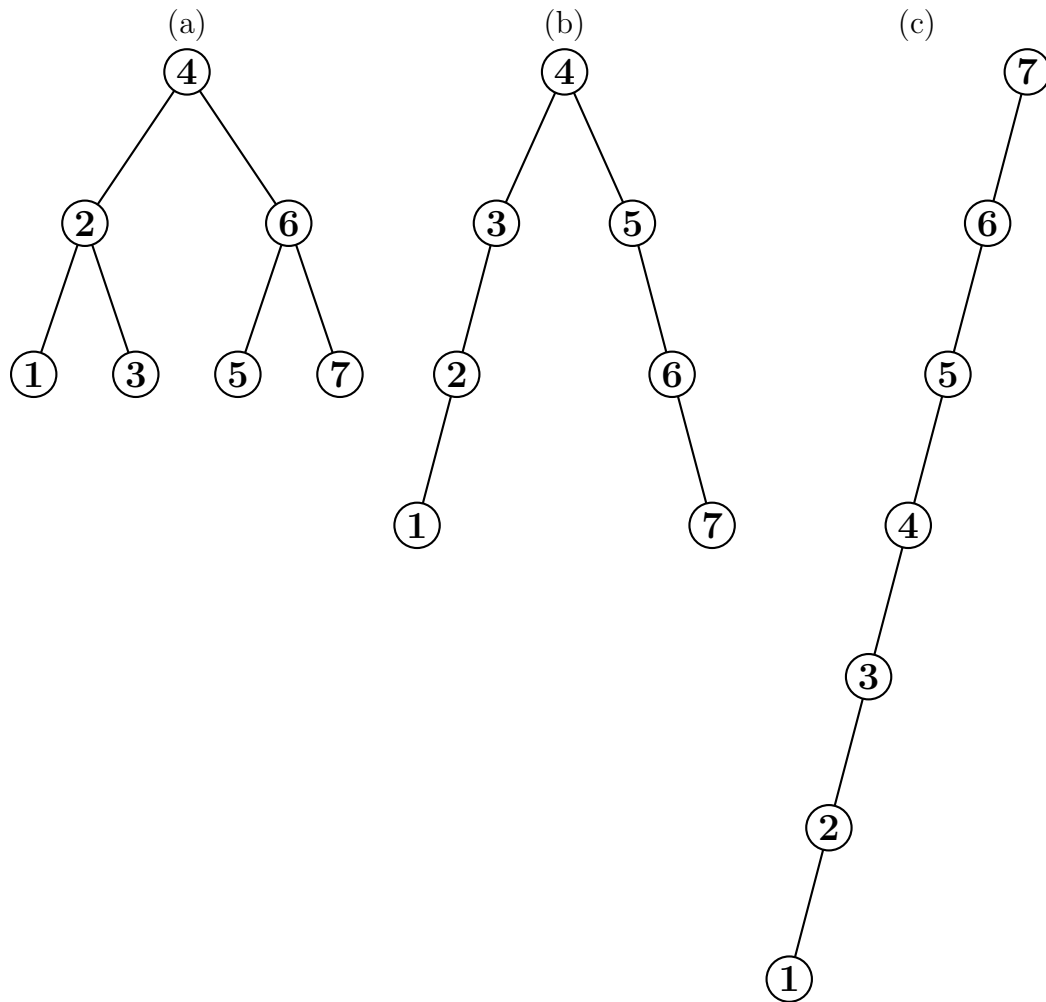


Figure 5.12: Three different binary search trees which are constructed by inserting numbers 1 through 7 into the tree in different orders

tree is in the level of $\mathcal{O}(\log N)$.

In AVL tree, the heights of the left and right subtrees of each node can differ by at most 1. If the heights of the left and right subtrees of a single node differ by more than 1, then that binary search tree is not an AVL tree. Figure 5.13 shows two binary search trees. The binary search tree on the left hand side is an AVL tree, but the one on the right side isn't. In Figure 5.13(a)

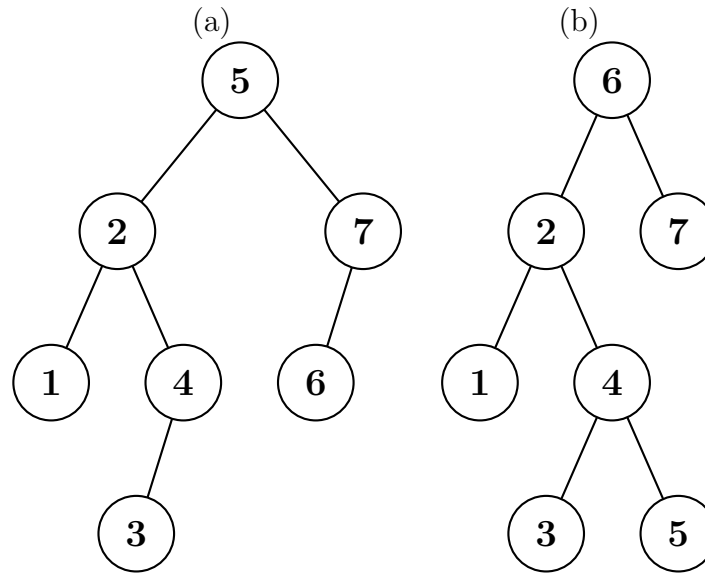


Figure 5.13: Binary search trees. (a) is an AVL tree (b) is not AVL tree.

- The height of the left subtree of the node with data 5 is 3, the height of the right subtree of the node with data 5 is 2, $3 - 2 \leq 1$
- The height of the left subtree of the node with data 2 is 1, the height of the right subtree of the node with data 2 is 2, $2 - 1 \leq 1$
- The height of the left subtree of the node with data 4 is 1, the height of the right subtree of the node with data 4 is 0, $1 - 0 \leq 1$
- The height of the left subtree of the node with data 7 is 1, the height of the right subtree of the node with data 7 is 0, $1 - 0 \leq 1$

satisfies the AVL tree property. On the other hand, in Figure 5.13(b) the height of the left subtree of the node with data 6 is 3, the height of the right subtree of the node with data 6 is 1 and their difference is $3 - 1 = 2 > 1$, which does not satisfy AVL tree property.

Table 5.16 shows the definition of any AVL tree node. The AVL node data structure, is defined as the usual node data structure. Each node contains a data field, two links pointing to the left and right children and a field showing the height of that node. Since both of these links are themselves AVL tree nodes, the definition of AVL node is also recursive.

CHAPTER 5. BINARY SEARCH TREES

Table 5.16: Definition of an AVL tree node which contains integers

<pre> class AvlNode{ private: int data; int height; AvlNode* left; AvlNode* right; public: AvlNode(int data); } AvlNode::AvlNode(int data){ this->data = data; left = nullptr; right = nullptr; height = 1; } </pre>	<pre> public class AvlNode{ int data; int height; AvlNode left; AvlNode right; public AvlNode(int data){ this.data = data; left = null; right = null; height = 1; } } </pre>
---	--

Table 5.17: Definition of AVL tree

<pre> class AvlTree{ private: AvlNode* root; public: AvlTree(); int height(AvlNode* d); } AvlTree::AvlTree(){ root = nullptr; } int AvlTree::height(AvlNode* d){ if (d == nullptr) return 0; else return d->height; } </pre>	<pre> public class AvlTree{ AvlNode root; public AvlTree(){ root = null; } int height(AvlNode d){ if (d == null) return 0; else return d.height; } } </pre>
---	---

Table 5.17 shows the definition of AVL tree which uses the definition of AVL node above. Since the definition of AVL node is recursive, it can define a tree alone. What we only need is the same as in the linked lists, only a

link pointing to the root node of the AVL tree. That is **root** in the definition above. Since the tree is empty at the beginning stage, the **root** is NULL.

5.5.2 Rotations

When we add a new node to the AVL tree we need to (i) modify the **height** field of all nodes from the new node to the root node (ii) restore the AVL tree property if the tree does not satisfy AVL tree property. For example, if we add a new node as the left or right child of 6, 3, or 1 to the AVL tree shown in the Figure 5.13, the tree will not satisfy the AVL tree property. In order to restore the AVL tree property, it is enough to do a simple rotation.

Let say node d does not satisfy the AVL tree property. Since each node in a binary search tree can have at most two children, the unbalanced tree is only possible if the heights of two subtrees of d differ by 2. Here, four different cases are possible:

- when we insert the left subtree of the left child of d
- when we insert the right subtree of the left child of d
- when we insert the left subtree of the right child of d
- when we insert the right subtree of the right child of d

Since the first and fourth cases and second and third cases are symmetries of each other, there are basically two cases possible. We can restore the AVL tree property for the first and fourth cases with a single rotation, for the second and third cases with a double rotation.

Single Rotation

Figure 5.14 shows the single rotation that solves the first case, Table 5.18 shows the application of that single rotation algorithm. In Figure 5.18(a), node k_2 does not satisfy AVL tree property. The height of the left subtree is larger than the height of the right subtree by 2. In order to restore the AVL tree property, we move node k_1 one level up, since due to the binary search tree property $k_2 > k_1$, we move node k_2 one level down (Figure 5.18(b)). The links updated are:

- Since $k_2 > B > k_1$, the left child of node k_2 is now the old right child of k_1 (Line 3).

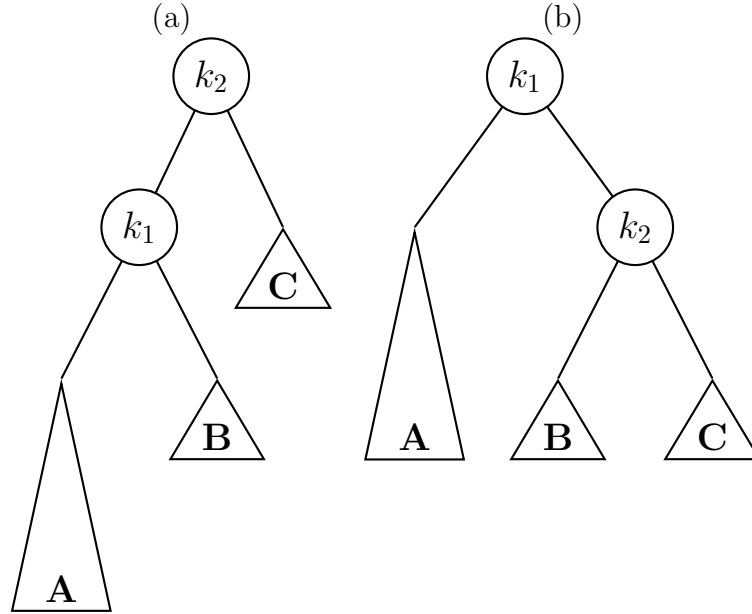


Figure 5.14: Single rotation to solve case 1

- The right child of k_1 is now k_2 (Line 4).

Note that, the root node of the subtree is now k_1 . In order to modify the parent link of k_2 , the new root of the subtree is returned by the function.

Figure 5.15 shows the single rotation that solves the fourth case, Table 5.19 shows the application of that single rotation algorithm. In Figure 5.19(a), node k_1 does not satisfy AVL tree property. The height of the right subtree is larger than the height of the left subtree by 2. In order to restore the AVL tree property, we move node k_2 one level up, since due to the binary search tree property $k_2 > k_1$, we move node k_1 one level down (Figure 5.19(b)). The links updated are:

- Since $k_2 > B > k_1$, the right child of node k_1 is now the old left child of k_2 (Line 3).
- The left child of k_2 is now k_1 (Line 4).

Note that, the root node of the subtree is now k_2 . In order to modify the parent link of k_1 , the new root of the subtree is returned by the function.

Table 5.18: The single rotation algorithm to solve case 1

```

AvlNode* AvlTree::rotateLeft (AvlNode* k2){
    AvlNode* k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max(height(k2->left), height(k2->right)) + 1;
    k1->height = max(height(k1->left), k1->right->height) + 1;
    return k1;
}

```

```

AvlNode rotateLeft(AvlNode k2){
    AvlNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = Math.max(height(k2.left), height(k2.right)) + 1;
    k1.height = Math.max(height(k1.left), k1.right.height) + 1;
    return k1;
}

```

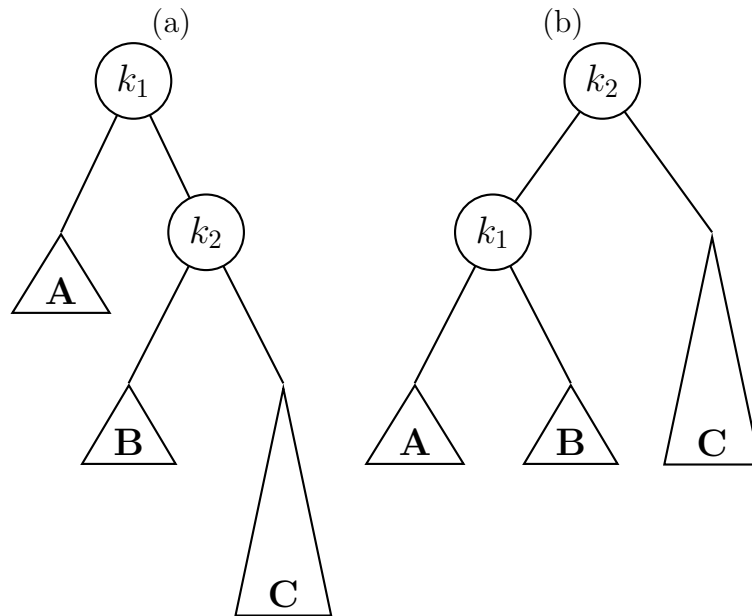


Figure 5.15: Single rotation to solve case 4

CHAPTER 5. BINARY SEARCH TREES

Table 5.19: Single rotation algorithm to solve case 4

```

AvlNode* AvlTree::rotateRight(AvlNode* k1){
    AvlNode* k2 = k1->right;
    k1->right = k2->left;
    k2->left = k1;
    k2->height = max(k2->left->height, height(k2->right)) + 1;
    k1->height = max(height(k1->left), height(k1->right)) + 1;
    return k2;
}

```

```

AvlNode rotateRight(AvlNode k1){
    AvlNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k2.height = Math.max(k2.left.height, height(k2.right)) + 1;
    k1.height = Math.max(height(k1.left), height(k1.right)) + 1;
    return k2;
}

```

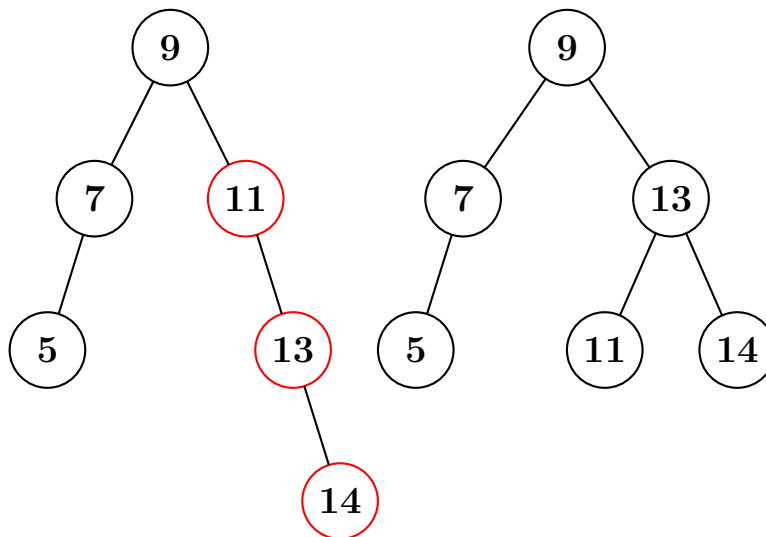


Figure 5.16: An example single rotation to solve case 4

In the left tree given in Figure 5.16, by adding 14 to the tree, node 11 does not satisfy the AVL tree property. In order to satisfy the AVL tree

property again, we only need to do single rotation. In this way, node 13 will be the parent of 11 and 14, and all nodes will satisfy the AVL tree property.

Double Rotation

Figure 5.17 shows the double rotation that solves the second case, Table 5.20 shows the application of this algorithm. In Figure 5.17(a), node k_3 does not satisfy AVL tree property. The height of the left subtree is larger than the height of the right subtree by 2. This time single rotation can not solve the problem.

In order to satisfy the AVL tree property again, in the first phase we will do single right rotation on the subtree rooted with k_1 (Line 2). With this rotation, the left child of node k_2 will be k_1 , whereas the right child of node k_1 will be B (the old left child of node k_2).

In the second phase, we will do single left rotation on the subtree rooted with k_3 (Line 3). With this rotation, the right child of node k_2 will be k_3 , whereas the left child of node k_3 will be C (the old right child of k_2).

Note that, the new root node of the subtree is now k_2 . In order to modify the parent link of k_3 , the new root of the subtree is returned by the function.

Table 5.20: The double rotation algorithm that solves case 2

<pre> AvlNode* AvlTree::doubleRotateLeft(AvlNode k3->left = rotateRight(k3->left); return rotateLeft(k3); } </pre>	<pre> AvlNode doubleRotateLeft(AvlNode k3){ k3.left = rotateRight(k3.left); return rotateLeft(k3); } </pre>
--	---

Figure 5.18 shows the double rotation that solves the third case, Table 5.21 shows the application of this algorithm. In Figure 5.18(a), node k_1 does not satisfy AVL tree property. The height of the right subtree is larger than the height of the left subtree by 2.

In order to satisfy the AVL tree property again, in the first phase we will do single right rotation on the subtree rooted with k_3 (Line 2). With this rotation, the right child of node k_2 will be k_3 , whereas the left child of node k_3 will be C (the old right child of node k_2).

In the second phase, we will do single right rotation on the subtree rooted with k_1 (Line 3). With this rotation, the left child of node k_2 will be k_1 , whereas the left child of node k_1 will be B (the old left child of k_2).

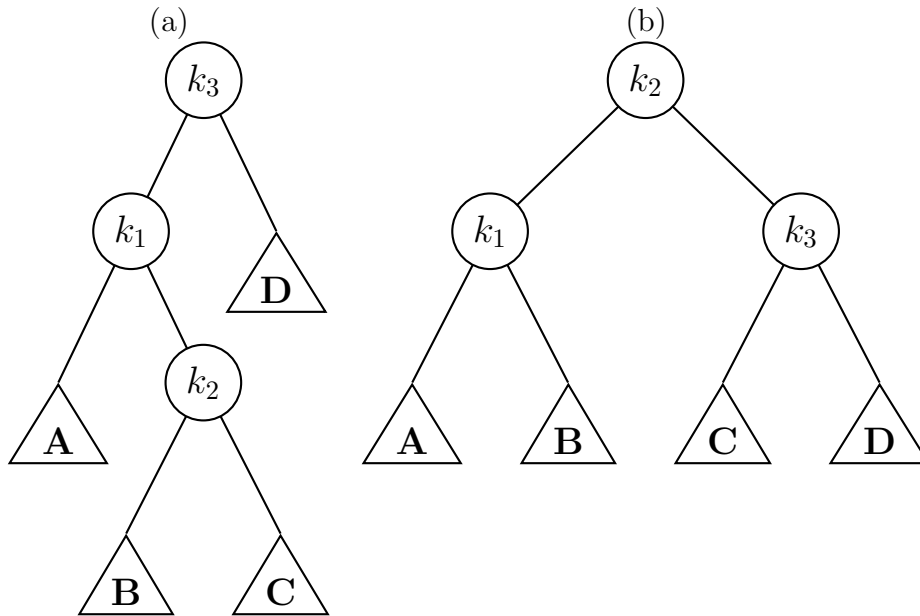


Figure 5.17: Double rotation that solves the case 2

Note that, the new root node of the subtree is now k_2 . In order to modify the parent link of k_1 , the new root of the subtree is returned by the function.

Table 5.21: The double rotation algorithm that solves case 3

<pre> AvlNode* AvlTree::doubleRotateRight(AvlNode k1){ k1->right = rotateLeft(k1->right); return rotateRight(k1); } </pre>	<pre> AvlNode doubleRotateRight(AvlNode k1){ k1.right = rotateLeft(k1.right); return rotateRight(k1); } </pre>
---	---

In the left tree given in Figure 5.19, by adding 12 to the tree, node 11 does not satisfy the AVL tree property. In order to satisfy the AVL tree property again, single rotation is not enough, we need to do right double rotation. In this way, node 12 will be the parent of 11 and 13, and all nodes will satisfy the AVL tree property.

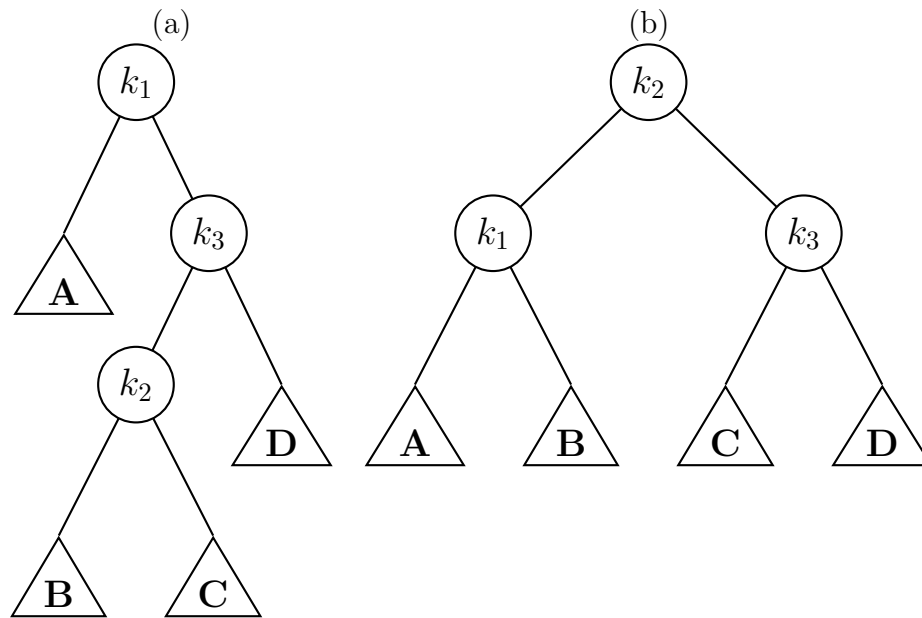


Figure 5.18: The double rotation that solves case 3

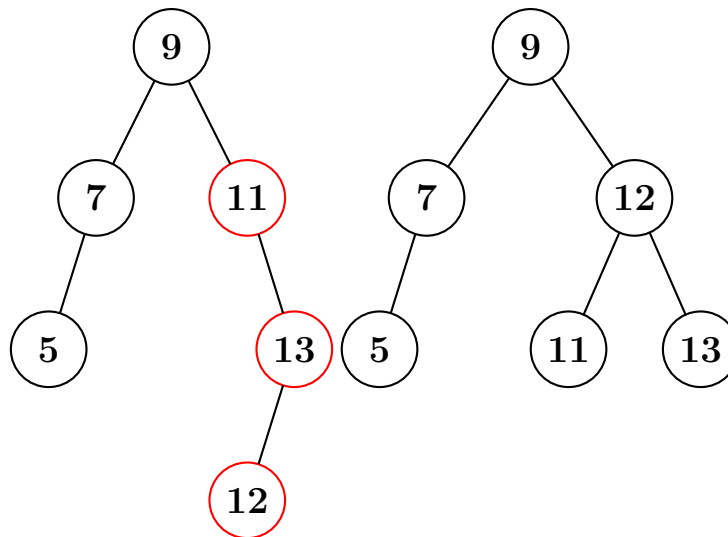


Figure 5.19: An example double rotation that is applied to solve case 3

5.5.3 Insertion

Table 5.23 shows the algorithm, which inserts a new node using the rotation functions defined above.

First we will proceed with the stages the same when we add a new node into a binary search tree. For this, we start from the root node (Line 2) and traverse in down manner. The current node we visit is represented with x and the previous node is represented with y . At each time we compare the value of the current node with the value of the new node (Line 9, 12). If the value of the new node is smaller than the value of the current node (Line 9), the new node will be inserted into the left subtree of the current node. For this, we will continue with the left child to process (Line 10). If the reverse is true, that is, if the value of the new node is larger than the value of the current node (Line 12), the new node will be inserted into the right subtree of the current node. In this case, we will continue with the right child to process (Line 13). At each step, we store the current node in a separate stack (Line 7).

When we insert a new node into an AVL tree, we need to change the heights of the nodes and check if the AVL tree property is satisfied or not. Only the height of the nodes, which we visit while we are finding the place for the new node, can be changed. So, what we should do is to pop those nodes from the stack one by one (Line 19) and change the heights of those nodes (Line 20).

Similarly the nodes, which we visit while we are finding the place for the new node, may no longer satisfy the AVL tree property. So what we should do is to pop those nodes from the stack one by one (Line 19) and calculate the difference of the heights of their left and right subtrees (Line 21). If the height difference is 2, the AVL tree property is not satisfied. If we added the new node into the left subtree of the left child (Line 22), we need to do left single rotation (Line 23), if we added into the right subtree of the left child (Line 24), we need to do left double rotation (Line 25), if we added into the left subtree of the right child (Line 26), we need to do right double rotation (Line 27), if we added into the right subtree of the right child (Line 28), we need to the right single rotation (Line 29). Since the root node of the subtree will be changed after a rotation, the new child of y will be the new root node t (Line 31).

Table 5.22: The algorithm that inserts a new node into an AVL tree (C++)

```

void AvlTree::insert(AvlNode* node){
    AvlNode* y = nullptr, *x = root, *t;
    Node* e;
    int dir1 = 0, dir2 = 0;
    Stack c = Stack(100);
    while (x != nullptr){
        y = x;
        e = new Node(y);
        c.push(e);
        dir1 = dir2;
        if (node->data < x->data){
            x = x->left;
            dir2 = LEFT;
        }else{
            x = x->right;
            dir2 = RIGHT;
        }
    }
    insertChild(y, node);
    while (!c.isEmpty()){
        e = c.pop();
        x = e->data;
        x->height = max(height(x->left), height(x->right)) + 1;
        if (fabs(height(x->left) - height(x->right)) == 2){
            if (dir1 == LEFT && dir2 == LEFT)
                t = rotateLeft(x);
            if (dir1 == LEFT && dir2 == RIGHT)
                t = doubleRotateLeft(x);
            if (dir1 == RIGHT && dir2 == LEFT)
                t = doubleRotateRight(x);
            if (dir1 == RIGHT && dir2 == RIGHT)
                t = rotateRight(x);
            e = c.pop();
            y = e->data;
            insertChild(y, t);
            break;
        }
    }
}

```

5.6 B+ Tree

In the computer science literature, ~~the~~ ¹⁷⁸ structures such as AVL tree, splay tree, red-black tree are proposed, which show the search tree property and

CHAPTER 5. BINARY SEARCH TREES

Table 5.23: The algorithm that inserts a new node into an AVL tree (Java)

```
void insert(AvlNode node){
    AvlNode y = null, x = root, t;
    Node e;
    int dir1 = 0, dir2 = 0;
    Stack c = new Stack(100);
    while (x != null){
        y = x;
        e = new Node(y);
        c.push(e);
        dir1 = dir2;
        if (node.data < x.data){
            x = x.left;
            dir2 = LEFT;
        }else{
            x = x.right;
            dir2 = RIGHT;
        }
    }
    insertChild(y, node);
    while (!c.isEmpty()){
        e = c.pop();
        x = e.data;
        x.height = Math.max(height(x.left), height(x.right)) + 1;
        if (Math.abs(height(x.left) - height(x.right)) == 2){
            if (dir1 == LEFT && dir2 == LEFT)
                t = rotateLeft(x);
            if (dir1 == LEFT && dir2 == RIGHT)
                t = doubleRotateLeft(x);
            if (dir1 == RIGHT && dir2 == LEFT)
                t = doubleRotateRight(x);
            if (dir1 == RIGHT && dir2 == RIGHT)
                t = rotateRight(x);
            e = c.pop();
            y = e.data;
            insertChild(y, t);
            break;
        }
    }
}
```

and also remain balanced after insertion and deletion operations. In the

previous section, we have given an example of these structures, namely AVL tree structure.

Another possibility of constructing a balanced tree structure is to store not only a single value but more than one value in a node. These type of tree structures are generalizations of the binary trees and called d -ary tree structures in the computer science literature. 2-3-4 trees, B-tree, B+ trees can be given as example d -ary tree structures. In this section, we will cover B+ trees, which is one of the d -ary tree structures and used often in database systems.

5.6.1 Definition

B+ tree is a dynamic search tree structure and consists of two parts, an index part and a data part. The index part is of d -ary tree structure, each node stores $d \leq m \leq 2d$ values. d is a parameter of B+ tree, shows the capacity of B+ tree and called as the degree of the tree. The root node is the single exception for this rule and can store $1 \leq m \leq 2d$ values. Each node also contains $m + 1$ links to point to its $m + 1$ child nodes. With the help of these links, the tree can be traversed in top-down manner. Let P_i represent the link pointing to the node i and K_i represent the i 'th value in the same node, the i 'th child and the ascendants of this child can take values between the interval $K_i \leq K < K_{i+1}$. The data are stored in the leaf nodes and due to the definition of a tree, the leaf nodes can not have children.

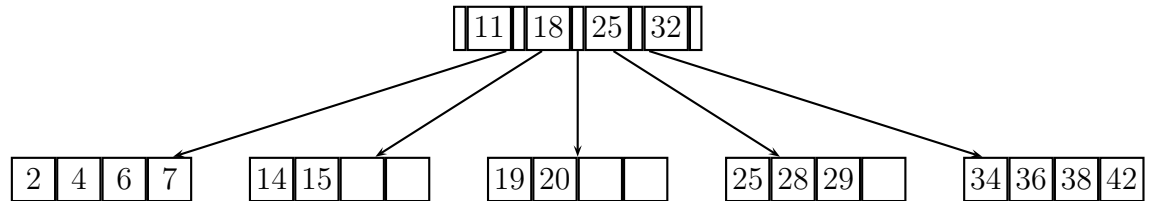


Figure 5.20: An example B+ tree of degree $d = 2$ storing 15 values

Figure 5.20 shows an example B+ tree of degree $d = 2$ which stores 15 values. Since $d = 2$, the nodes can store minimum 2, maximum 4 values. In the upper part there is only the root node, and stores 4 values (11, 18, 25, 32) and 5 links. In the lower part, in the interval $K < 11$ 4 values, in the interval $11 \leq K < 18$ 2 values, in the interval $18 \leq K < 25$ 2 values, in the

CHAPTER 5. BINARY SEARCH TREES

interval $25 \leq K < 32$ 3 values and lastly in the interval $K \geq 32$ 4 values are stored.

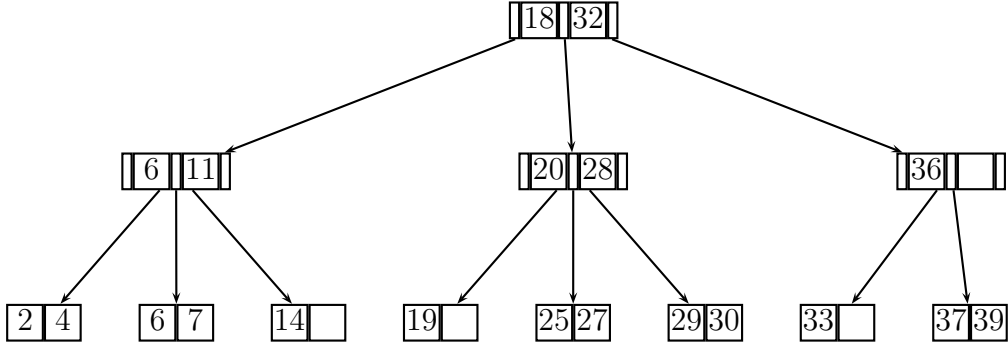


Figure 5.21: An example B+ tree of degree $d = 1$ storing 13 values

Figure 5.21 shows an example B+ tree of degree $d = 1$ which stores 13 values. In the upper part, including the root node, we have a total of 4 nodes. Since $d = 1$, the nodes can store minimum 1, maximum 2 values. Different than the tree in Figure 5.20, there is an empty node in this tree (the node storing 36), therefore compared to other nodes, this node has 2 children.

Table 5.24 shows the definition of any node of B+ tree. In a node order to store the values we will use an array field **K**, to store all links to point to all children we will use an array field **children**, to show if the node is a leaf node or not we will use the field **leaf** and to store the number of values we will use a counter field **m**. Since the child nodes themselves are also B+ nodes, the definition of a B+ node is recursive as the definition of an AVL node.

Table 5.25 shows the definition of B+ tree which uses the definition of B+ node given above. Since the definition of B+ node data structure is recursive, it can define a tree alone. What we need is a link to point to the root node of the B+ tree like in the linked list structure. That is **root** in the definition above. Since the tree is empty at the beginning stage, the root is NULL.

5.6.2 Search

Searching a specific element in B+ is similar to searching a specific element in binary search tree. The search, like in the binary search tree, is started from

Table 5.24: Definition of B+ tree node

<pre> class BNode{ private: int [] K; int m; int d; bool leaf; BNode*[] children; public: BNode(int d); ~BNode(); } BNode::BNode(int d){ m = 0; this->d = d; leaf = true; K = new int[2 * d + 1]; children = new BNode*[2 * d + 1]; } BNode::~~BNode(){ delete [] K; for (int i = 0; i < 2 * d + 1; i++) delete children[i]; delete [] children; } </pre>	<pre> public class BNode{ int [] K; int m; int d; boolean leaf; BNode[] children; public BNode(int d){ m = 0; this.d = d; leaf = true; K = new int[2 * d + 1]; children = new BNode[2 * d + 1]; } } </pre>
---	--

the root node and at each stage we go one level down. Compared to binary search tree, searching an element in B+ is different in two points. First, in binary search tree, the values can be in leaf or in non leaf nodes, whereas in B+ tree the values are only stored in the leaf nodes. Therefore, in the binary search tree the search can be finished anywhere in the tree, whereas in B+ tree the search will be finished only if we reach the leaf nodes. Second, in binary search tree, we compare the searched value with the value of the current node and continue the search with the left or right child, whereas since there are m values in B+ tree nodes, we need to compare the searched value with all or some of these values, and decide accordingly to continue with corresponding child.

Table 5.26 shows the algorithm that searches a given value in a B+ tree. We start searching from the root node (Line 4), the node with which we

CHAPTER 5. BINARY SEARCH TREES

Table 5.25: Definition of B+ tree

<pre> class BTree{ private: BNode* root; public: BTree(); ~BTree(); } BTree::BTree(){ root = nullptr; } BTree::~BTree(){ delete root; } </pre>	<pre> public class BTree{ BNode root; public BTree(){ root = null; } } </pre>
--	--

Table 5.26: The algorithm that searches a given value in B+ tree

<pre> BNode* BTree::search(int value){ int child; BNode* b; b = root; while (!b->eaf){ child = b->position(value); b = b->children[child]; } return b; } int BNode::position(int value){ int i; if (value > K[m - 1]) return m; else for (i = 0; i < m; i++) if (value < K[i]) return i; return -1; } </pre>	<pre> BNode search(int value){ int child; BNode b; b = root; while (!b.leaf){ child = b.position(value); b = b.children[child]; } return b; } int position(int value){ int i; if (value > K[m - 1]) return m; else for (i = 0; i < m; i++) if (value < K[i]) return i; return -1; } </pre>
--	---

compare the searched value at each stage is represented by **b** (Line 3) and we continue the search until we arrive the leaf nodes (Line 5). In order to

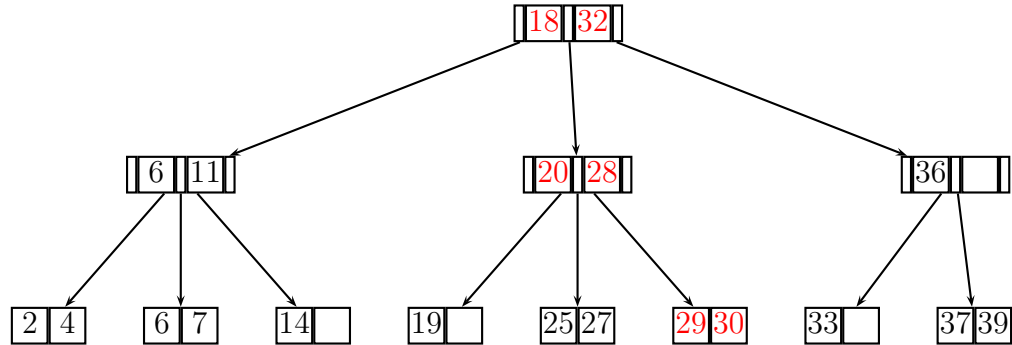


Figure 5.22: Searching 30 in the B+ tree given in Figure 5.21

understand the subtree of node **b** where our searched value resides, we need to compare the searched value with the values K_i . For this, the function named **position** is given. If the searched value is larger than the last value of node **b** (Line 13), we need to continue the search with the rightmost child (Line 14). If the searched value is smaller than the i . value of node **b** (Line 17), we need to continue the search with the i . child (Line 18). As a last step, the function returns the leaf node of node **b** (Line 9).

Figure 5.22 shows the visited nodes when we search 30 in an example tree (Figure 5.21). First 30 is compared with 18 and 32 in the root node, since 30 is larger than 18 and smaller than 32 we continue the search with the middle node. In the second step, 30 is compared with 20 and 28, since 30 is larger than both 20 and 28 we continue with the rightmost node. In the third step, since we come to the leaf node, the search is finished.

5.6.3 Insertion

When we insert a new value into a B+ tree, we may require to modify the tree structure like in AVL tree or we may just add it to a corresponding leaf node. Since the data are only stored in the leaves of B+ tree, we need to first find out the leaf node where we will insert the new value. After finding this leaf node, first the value is added to the leaf node, then if the number of values is more than the maximum number in a leaf node, the leaf node is divided into two and we add a new child node to the parent of the current leaf node. If adding a child node to the parent node results in that we exceed the maximum number of child nodes, the parent node is divided

CHAPTER 5. BINARY SEARCH TREES

into two and we add a new child node to the grandparent of the leaf node. This process continues until we do not exceed the number of child nodes in a parent node when we add a new child node. For this reason, we will examine the insertion algorithm in two phases. In the first phase, we will talk about the algorithm that adds a node into the first part, the index part, of B+ tree, and in the second phase, we will talk about the algorithm that adds a value in the second part, the data part, of B+ tree.

Table 5.28 shows the algorithm that add a new node to the index part of B+ tree. First the function **position** is used to determine the node or the subtree to which the new node will be added (Line 4). If this subtree is a leaf node, we call the function **insertLeaf** that will add the value to a leaf node (Line 8). If this subtree is not a leaf node the function calls itself with the determined subtree (Line 6). Both **insertNode** and **insertLeaf** functions, if adding a new value/node to that node/subtree necessitates a new child node to be added to the parent node, they will both return the new added node and the node obtained by dividing the original node. If there is not such a restructuring, these functions will return NULL (Line 10). If we add a new child node to the parent node, first we open a space for that child node in the value array **K** (Lines 11-12), than we add this new node to the array **K** (Line 13). After adding there are two possibilities:

- After inserting the new child node, the current node did not exceed its capacity (Lines 14-19). In this case, we open space for the link, which points to the new node, in the array **children** (Lines 15-16) and place that link inside of this array (Line 17).
- After inserting the new child node, the current node exceed its capacity (Lines 20-36). In this case, we need to create **newNode** (Line 21), transfer the last d values (Lines 22-23) and the last d links of the current node (Lines 24-25) to the **newNode**. As a last case, if the divided node is the root node (Line 28), we need to create a new root node (Line 29) and the first child of this new root node will be **b** (Line 31), and the second child of the new root node will be **newNode** (Line 32).

Table 5.29 shows the algorithm that add a new value to the data part of B+ tree. First the function **position** is used to determine the position where the new value will be placed (Line 4). Then we open a space for that value in the value array **K** (Lines 5-6), than we add this new value to the array

Table 5.27: The algorithm that adds a new node into the index part of B+ tree (C++)

```

BNode* BNode::insertNode(BTree a, int value){
    BNode* s, *newNode;
    int child, i;
    child = position(value);
    if (!children[child]->leaf)
        s = children[child]->insertNode(a, value);
    else
        s = children[child]->insertLeaf(value);
    if (s == nullptr)
        return nullptr;
    for (i = m; i > child; i--)
        K[i] = K[i - 1];
    K[child] = s->K[2 * d];
    if (m < 2 * d){
        for (i = m + 1; i > child; i--)
            children[i] = children[i - 1];
        children[child] = s;
        m++;
        return nullptr;
    } else {
        newNode = new BNode(d);
        for (i = 0; i < d; i++)
            newNode->K[i] = K[d + i + 1];
        newNode->K[2 * d] = K[d];
        for (i = 0; i < d; i++)
            newNode->children[i] = children[d + i + 1];
        newNode->m = d;
        m = d;
        if (this == a->root){
            a->root = new BNode(d);
            a->root->m = 1;
            a->root->children[0] = this;
            a->root->children[1] = newNode;
            a->root->K[0] = this->K[d];
            return nullptr;
        } else
            return newNode;
    }
}

```

CHAPTER 5. BINARY SEARCH TREES

Table 5.28: The algorithm that adds a new node into the index part of B+ tree (Java)

```
BNode insertNode(BTree a, int value){
    BNode s, newNode;
    int child, i;
    child = position(value);
    if (!children[child].leaf)
        s = children[child].insertNode(a, value);
    else
        s = children[child].insertLeaf(value);
    if (s == null)
        return null;
    for (i = m; i > child; i--)
        K[i] = K[i - 1];
    K[child] = s.K[2 * d];
    if (m < 2 * d){
        for (i = m + 1; i > child; i--)
            children[i] = children[i - 1];
        children[child] = s;
        m++;
        return null;
    } else {
        newNode = new BNode(d);
        for (i = 0; i < d; i++)
            newNode.K[i] = K[d + i + 1];
        newNode.K[2 * d] = K[d];
        for (i = 0; i < d; i++)
            newNode.children[i] = children[d + i + 1];
        newNode.m = d;
        m = d;
        if (this == a.root){
            a.root = new BNode(d);
            a.root.m = 1;
            a.root.children[0] = this;
            a.root.children[1] = newNode;
            a.root.K[0] = this.K[d];
            return null;
        } else
            return newNode;
    }
}
```

Table 5.29: The algorithm that adds a new value into the data part of B+ tree

<pre> BNode* BNode::insertLeaf(int value){ int i, child; BNode* newNode; child = position(value); for (i = m; i > child; i--) K[i] = K[i - 1]; K[child] = value; if (m < 2 * d){ m++; return nullptr; } else { newNode = new BNode(d); for (i = 0; i < d + 1; i++) newNode->K[i] = K[d + i]; newNode->K[2 * d] = K[d]; newNode->m = d + 1; m = d; return newNode; } } </pre>	<pre> BNode insertLeaf(int value){ int i, child; BNode newNode; child = position(value); for (i = m; i > child; i--) K[i] = K[i - 1]; K[child] = value; if (m < 2 * d){ m++; return null; } else { newNode = new BNode(d); for (i = 0; i < d + 1; i++) newNode.K[i] = K[d + i]; newNode.K[2 * d] = K[d]; newNode.m = d + 1; m = d; return newNode; } } </pre>
--	--

K into the calculated position (Line 7). At this stage there are again two possibilities:

- After inserting the new value, the current leaf node did not exceed its capacity (Lines 8-9). The function returns NULL.
- After inserting the new value, the current leaf node exceed its capacity (Lines 10-16). In this case, we need to create the **newNode** (Line 11), and transfer the last d values of node **b** to this **newNode** (Lines 12-13).

Figure 5.23 shows the case where we insert 22 into the tree shown in Figure 5.20. The leaf node, where 22 will be inserted, contains currently 19 and 20 and has two more extra spaces. As a result, inserting 22 into the leaf node did not cause any restructuring.

Figure 5.24 shows the case where we insert 22 into the tree shown in Figure 5.21:

CHAPTER 5. BINARY SEARCH TREES

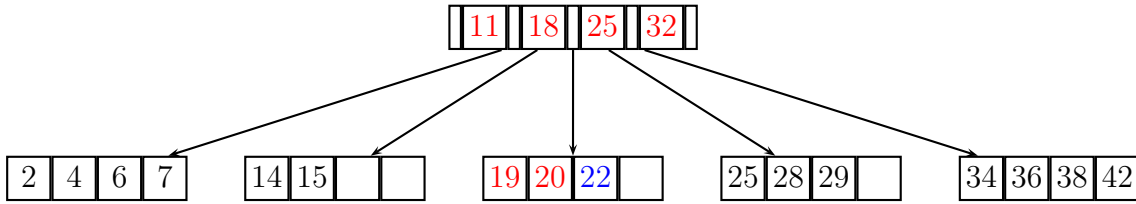


Figure 5.23: Inserting 22 to the tree shown in Figure 5.20

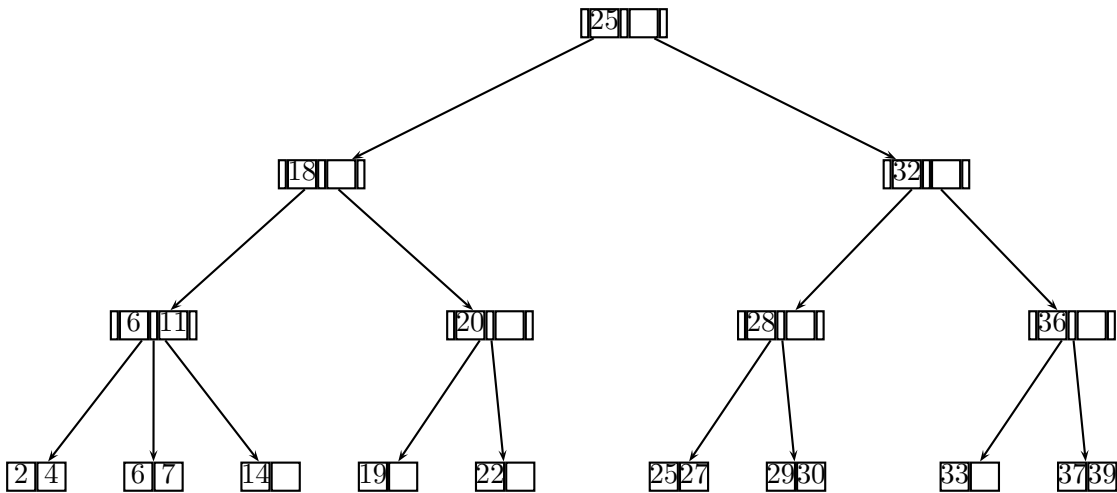


Figure 5.24: Adding 22 to the tree shown in Figure 5.21

- The leaf node, where 22 will be inserted, contains currently 25 and 27 and does not have extra space for any new value. First we insert 22 into this leaf node, then this leaf node is divided into two parts resulting in two leaves where the first one contains 22 and the second one contains 25 and 27. For this case we need to insert 25 into the parent node.
- The first parent node contains 20 and 28 and does not have extra space for a new node. First we insert 25 into this parent node, then this node is divided into two parts resulting in two nodes where the first one contains 20 and the second one contains 28, now for this case we need to insert 25 into the grandparent node.
- The second parent node (grandparent node) contains 18 and 32 and

does not have extra space for a new node. First we insert 25 to this grandparent node, then this node is divided into two nodes where the first one contains 18 and the second one contains 32, and since the divided node is the root node, we create a new root node and insert 25 into this new root node.

5.7 Application: Tree Index

Database management systems is one of the most important subjects of computer science. Basically database systems consist of (i) a database to store permanent information and (ii) queries to ask questions and retrieve answers to these questions. For example, let's think a database system which stores the registration information of a university. This database contains

- The **student** table which contains information about the undergraduate, graduate, Ph. D. students registered to this university
- The **staff** table which contains information about the instructors who give lectures in this university
- The **course** table which contains information about all courses opened in this university until this day
- The **grades** table which contains the grades of all courses of all students registered to this university
- ...

On the other hand, depending on these information, we can retrieve, the name and surname of the student from all registered students in this university, whose student number is 18, from the table **student** with the query of

```
SELECT Name, Surname
FROM Student
WHERE No = 18
```

and the number of all students whose student number are larger than 23 from the table **student** with the query of

CHAPTER 5. BINARY SEARCH TREES

```
SELECT Count(*)  
FROM Student  
WHERE No > 23
```

The most important part in both queries is the condition part defined with the command WHERE. If a condition contains the character ‘=’ then that condition is called an equality condition, if it contains the characters ‘<’ or ‘>’ then that condition is called an interval condition.

To find the result of a query the database systems (i) either we examine the data in the table one by one, or (ii) use an index defined for this table and return the data satisfying the WHERE condition as a result. Database indexes are divided into two as tree indexes depending on the search tree structures and hash indexes depending on the hash data structure. Depending on the type of queries that can be asked for a table, one or more indexes can be defined. For example, if a query is on student number information in **student** table, we can define a primary tree index on student number, if another query is on student surname information, we can define a secondary hash index on student surname. In this chapter we will study the tree index based on binary search tree structure and in the next chapter we will study the hash index based on hash data structure.

5.7.1 Tree Index Structure

Tree index is an index which can be used in queries containing both equality and interval conditions. For example, Table 5.30 shows the node structure that contains the information (no, name, surname) in the **student** table and indexed on the no field. Since no information is stored in the **data** field, all search, insertion, and deletion operations will be done based on this field. Each student will be stored in a node of a binary search tree, and the queries related to the student number are easily answered using the basic property of the binary search tree.

5.7.2 Filling the Binary Search Tree

Let’s assume that the information in the student table are stored in a file named “student.txt” in the hard disk and also assume that all these information can be stored in memory. In order to answer queries on student table,

5.7. APPLICATION: TREE INDEX

Table 5.30: The student class that contains the student information (no, name, surname).

<pre> class Student{ private: string name; string surname; int no; public: Student(int no, string name, string surname); } Student::Student(int no, string name, string surname){ this->no = no; this->name = name; this->surname = surname; } </pre>	<pre> public class Student{ String name; String surname; int no; public Student(int no, String name, String surname){ this.no = no; this.name = name; this.surname = surname; } } </pre>
---	---

we need to upload all student information into computer’s memory. For simplicity, let’s assume that the first line of the student file stores the number of students and starting from the second line each line stores information about a single student. An example student file is given below.

```

4
21 Oguz Kerem
18 Aysel Serhat
42 Aysu Ipek
26 Ergin Dogan

```

According to these information, there are 4 students in the university, the names of these students are Oguz Kerem, Aysel Serhat, Aysu Ipek and Ergin Dogan, and the numbers of these students are 21, 18, 42 and 26 respectively.

Table 5.31 shows the function that reads the student information from such a structured file shown above and places these information into a binary search tree. First we open the file “student.txt” (Line 8) and read the total number of students from the student file (Line 9). For each student (Line 15) we read his/her student number (Line 12), name (Line 13) and surname (Line 14) and we insert the node created with this information (Line 15) into a binary search tree (Line 16). As a last step we close the data file (Line 18).

CHAPTER 5. BINARY SEARCH TREES

Table 5.31: Filling the binary search tree using the information in the student file

```
Tree readFile() throws FileNotFoundException{
    Scanner file ;
    TreeNode d;
    String name;
    String surname;
    int no, i, count = 0;
    Tree tree;
    file = new Scanner(new File("student.txt"));
    count = file.nextInt();
    tree = new Tree();
    for (i = 0; i < count; i++){
        no = file.nextInt();
        name = file.next();
        surname = file.next();
        d = new TreeNode(new Student(no, name, surname));
        tree.insert(d);
    }
    file.close();
    return tree;
}
```

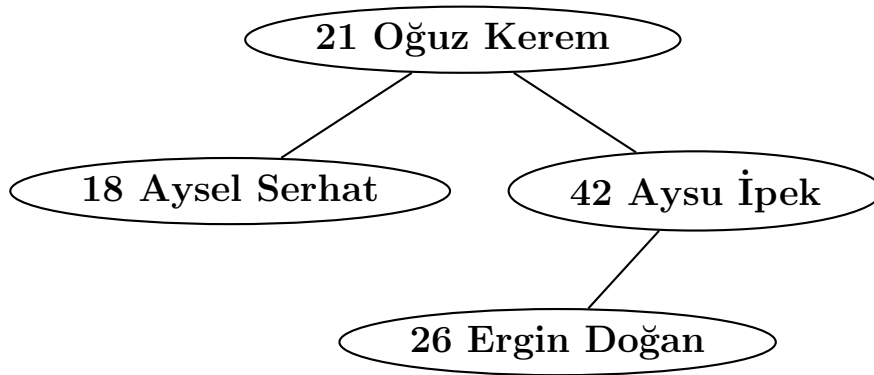


Figure 5.25: The binary search tree filled with the information in the example student file

Figure 5.25 shows the binary search tree filled with the information of 4 students in the example student file. Contrary to the previous examples,

5.7. APPLICATION: TREE INDEX

each node of the tree does not store a single number but a student in the student file. In order to have a balanced tree we can prefer an AVL-tree.

5.7.3 Query

Using the tree index in a database system, we can find the information that satisfy both equality and interval conditions. For example, Table 5.32 shows the function that finds the answer to the equality constrained query

```
SELECT Name, Surname
FROM Student
WHERE No = 18
```

Basically we are searching 18 in the binary search tree. We start the search from the root node (Line 3), if the number of the student in the current node that is compared with is smaller than 18 (Line 5) we continue the search with the right subtree (Line 6), if it is larger than 18 (Line 8) we continue the search with the left subtree (Line 9). When we find the student with number 18 (Line 10), we write the name (Line 11) and surname (Line 12) of the student to the screen and finish the search (Line 13).

Table 5.32: The non-recursive function that finds the name and surname of the student with student number 18

<pre>void query(Tree a){ TreeNode* d; d = a.root; while (d != nullptr){ if (18 > d->data.no) d = d->right; else if (18 < d->data.no) d = d->left; else{ cout << d->data.name; cout << d->data.surname; break; } } }</pre>	<pre>void query(Tree<Student> a){ TreeNode<Student> d; d = a.root; while (d != null){ if (18 > d.data.no) d = d.right; else if (18 < d.data.no) d = d.left; else{ System.out.print(d.data.name); System.out.print(d.data.surname); break; } } }</pre>
---	---

CHAPTER 5. BINARY SEARCH TREES

Table 5.33: The functions that finds the number of students whose numbers are larger than 23

<pre>int query2(TreeNode* d){ int count = 0; if (23 < d->data.no){ count = 1; if (d->left != nullptr) count += query2(d->left); } if (d->right != nullptr) count += query2(d->right); return count; } int query2(Tree a){ if (a.root != nullptr) return query2(a.root); else return 0; }</pre>	<pre>int query2(TreeNode d){ int count = 0; if (23 < d.data.no){ count = 1; if (d.left != null) count += query2(d.left); } if (d.right != null) count += query2(d.right); return count; } int query2(Tree a){ if (a.root != null) return query2(a.root); else return 0; }</pre>
--	--

On the other hand, Table 5.33 show the functions that find the answer to the interval query

```
SELECT Count(*)
FROM Student
WHERE No > 23
```

If there are no students (Line 15), the root node of the tree will be NULL and the function will return 0 (Line 16). If there is at least one student (Line 13), the root node will not be NULL and the recursive function `query2` will be called with this root node (Line 14). If the node that is processed currently is larger than 23 (Line 3), in the nodes on the left and right subtrees of the current node there can be students whose student numbers are larger than 23, therefore for both the nodes on the left subtree (Lines 5, 6) and the nodes on the right subtree (Lines 8, 9) the function will be called again. If the node that is processed currently is smaller than 23, only in the nodes in the right subtree of the current node there can be students whose numbers are larger than 23, therefore only for the nodes in the right subtree the function will be called again.

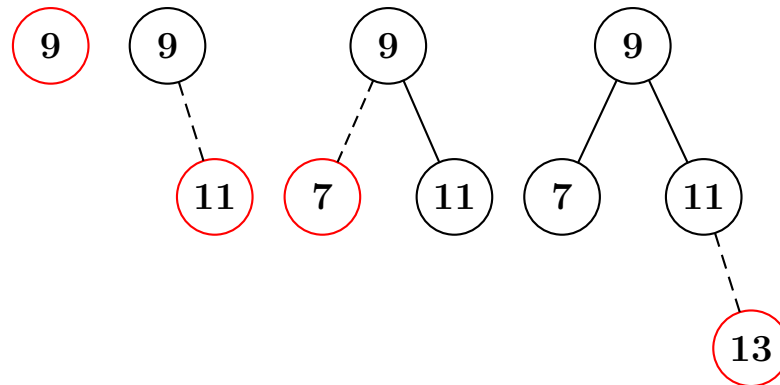
5.8 Notes

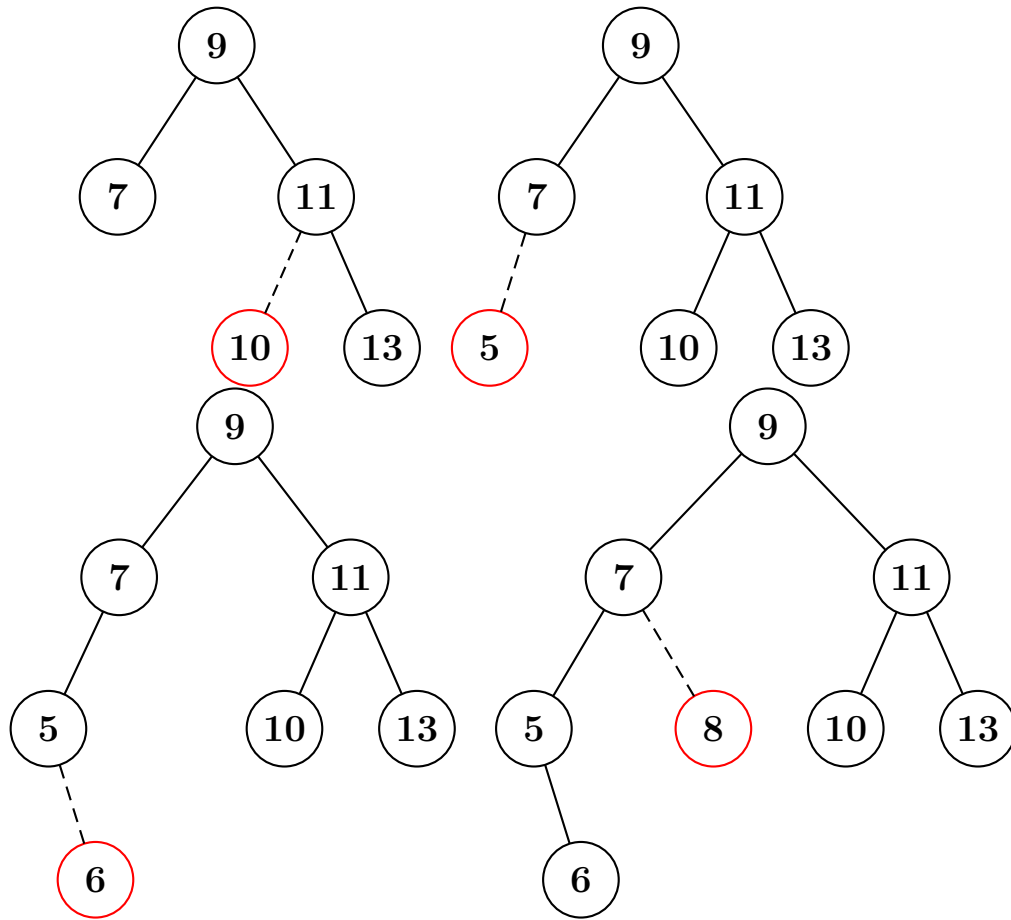
More information about binary search trees and other tree structures can be found in Knuth's two books [10], [12]. AVL trees are proposed by Adelson-Velskii and Landis [1]. The algorithm that removes an element from AVL tree can be found in [12].

5.9 Solved Exercises

1. Show the result of inserting 9, 11, 7, 13, 10, 5, 6, 8 into an initially empty binary search tree.

We see the binary search tree after each operation in the following figure. We add 9 to the root node, 11 to the right of 9, 7 to the left of 9, 13 to the right of 11, 10 to the left of 11, 5 to the left of 7, 6 to the right of 5 and 8 to the right of 7.





2. In which order should we insert the numbers from 1 to $2^k - 1$ such that the resulting tree is perfectly balanced?

The recursive function that finds the order of inserting the numbers from 1 to $2^k - 1$ such that the resulting tree is unbalanced is given below. In order to find a perfect balanced tree, we need to insert the number in the root node, then we need to insert numbers into the left and right children. For this reason, the function determines the number in the root node of each subtree (Line 2). In a perfectly balanced tree, the number of nodes in the left subtree and the number of nodes in the right subtree are the same. Then, the number in the root node is the average of the minimum and maximum numbers. The function then calls itself with its left and right subtrees (Lines 5 and 6). Following the same reasoning we will determine the left subtree of the left subtree

5.9. SOLVED EXERCISES

of etc., then we will backtrack and determine the numbers in the right subtree.

```
void balanced(int left, int right){
    int middle = (left + right) / 2;
    cout << middle;
    if (left != middle){
        balanced(left, middle - 1);
        balanced(middle + 1, right);
    }
}
```

```
void balanced(int left, int right){
    int middle = (left + right) / 2;
    System.out.print(middle);
    if (left != middle){
        balanced(left, middle - 1);
        balanced(middle + 1, right);
    }
}
```

3. Write a function that determines the number of leaves in a binary search tree.

```
int leafCount()
```

The functions that determine the number of leaves in a binary search tree is given below. The function `leafCount` in the class `tree` determines the number of leaves in a tree. To do this, we call the `leafCount2` function in the class `node` with the root node of the tree (Line 13). If there are no nodes in the tree, the function will return 0 (Line 15).

The function `leafCount2` in the class `node` will determine the number of leaf nodes in a subtree given its root node. The number of leaf nodes in a subtree is the sum (Line 9) of the leaf nodes in its left subtree (Lines 5-6) and in its right subtree (Lines 7-8). If a subtree consists of a single leaf node, the function will return 1 (Lines 3-4).

CHAPTER 5. BINARY SEARCH TREES

```
int leafCount2(){
    int total = 0;
    if ( left == nullptr &&
        right == nullptr)
        return 1;
    if ( left != nullptr)
        total += left->leafCount2();
    if ( right != nullptr)
        total += right->leafCount2();
    return total;
}
int leafCount(){
    if (root != null)
        return root->leafCount2();
    else
        return 0;
}
```

```
int leafCount2(){
    int total = 0;
    if (left == null && right == null)
        return 1;
    if (left != null)
        total += left.leafCount2();
    if (right != null)
        total += right.leafCount2();
    return total;
}
int leafCount(){
    if (root != null)
        return root.leafCount2();
    else
        return 0;
}
```

4. Write a function that prints the nodes of a binary search tree in level order. You should list the root, then nodes at depth 1, followed by nodes at depth 2, and so on.

```
void levelOrder ()
```

The function that prints the nodes of a binary search tree in level order is given below. In order to process the tree in level order we need to write a non-recursive function. In this problem we will process the binary tree non-recursively using queue data structure.

Starting from the root node (Line 6-10) each node is added into the queue. At each step, we dequeue one node from the queue (Line 12-13) and if there exists we enqueue the left (Lines 15-18) and right children to the end of the queue (Lines 16-19). The loop continues until there is no node left in the queue (Line 11). For each node, since the children of this node are enqueued to the queue after this node, the process is in level order. First the nodes in level 1, then the nodes in level 2, etc. are processed.

```

void levelOrder(){
    Queue k;
    TreeNode* d;
    Node* e;
    k = Queue();
    d = root;
    if (d != nullptr){
        e = new Node(d);
        k.enqueue(e);
    }
    while (!k.isEmpty()){
        e = k.dequeue();
        d = e->data;
        cout << d->data;
        if (d->left != nullptr){
            e = new Node(d->left);
            k.enqueue(e);
        }
        if (d->right != nullptr){
            e = new Node(d->right);
            k.enqueue(e);
        }
    }
}

```

```

void levelOrder(){
    Queue k;
    TreeNode d;
    Node e;
    k = new Queue();
    d = root;
    if (d != null){
        e = new Node(d);
        k.enqueue(e);
    }
    while (!k.isEmpty()){
        e = k.dequeue();
        d = e.data;
        System.out.print(d.data);
        if (d.left != null){
            e = new Node(d.left);
            k.enqueue(e);
        }
        if (d.right != null){
            e = new Node(d.right);
            k.enqueue(e);
        }
    }
}

```

5. Write a function that solves case 2 without using single rotation functions.

```
AvlNode doubleRotateLeft(AvlNode k3)
```

The left double rotation algorithm that solves case 2 in an AVL tree is given below. 4 links are modified.

- The right child of k_1 is not k_2 anymore but B in other words the left child of k_2 .
- The left child of k_3 is not k_1 anymore but C in other words the right child of k_2 .
- The left child of k_2 is not B anymore but k_1 .
- The right child of k_2 is not C anymore but k_3 .

```

AvlNode* doubleRotateLeft(AvlNode* k3){
    AvlNode *k1, *k2;

```


CHAPTER 5. BINARY SEARCH TREES

```
k1 = k3->left;
k2 = k1->right;
k1->right = k2->left;
k3->left = k2->right;
k2->left = k1;
k2->right = k3;
k1->height = max(height(k1->left), height(k1->right)) + 1;
k3->height = max(height(k3->left), height(k3->right)) + 1;
k2->height = max(k1->height, k3->height) + 1;
return k2;
}
```

```
AvlNode doubleRotateLeft(AvlNode k3){
    AvlNode k1, k2;
    k1 = k3.left ;
    k2 = k1.right;
    k1.right = k2.left ;
    k3.left = k2.right;
    k2.left = k1;
    k2.right = k3;
    k1.height = Math.max(height(k1.left), height(k1.right)) + 1;
    k3.height = Math.max(height(k3.left), height(k3.right)) + 1;
    k2.height = Math.max(k1.height, k3.height) + 1;
    return k2;
}
```

5.10 Exercises

1. Delete node 9 from the binary search tree obtained in the first exercise.
2. Show the inorder, preorder, and postorder traversals of the binary search tree obtained in the first exercise.
3. Which one(s) of the binary search trees obtained in the first exercise is (are) not AVL tree(s)? Which rotations are necessary to convert them into AVL trees?
4. Write a function that prints the numbers in the binary search tree in increasing order.

```
void printSorted ()
```

5.10. EXERCISES

5. Write a function that computes the number of full nodes in a binary search tree. A node is full if it has both left and right children.

int numberOfFullNodes()

6. Write a function that computes the number of singleton nodes in a binary search tree. A node is singleton if it has only one child.

int numberOfSingletonNodes()

7. Write a function that returns a random leaf node from a binary search tree.

Node randomLeaf()

8. Given an integer X , write a function that computes the number of nodes whose key is less than X .

int lessThanX()

9. Write a function that deletes all leaf nodes from a binary search tree.

void deleteLeafNodes()

10. Write a function that computes the sum of all keys in a binary search tree.

int sumOfTree()

11. Write a function that verifies that a tree satisfies binary search tree property.

boolean satisfyTreeProperty()

12. Write a function that swaps left and right children of all nodes in a binary search tree.

void swapChildren()

13. Write a function that determines the height of a tree.

int height()

14. Write a function that prints all nodes in the binary search tree whose key is between k_1 and k_2 .

void printBetween(**int** k1, **int** k2)

CHAPTER 5. BINARY SEARCH TREES

15. Write a function that computes the total height of a tree. The total height of a binary search tree is the sum of heights of all nodes in the tree.

`int totalHeight()`

16. Write a function that solves case 3 without using single rotation functions.

`AvlNode doubleRotateRight(AvlNode k3)`

17. Modify the `TreeNode` data structure such that each node has four children, namely, `left`, `leftOfMiddle`, `rightOfMiddle` and `right`. Then write the function `search` given that the search tree property is satisfied with this kind of tree.

5.11 Problems

1. Write a function that finds the difference between the number of leftist nodes and rightist nodes in a binary search tree. A node is leftist (rightist) if it has only left (right) child.

`int leftistOrRightist ()`

2. Write a function which returns the number of mean nodes. A node is a mean node if its value is the mean of its left and right children's values.

`int number_of_mean_nodes(Treeptr a), int numberOfMeanNodes()`

3. Write a recursive function that

- puts the left child to the right if it has only left child
- puts the right child to the left if it has only right child
- does nothing if it has two children

for all nodes in a binary search tree.

`void changeChildOfSingleton()`

4. Write a recursive function that checks whether the binary search tree contains two same numbers or not.

`boolean containsTwoSameNumbers()`

5.11. PROBLEMS

5. Write a recursive function that returns the number of nodes in a binary search tree whose data is divisible by 3.
6. Write a recursive function that finds the number of non-leaf nodes in the binary search tree.

```
int numberOfNonLeafNodes()
```

7. Write a class method in `TreeNode` class that returns the number of **leftist** nodes in a binary tree. A node is **leftist** if
 - Its left child's data is larger than its right child's data or,
 - It has only left child.

```
int leftist ()
```

8. Write a method that computes the products of all keys in a binary search tree.

```
int productOfTree()
```

9. Write a recursive method , which returns the number of nodes in the binary search tree which have value larger than X . Your method should run in $\mathcal{O}(\log N + K)$ time, where N is total number of nodes and K is the number of nodes which have value larger than X in the tree. Do not use any class or external methods.

```
int higherThanX(int X)
```

10. Write a recursive method in `TreeNode` class that finds the number of duplicate keys in a binary search tree. Assume that if a key is duplicate, it occurs at most twice. Hint: The duplicate of a key is either the maximum number on its left subtree or the minimum number on its right subtree.

```
int numberOfDuplicates()
```

11. Write a recursive method in `TreeNode` class that computes the sum of all keys that are less than X in a binary search tree. You are not allowed to use any tree methods, just attributes, constructors, getters and setters.

```
int sumOfTree(int X)
```

CHAPTER 5. BINARY SEARCH TREES

12. Write a non-recursive method in `Tree` class that returns the depth of the node containing a given data X in a binary search tree. You are not allowed to use any tree methods, just attributes, constructors, getters and setters.

```
int depthOfNode(int X)
```

13. Write a recursive method

```
void pathList(LinkedList l)
```

in the **TreeNode** class, which returns the keys on the path in the linked list l , where the path is defined by the current parent as follows: If the parent is odd, go left; otherwise go right. Assume that the function is called with an empty linked list for the root node.

14. Write a non-recursive method

```
int* pathList()
```

in the **Tree** class, which returns the keys on the path as an array, where the path is defined by the current parent as follows: If the parent is odd, go left; otherwise go right. The array should contain only that many items not more not less.

15. Write the recursive method

```
int averages()
```

in `TreeNode` class which returns the number of nodes in the tree that satisfy the following property: The node's key is the average of its children (left and right children).

16. Write a non-recursive method

```
int sumOfPath(String path)
```

in `Tree` class, which sums the keys on the path, where the path is defined by the parameter `path` as follows: (i) Path consists of 0's and 1's such as 10011. (ii) A 1 represents to go right, a 0 represents to go left. If the path is 1011, you start from root, you go first right, then left, then right, then right. If the path is 001, you start from root, you go first left, then left, then right. You will use `charAt` function in strings.

17. Write the recursive method

void accumulateLeafNodes(**Queue** queue)

in **TreeNode** class which accumulates the contents (integer as data field) of all leaf nodes in queue. For queue, you are only allowed to use enqueue function. You should use array implementation for the queue in this question.

18. Write a non-recursive method

double simulateSearch(**int** N)

in **Tree** class, which first finds the minimum (A) and maximum (B) elements in the tree. The method will then randomly search a number between [A, B] N times and returns the average number of nodes visited in this search. You are not allowed to use any tree methods.

19. Write a recursive method in **TreeNode** class

void accumulate(**int*** a, **int&** index)

that accumulates all contents (integers as data field) in the tree in array a, where the values in the array are will be sorted. Use and modify index to store the integers into correct positions. Your method should run in $\mathcal{O}(N)$ time.

20. Write the recursive method

int sumOfNodesBetween(**int** p, **int** q)

in **TreeNode** class which returns the sum of the keys between p and q in the tree. Your algorithm should run in $\mathcal{O}(\log N + K)$, where K is the number of nodes which have value larger than p and less than q in the tree.

21. Write the recursive method

int [] collectNodes()

in **TreeNode** class, which collects all values in all nodes in the tree in a sorted manner. You are not allowed to use any tree methods.

22. Write a recursive method in **TreeNode** class

void accumulateLeaves(**int*** a, **int&** index)

CHAPTER 5. BINARY SEARCH TREES

that accumulates all leaf contents in the tree in array `a`, where the values in the array are will be sorted. Use and modify `index` to store the integers into correct positions. Your method should run in $\mathcal{O}(N)$ time.

23. T1 and T2 are two binary trees. Write the recursive method

boolean `isIdentical (TreeNode T1, TreeNode T2)`

in `Tree` or `TreeNode` class to determine if T1 is identical to T2.

24. Write the non-recursive method

int `product()`

in `Tree` class that computes the products of all keys in a binary search tree by using stack.

25. Given a binary tree (not necessarily search tree), implement method

bool `isMirror (TreeNode* left, TreeNode* right)`

in `TreeNode` class to check whether an input binary tree is a mirror of itself (symmetric). You may not use any additional data structure or array. Below is an example of a symmetric tree.

5.11. PROBLEMS

With the binary search tree structure we have seen in the previous chapter, we can do insertion, deletion, and search operations on a set of elements in $\mathcal{O}(\log n)$ time. In this chapter we will study hash table data structure. Using the hash table structure, we can do these basic operations more faster, on the average $\mathcal{O}(1)$ time. On the other hand, some of the operations supported in binary search data structure such as finding the minimum and maximum elements and writing the elements in sorted order in linear time are not supported in the hash table data structure.

6.1 Hash Table

6.1.1 Definition

Hash table data structure is a simple but yet powerful data structure and consists of an array of elements. Each element in the array can be a number or a string or can be any data structure. If the table contains N elements, the size of the hash table is said to be N and the elements are addressed between 0 and $N - 1$ similar to a normal array.

The important property of hash data structure is that we use a one-to-one function that maps the elements to its addresses. With the help of this function, the position of the elements can be determined in time of calculation of the function (mostly constant time). This function is called hash function. A good hash function must both scatter the elements in the array well and if

6.1. HASH TABLE

possible must not address two elements into the same position. For example, using an hash function, the elements in Figure 6.1 are addressed, 71 to the position numbered 1, 9 to the position numbered 2, 423 to the position numbered 3, 11 to the position numbered 4 and 76 to the position numbered 6.

0	
1	71
2	9
3	423
4	11
5	
6	76

Figure 6.1: An example hash table which contains 5 elements

Table 6.1 shows the definition of an hash table, which contains integers. Field **table** represents the hash table. Since the hash table is defined with an array, its size is constant and represented with the field **N**.

The basic operations in the hash table data structure are: which hash function is used in determining the addresses of the elements, how is the size of the hash table chosen and if the hash function addresses more than one element into the same position (this is called collision) what will be done.

6.1.2 Hash Functions

If the elements in a hash table are integers, a logical hash function can be $\text{data} \% N$ (Table 6.2). One problem that happens here is that some values may occur more or some values may not occur. For example, if the data

CHAPTER 6. HASHING

Table 6.1: Definition of the hash table

<pre> class Hash{ private: Element** table; bool* deleted; int N; public: Hash(int N); ~Hash(); } Hash::Hash(int N){ table = new Element*[N]; deleted = new bool[N]; this->N = N; } Hash::~~Hash(){ delete [] table; delete [] deleted; } </pre>	<pre> public class Hash{ Element[] table; boolean[] deleted; int N; public Hash(int N){ table = new Element[N]; deleted = new boolean[N]; this.N = N; } } </pre>
---	--

Table 6.2: An example hash function which can be used for integers

<pre> int Hash::hashFunction(int value) { return value % N; } </pre>	<pre> public int hashFunction(int value) { return value % N; } </pre>
--	---

consists of mainly the numbers which are divisible by 10 (the the money in the bank account), choosing N as 10 or a number that is divisible by 10 may not be a good idea. In such cases, the solution is to choose N from prime numbers. For example, for the elements in Figure 6.1, $N = 7$.

If the elements consists of strings which also consists of characters, what we expect from the hash function is first convert the character array into an integer than address this number. Since there are 26 different letters in the English language and since a character array may also contain digits, the number of distinct possible characters in a string will be 36. If there is also case sensitivity, in this case the number of distinct characters that can be observed will be $2 * 26 + 10 = 62$. Accordingly, if we think each character in a string as a digit and starting from the idea that the number of distinct

Table 6.3: An example hash function which can be used for strings

1	int Hash::hashFunction(string value) {	1	public int hashFunction(String value) {
2	int i, pos = 0;	2	int i, pos = 0;
3	for (i = 0; i < value.length(); i++) {	3	for (i = 0; i < value.length(); i++) {
4	pos = 36 * pos + value[i];	4	pos = 36 * pos + value.charAt(i);
5	}	5	}
6	pos = pos % N;	6	pos = pos % N;
7	return pos;	7	return pos;
8	}	8	}

observable characters in a string is 39, we can safely assume that each string corresponds to a number in the base of 39. This way we have successfully construct the mapping from strings to integers.

Table 6.3 shows the hash function which is explained above. The function takes a character array and the size of the hash table as input parameters and returns the position of the new element which will be placed in the hash table. Although this hash function is not the best hash function, it has some of the properties of a good hash function, such as being simple and easily calculable.

In the next section, we will see what we can do when an hash function addresses more than one element to the same position (when there is collision).

6.2 Separate Chaining

One strategy to solve the collision problem is to put all same position addressed elements in a linked list. The linked list structure we have seen in Chapter2 is appropriate for this purpose.

Figure 6.2 shows an example hash table where each element is a linked list. This hash table consists of 10 positions. Using the hash function `sayi % 10`, we add number 10 to the position 0; numbers 1 and 71 to the position 1; numbers 64, 14, and 24 to the position 4; number 75 to the position 5; numbers 6 and 36 to the position 6; numbers 49, 19, and 29 to the position 9. The positions 2, 3, 7, and 8 are empty. The basic assumption in solving the collision problem using linked lists is that the number of same addressed

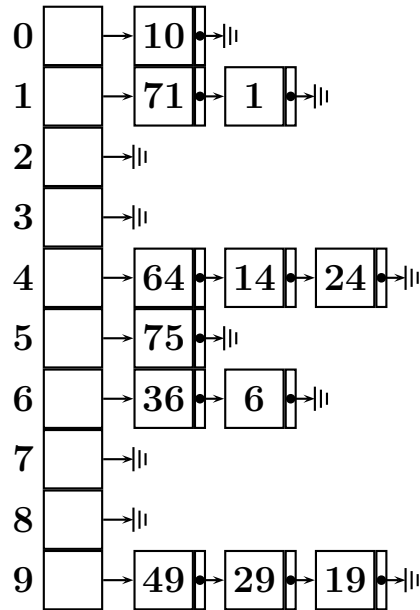


Figure 6.2: An example hash table where each element is a linked list

elements is very small ($\mathcal{O}(1)$ number of elements). In this case, both inserting a specific element and searching a specific element will take $\mathcal{O}(1)$ time.

Table 6.4 shows the definition of the hash table where each element is a linked list. First we allocate memory for the hash table, then we create a linked list for each element of the hash table. In the beginning, all these linked lists are empty. When we place new elements in the hash table, we will insert those element into these linked lists.

6.2.1 Search

Table 6.5 shows the function that searches a specific number in an hash table where each element is a linked list. The function first sends the searched number to the hash function and determines the address of that searched number in the hash table (Line 3). The search operation is completed by searching the same number in the linked list in that calculated address (Line 4). If the number does not exist in the linked list, the function will return NULL.

6.2. SEPARATE CHAINING

Table 6.4: Definition of an hash table where each element is a linked list

<pre> class Hash{ private: LinkedList* table; int N; public: Hash(int N); ~Hash(); } Hash::Hash(int N){ int i; table = new LinkedList[N]; for (i = 0; i < N; i++) table[i] = LinkedList(); this.N = N; } Hash::~~Hash(){ for (i = 0; i < N; i++) delete table[i]; delete [] table; } </pre>	<pre> public class Hash{ LinkedList[] table; int N; public Hash(int N){ int i; table = new LinkedList[N]; for (i = 0; i < N; i++) table[i] = new LinkedList(); this.N = N; } } </pre>
---	--

Table 6.5: Searching a value in a hash table where each element is a linked list

<pre> Node* Hash::search(int value){ int address; address = hashFunction(value); return table[address].search(value); } </pre>	<pre> Node search(int value){ int address; address = hashFunction(value); return table[address].search(value); } </pre>
--	---

6.2.2 Insertion

Table 6.6 shows the function that adds a new item into an hash table where each element is a linked list. The function, similar to the search function, first sends the value of the item to the hash function and determines the address of that new item in the hash table (Line 3). The insertion operation is completed by inserting the new item in the linked list in that calculated address (Line 4).

CHAPTER 6. HASHING

Table 6.6: Inserting a new element into the hash table where each element is a linked list

<pre>void Hash::insert (Node* node){ int address; address = hashFunction(node->data); table[address].insertLast (node); }</pre>	<pre>void insert(Node node){ int address; address = hashFunction(node.data); table[address].insertLast (node); }</pre>
--	--




Table 6.7: Deleting an element from an hash table where each element is a linked list

<pre>1 void Hash::delete(int value){ 2 Node* node; 3 int address; 4 address = hashFunction(value); 5 node = table[address].search(value); 6 if (node != nullptr) 7 table[address].delete(node); 8 }</pre>	<pre>1 void delete(int value){ 2 Node node; 3 int address; 4 address = hashFunction(value); 5 node = table[address].search(value); 6 if (node != null) 7 table[address].delete(node); 8 }</pre>
---	---

6.2.3 Deletion

Table 6.7 shows the function that removes an element from the hash table where each element is a linked list. The function first sends the value of the deleted element to the hash function and determines the address of that deleted element in the hash table (Line 4). We search the element in the linked list in the determined address (Line 5), if it exists in that linked list (Line 6), it is removed from that linked list (Line 7).

Hash Table Operations (Linked List)

-  Search: $\mathcal{O}(1)$
-  Insertion: $\mathcal{O}(1)$
-  Deletion: $\mathcal{O}(1)$

6.3 Open Addressing

Although we can solve collision problem using linked lists, this approach has the following disadvantages (i) we have to use a new data structure and (ii) there is a slow down due to the allocation memory when a new element is added. Yet another alternative to solve collision problem is open addressing. In open addressing, when there is collision, we try alternative positions until we get an empty position. If we express mathematically, when there is collision, we try the addresses $\text{hashFunction}(x) + f(1) \% N$, $\text{hashFunction}(x) + f(2) \% N$, $\text{hashFunction}(x) + f(3) \% N$, ..., one after another. Here function f is called the collision resolution strategy and with different strategies one can find different addresses.

6.3.1 Linear Probing

0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 6.3: Open addressing hash table with linear probing

In linear probing, f is a linear function and is typically $f(i) = i$. So, in this probing we test the positions after the initial position one by one.

CHAPTER 6. HASHING

Figure 6.3 shows an open addressing hash table with linear probing and the final state of the hash table after inserting the numbers 89, 18, 49, 58, 69 in this order. The numbers 89 and 18 do not cause a collision. 49 causes the first collision ($49 \% 10 = 9$ and the position numbered 9 is full). The position after the position numbered 9 is the position numbered 0 and is also empty. For this reason, 49 is added to this position. 58 causes the second collision ($58 \% 10 = 8$ and the position numbered 8 is full). The two positions after the position numbered 8, namely the positions 9 and 0, and both of them are full, and the first empty position after is the position numbered 1 and 58 is added to this position. As a last step, 69 causes the third collision ($69 \% 10 = 9$ and the position numbered 9 is full). The two positions after the position numbered 9 are 0 and 1, and both of them are full, and the first empty position after these positions is the position numbered 2, and 69 is added to this position.

Table 6.8: Searching a number in an hash table (linear probing)

```
1 Element Hash::search(int value){
2     int address;
3     address = hashFunction(value);
4     while (table[address] != nullptr){
5         if (!deleted[address] && table[address]->data == value)
6             break;
7         address = (address + 1) % N;
8     }
9     return table[address];
10 }
```

```
1 Element search(int value){
2     int address;
3     address = hashFunction(value);
4     while (table[address] != null){
5         if (!deleted[address] && table[address].data == value)
6             break;
7         address = (address + 1) % N;
8     }
9     return table[address];
10 }
```

Table 6.8 shows the algorithm that searches a number in an open addressing hash table with linear probing. The function first sends the searched

6.3. OPEN ADDRESSING

number to the hash function and determines the possible address of that number in the hash table (Line 3). If there exists such a number and if that number is not deleted before (Line 5), the element in that address is returned. If the number searched is not in the first address or is deleted before, we search possible addresses by incrementing the address one by one (Line 7).

Table 6.9: Inserting an element into the open addressing hash table with linear probing

```
1 void Hash::insert(Element* element){
2     int address;
3     address = hashFunction(element->data);
4     while (table[address] != nullptr && !deleted[address])
5         address = (address + 1) % N;
6     if (table[address] != nullptr)
7         deleted[address] = false;
8     table[address] = element;
9 }
```

```
1 void insert(Element element){
2     int address;
3     address = hashFunction(element.data);
4     while (table[address] != null && !deleted[address])
5         address = (address + 1) % N;
6     if (table[address] != null)
7         deleted[address] = false;
8     table[address] = element;
9 }
```

Table 6.9 shows the function that inserts a new element into an open addressing hash table with linear probing. The function first sends the value of the new element to the hash function and determines the possible address of that element in the hash table (Line 3). Two cases are possible:

- The determined address is not used before (The value in that address of the array **table** is NULL). In this case, we can insert a new element in that address (Line 8).
- There was an element in that address before but it was deleted (The value in that address of the array **deleted** is 1). In this case, we can

CHAPTER 6. HASHING

insert a new element into the place of the deleted element (Line 8) and the value in that address of the array `deleted` is now 0 (Line 7).

If the address is full and was not deleted before, we need to determine the second possible address according to the collision resolution strategy. If the collision resolution strategy is linear probing, the next address is the second address. If the second address is also full and also was not deleted before, we continue incrementing the address one by one until we get an empty address (Line 5).

Table 6.10: Deleting an element from an open addressing hash table with linear probing

```
1 void Hash::delete(int value){
2     int address;
3     address = hashFunction(value);
4     while (table[address] != nullptr){
5         if (!deleted[address] && table[address]->data == value)
6             break;
7         address = (address + 1) % N;
8     }
9     deleted[address] = true;
10 }
```

```
1 void delete(int value){
2     int address;
3     address = hashFunction(value);
4     while (table[address] != null){
5         if (!deleted[address] && table[address].data == value)
6             break;
7         address = (address + 1) % N;
8     }
9     deleted[address] = true;
10 }
```

Table 6.10 shows the algorithm that removes a value from an open addressing hash table with linear probing. The function first sends the to be deleted value to the hash function and determines the first possible address of that value in the hash table (Line 3). If the value in that determined address is not equal to the deleted value or if the element in that address is deleted, we need to determine the second possible address according to the

collision resolution strategy. If the collision resolution strategy is linear probing, this is the next address. If the value in that second address is not equal to the deleted value or if the element in that second address is deleted, we continue incrementing the address one by one until we get an empty address (Line 7). If the address of to be deleted element is found, we set the value in that address of `deleted` array to 1 to show that the element in that address is deleted (Line 9).

As long as the hash table is large enough, we can find an empty position with linear probing. But linear probing can spend much more time to find this empty position. Worse than that, if the table is empty, blocks of full positions (primary clusters) starts forming. When the hash function maps a value to a position in such a cluster, more attempts are required to resolve that collision, which takes time.

6.3.2 Quadratic Probing

Quadratic probing is an alternative collision strategy that was proposed to bring solution to the primary clustering problem of the linear probing. In quadratic probing, the collision function is a quadratic function, and is typically $f(i) = i^2$.

Figure 6.4 shows an open addressing hash table with quadratic probing and the final state of the hash table after inserting the numbers 89, 18, 49, 58, 69 in this order. The numbers 89 and 18 do not cause a collision. 49 causes the first collision ($49 \% 10 = 9$ and the position numbered 9 is full). The position after the position numbered 9 is the position numbered $(9 + 1^2) \% 10 = 0$ and is empty. For this reason, 49 is added to this position. 58 causes the second collision ($58 \% 10 = 8$ and the position numbered 8 is full). The position after the position numbered 8, namely the position numbered $(8 + 1^2) \% 10 = 9$ is full, and the next empty position is the position numbered $(8 + 2^2) \% 10 = 2$ and 58 is added to this position. As a last step, 69 causes the third collision ($69 \% 10 = 9$ and the position numbered 9 is full). The position after the position numbered 9, namely the position numbered $(9 + 1^2) \% 10 = 0$ is full, and the first empty position after this position is the position numbered $(9 + 2^2) \% 10 = 3$, and 69 is added to this position.

In quadratic probing, if the size of the hash table is prime and the the table is half full, it can be proved that we can always find an empty position for the new element. Although it is a very low probability event, if the table is one more full than half, it is sometime impossible to insert a new element.

0			49	49	49
1					
2				58	58
3					69
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figure 6.4: An open addressing hash table with quadratic probing

Quadratic probing brings solution to the primary clustering problem. On the other hand, the elements that are addressed to the same position will again try those same positions as alternatives. This problem is known as secondary clustering in the literature.

6.3.3 Double Hashing

Double hashing is proposed to bring solution to the secondary clustering problem of quadratic probing. In double hashing, the elements, those addressed to the same positions, will use a second hash function to determine the alternative positions. For that reason, this collision resolution strategy is called double hashing. Since a second hash function is used, it is a low possibility that, the elements that are addressed to the same positions with the first hash function, will be addressed to the same position with the second hash function.

A typical collision function used in double hashing is $f(i) = i \times hashFunction_2(x)$.

6.3. OPEN ADDRESSING

As secondary hash function

$$\text{hashFunction}_2(x) = S - (x \% S)$$




S is a prime number and also smaller than the size of the hash table.

0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 6.5: An open addressing hash table with double hashing

Figure 6.5 shows an open addressing hash table with double hashing and the final state of the hash table after inserting the numbers 89, 18, 49, 58, 69 in this order. As a second hash function we use $7 - (x \% 7)$. The numbers 89 and 18 do not cause a collision. 49 causes the first collision ($49 \% 10 = 9$ and the position numbered 9 is full). The position after the position numbered 9 is the position numbered $(9 + (7 - (49 \% 7))) \% 10 = 6$ and is empty. For this reason, 49 is added to this position. 58 causes the second collision ($58 \% 10 = 8$ and the position numbered 8 is full). The position after the position numbered 8, namely the position numbered $(8 + (7 - (58 \% 7))) \% 10 = 3$ is empty, and 58 is added to this position. As a last step, 69 causes the third collision ($69 \% 10 = 9$ and the position numbered 9 is full). The position after the position numbered 9, namely the position numbered $(9 + (7 - (69 \% 7))) \% 10 = 0$ is empty, and 69 is added to this position.

Hash Table Operations (Array)

-  Search: $\mathcal{O}(1)$
-  Insertion: $\mathcal{O}(1)$
-  Deletion: $\mathcal{O}(1)$

6.4 Rehashing

When the hash table starts to get full, the insertion operations will take more time and with quadratic probing it will be the case that, we may not be able to insert more elements. In this case, the solution is to create a new hash table of double size of the old hash table and move the elements in the old hash table to this new hash table by calculating the new addresses.

Table 6.11: Rehashing an hash table

<pre> 1 void Hash::rehash(){ 2 int i; 3 Element*[] table; 4 bool[] deleted; 5 table = new Element*[N]; 6 deleted = new bool[N]; 7 for (i = 0; i < N; i++){ 8 table[i] = this->table[i]; 9 deleted[i] = this->deleted[i]; 10 } 11 this->table = new Element*[2 * N]; 12 this->deleted = new bool[2 * N]; 13 N = 2 * N; 14 for (i = 0; i < N / 2; i++) 15 if (table[i] != nullptr && !deleted[i]) 16 insert (table[i]); 17 }</pre>	<pre> void rehash(){ int i; Element[] table; boolean[] deleted; table = new Element[N]; deleted = new boolean[N]; for (i = 0; i < N; i++){ table[i] = this.table[i]; deleted[i] = this.deleted[i]; } this.table = new Element[2 * N]; this.deleted = new boolean[2 * N]; N = 2 * N; for (i = 0; i < N / 2; i++) if (table[i] != null && !deleted[i]) insert (table[i]); }</pre>
---	---

Table 6.11 shows the function that enlarges the hash table when it gets full (enlarging the capacity of the hash table by 2 times). First the information

6.4. REHASHING

that represents which elements are deleted or not (array **deleted**) is transferred to a temporary array (Lines 5 - 10). Then the original hash table and array **deleted** are enlarged to two times of their original size (Line 11-12). The new table size is set (Line 13), the elements those transferred to the temporary array are inserted back to this new hash table (Lines 14-16).

0	
1	71
2	
3	423
4	
5	
6	76
7	
8	
9	9
10	
11	11
12	
13	

Figure 6.6: Rehashing of the original hash table in Figure 6.1

Figure 6.6 shows rehashing of the hash table given in Figure 6.1. While the places of 71, 423, and 76 do not change, the elements 9 and 11 placed to the positions numbered 9 and 11 respectively.

6.5 Application: Dart

In Chapter 4 we see breadth first search algorithm to solve the dart problem. As we remember, in breadth first search algorithm, first we examine the states that can be accessed with one shot from the initial state (0), then the states that can be accessed with two shots, then the states that can be accessed with three shots, etc. and if one of these states is a final state, we will finish the search.

For example, in the Dart game given in Figure 4.7, with one shot we can move from the beginning state (0) to the states 11, 21, 27, 33, and 36. On the other hand, with two shots, we can move to the states 22 (11 + 11), 32 (11 + 21), 38 (11 + 27), 42 (21 + 21), 44 (11 + 33), 47 (11 + 36), 48 (21 + 27), 54 (27 + 27 or 21 + 33), 57 (21 + 36), 60 (27 + 33), 63 (27 + 36), 66 (33 + 33), 69 (33 + 36), and 72 (36 + 36). Figure 4.8 shows the states that can be accessed with two shots in a state graph.

Note that, we can visit a state in more than one way. For example, to visit the state 54, we can shot 27 two times or shot 21 and 33 once. Similarly, staying the values of the shots the same, each possible permutation of those shots will lead us to the same state. For example, we can shot three different areas valued 11, 21, 33 in $3! = 6$ different ways and get the same sum each time. Therefore, while solving the problem using breadth first search, if we can visit a state in N different ways, that state will be enqueued and dequeued N times. So, although there are 101 different possible states (0, 1, ..., 100), we will enqueue much more states than that. Instead of this, when we try to insert a state to the queue, we will check if we have already visited that state (enqueued state), and if we haven't visited that state, then we will insert that state to the queue.

The fastest way of checking, if we have visited that state or not, is to hold the states we have already visited in an hash table. Since insertion and deletion operations in an hash table takes constant time, (i) we can search the hash table in constant time to check the newness of each state we encounter, (ii) if the state we visit is a new state, we can insert that state into the hash table in constant time.

The solution to the Dart game problem using breadth first search which also holds the visited states in an hash table is given in Table 6.13. As the first step, the beginning state, the empty board state (Line 8), is added as

Table 6.12: Solving the Dart game problem using breadth first search (2) (C++)

```

1 string dartGame(int* board){
2     int i, t;
3     string a;
4     Node* e;
5     Queue k;
6     Element* o;
7     Hash kt;
8     e = new Node(State(0, ""));
9     k = Queue();
10    k.enqueue(e);
11    o = new Element(0);
12    kt = Hash(1000);
13    kt.insert(o);
14    while (!k.isEmpty()){
15        e = k.dequeue();
16        if (e->data.total == 100)
17            return e->data.darts;
18        for (i = 0; i < 5; i++){
19            t = e->data.total + board[i];
20            if (t <= 100)
21                if (kt.search(t) != nullptr){
22                    a = e->data.darts;
23                    a = a + "," + board[i];
24                    e = new Node(State(t, a));
25                    k.enqueue(e);
26                    o = new Element(t);
27                    kt.insert(o);
28                }
29        }
30    }
31    return "";
32 }

```

the first element to the queue (Line 10). Since this state is already visited, it is also inserted into the hash table (Line 13). At each stage, one state is removed from the queue (Line 15), if the sum of the shots of this state is 100 (Line 16), the function returns the shots list of this state (Line 17). For each state retrieved from the queue, the states that can be reached from that state are analyzed (Line 18). If the reachable state

CHAPTER 6. HASHING

Table 6.13: Solving the Dart game problem using breadth first search (2)
(Java)

```
1 static String dartGame(int[] board){
2     int i, t;
3     String a;
4     Node e;
5     Queue k;
6     Element o;
7     Hash kt;
8     e = new Node(new State(0, ""));
9     k = new Queue();
10    k.enqueue(e);
11    o = new Element(0);
12    kt = new Hash(1000);
13    kt.insert(o);
14    while (!k.isEmpty()){
15        e = k.dequeue();
16        if (e.data.total == 100)
17            return e.data.darts;
18        for (i = 0; i < 5; i++){
19            t = e.data.total + board[i];
20            if (t <=100)
21                if (kt.search(t) != null){
22                    a = e.data.darts;
23                    a = a + "-" + board[i];
24                    e = new Node(new State(t, a));
25                    k.enqueue(e);
26                    o = new Element(t);
27                    kt.insert(o);
28                }
29        }
30    }
31    return null;
32 }
```

- is a defined state, that is, the sum of its shots is less than or equal to 100 (Line 20)
- and we haven't visited the state yet, in other words if this state does not exist in the hash table (Line 21)

new state is added to the queue (Lines 24-25) and the hash table (Lines 26-

27). The shots list of the new state is obtained by adding the number (as a string) to the shots list of the previous state (Line 22-23).

6.6 Application: Hash Index

6.6.1 Hash Index Structure

Hash index is an index that can only be used in queries that contain equality conditions. Remember that, Table 5.30 shows the node structure that contains the information (no, name, surname) in the **student** table and indexed on the no field. Since student number info is stored in the field **no**, all insertion, deletion and search operations will be done based on this field. Each student will be stored in an element of an hash table, the queries about the student number will be answered faster using the hash data structure.

6.6.2 Filling the Hash Table

Let's assume that similar to the Section 5.7.2, the information in the student table are stored in a file named "student.txt" in the hard disk. Assume also that the first line of the student file stores the number of students and starting from the second line each line stores information about a single student.

Table 6.14 shows the function that reads the student information from such a structured file shown above and places these information into an hash table. First we open the file "student.txt" (Line 8) and read the total number of students from the student file (Line 9). For each student (Line 11) we read his/her student number (Line 12), name (Line 13) and surname (Line 14) and we insert the element created with this information (Line 15) into an hash table (Line 16). As a last step we close the data file (Line 18).

6.6.3 Query

Using the hash index in a database system, we can only find the information that satisfy equality conditions. For example, Table 6.15 shows the function that finds the answer to the equality constrained query

```
SELECT Name, Surname
FROM Student
```

CHAPTER 6. HASHING

Table 6.14: Filling the hash table using the information in the student file

```
1 Hash readFile() throws FileNotFoundException{
2     Scanner file ;
3     Element e;
4     String name;
5     String surname;
6     int no, i, count;
7     Hash k;
8     file = new Scanner(new File("student.txt"));
9     count = file.nextInt();
10    k = new Hash<Student>(new StudentHashFunction(1000));
11    for (i = 0; i < count; i++){
12        no = file.nextInt();
13        name = file.next();
14        surname = file.next();
15        e = new Element<Student>(new Student(no, name, surname));
16        k.insert(e);
17    }
18    return k;
19 }
```

WHERE No = 18

Basically we are searching 18 in the hash table (Line 3). When we find the student with number 18, we write the name (Line 4) and surname (Line 5) of the student to the screen.

Table 6.15: The function that finds the name and surname of the student with student number 18 using hash table

```
1 void query(Hash k){
2     Element o;
3     o = k.search(new Student(18, "", ""));
4     System.out.print(o.data.name);
5     System.out.print(o.data.surname);
6 }
```

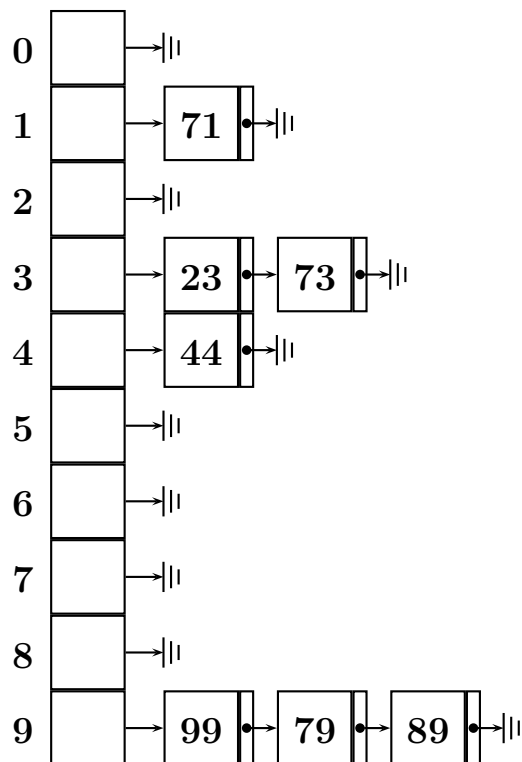
6.7 Notes

One of the first papers about hashing and hash functions is [13]. More detailed information and a more through analysis on hashing using linear probing can be found in Knuth's book [12]. Time complexity analysis of double hashing can be found in [6].

6.8 Solved Exercises

1. Given input $\{71, 23, 73, 99, 44, 79, 89\}$ and a hash function $h(x) = x \% 10$ with an hash table of size 10, show the resulting separate chaining hash table.

The separate chaining hash table of size 10 after inserting numbers 71, 23, 73, 99, 44, 79, 89 is given below.



CHAPTER 6. HASHING

2. Write an hash function that maps a binary search tree into an hash value. Assume that the hash value of a tree can be obtained first by summing up the key values of the nodes and then hashing the sum.

```
int hashFunction(Tree a)
```

The function that maps a binary search tree into an hash value is given below. Since the hash value of a tree can be obtained first by summing up the key values of the nodes and then hashing the sum, all nodes of the binary search tree are visited non-recursively using queue data structure. As long as the queue is not empty (Line 12), the elements are dequeued from the queue one by one (Line 13). Since each element of the queue corresponds to a node of the binary tree (Line 14), the hash values of the elements are accumulated in the variable `sum` (Line 15). As a last step, the left and right subtrees of the node, if they exist (Lines 16, 20), are encapsulated in an element (Lines 17, 21), and added to the queue (Lines 18, 22). In this way, each node of the tree is inserted into the queue, removed from the queue, mapped with the hash function, and summed once. As a last step, this sum is mapped again with the hash function to an address (Line 25).

```

1  int hashFunction(Tree a){
2      TreeNode d;
3      Node e;
4      Queue k;
5      int total = 0;
6      k = new Queue();
7      d = a.root;
8      if (d != NULL){
9          e = new Node(d);
10         k.enqueue(e);
11     }
12     while (!k.isEmpty()){
13         e = k.dequeue();
14         d = e.treenode;
15         total += d.data % N;
16         if (d.left != null){
17             e = new Node(d.left);
18             k.enqueue(e);
19         }
20         if (d.right != null){
21             e = new Node(d.right);
22             k.enqueue(e);
23         }
24     }
25     return total % N;
26 }

```

3. Another strategy for solving the collusion problem is to use binary search trees rather than linked lists in the separate chaining. Define an hash structure where each element of the hash table is a binary search tree instead of a linked list.

The hash table structure where each element is a binary search tree is given below. First we allocate memory for the hash table, then we create a new binary search tree for each element of the hash table. These binary search trees are empty in the beginning. While inserting element into the hash table, we will insert into these binary search trees.

CHAPTER 6. HASHING

```
1 public class Hash{
2     Tree[] table;
3     int N;
4     public Hash(int N){
5         int i;
6         table = new Tree[N];
7         for (i = 0; i < N; i++)
8             table[i] = new Tree();
9         this.N = N;
10    }
11 }
```

4. In open addressing, write a function that inserts a new key to the hash table with quadratic probing as the collision resolution strategy.

void insert (Element element)

The function that inserts a new key into the hash table with quadratic probing as the collision resolution strategy is given below. The function first sends the value of the new element to the hash function and determines the possible address of that element in the hash table (Line 4). Two cases are possible:

- The determined address is not used before (The value in that address of the array `table` is `NULL`). In this case, we can insert a new element in that address (Line 11).
- There was an element in that address before but it was deleted (The value in that address of the array `deleted` is 1). In this case, we can insert a new element into the place of the deleted element (Line 11) and set the value in that address of the array `deleted` to 0 (Line 10).

If the address is full and was not deleted before, we need to determine the second possible address by first incrementing 1, then 4, 9, 16, ..., i^2 (Line 6).

```
1 void insert(Element element){
2     int i, address;
3     i = 1;
4     address = hashFunction(element.data);
5     while (table[address] != null && !(deleted[address])){
6         address = (address + 1) % N;
```

```

7      i++;
8  }
9      if (table[address] != null)
10         deleted[address] = false;
11         table[address] = element;
12 }

```

6.9 Exercises

1. Given input {71, 23, 73, 99, 44, 79, 89} and a hash function $h(x) = x \% 10$ with an hash table of size 10, show the resulting open addressing hash table using linear probing.
2. Write an hash function that maps a linked list into an hash value. Assume that the hash value of a linked list can be obtained first by summing up the key values of the nodes in the linked list and then hashing the sum.

```
int hashFunction(LinkedList l)
```

3. Write an hash function that maps a queue into an hash value. Assume that the hash value of a queue can be obtained first by summing up the key values of the elements in the queue and then hashing the sum.

```
int hashFunction(Queue k)
```

4. Write an hash function that maps a stack into an hash value. Assume that the hash value of a stack can be obtained first by summing up the key values of the elements in the stack and then hashing the sum.

```
int hashFunction(Stack c)
```

5. Given the binary search tree approach as a separate chaining strategy (Solved Exercise 3), write a function that finds a specific value in that structure.

```
Node search(int value)
```

6. Given the binary search tree approach as a separate chaining strategy (Solved Exercise 3), write a function that inserts a new value in that structure.

CHAPTER 6. HASHING

void insert (Node newNode)

7. Write a function that finds the maximum element in an hash table where collision solving strategy is separate chaining.

Node maximum()

8. In open addressing, write a function that searches a key value in the hash table with quadratic probing as the collision resolution strategy.

Element search(**int** value)

9. Write a function that computes the load factor of an hash table implemented with a fixed array. The load factor of a hash table is the ratio of the number of elements in the hash table to the table size.

double loadFactor()

10. Write a function that computes the load factor of an hash table implemented with an array of linked lists (Separate chaining). The load factor of a hash table is the ratio of the number of elements in the hash table to the table size.

double loadFactor()

11. Write a function which returns the number of items in the hash table whose values are between X and Y. Your method should run in $\mathcal{O}(N)$ time.

int between(**int** X, **int** Y)

12. Write a function which undeletes the recently deleted value from the hash table. Assume that linear probing is used as the collision strategy.

void undelete(**int** value)

13. Write a function that finds the minimum element in a hash table with linear probing.

Element minimum()

6.10 Problems

1. A sequence of $n > 0$ integers is called a jolly jumper if the absolute values of the differences between successive elements take on all possible values 1 through $n - 1$. For instance, 1 4 2 3 is a jolly jumper, because the absolute differences are 3, 2, and 1, respectively. Write a function to determine whether a sequence of numbers is a jolly jumper.

boolean jollyJumper(**int**[] sequence)

2. Write function that finds the number of empty slots in an hash table (For both array and linked list implementations).

int numberOfEmptySlots()

3. Write a method that deletes all elements having value X . Assume also that X can exist more than once in the hash table. Write the function for both array and linked list implementations. For array implementation assume that linear probing is used as the collision strategy. Do not use any class or external methods except hashFunction.

void deleteAll (**int** X)

4. Write an hash function that maps the key values in an hash table into an hash value. Assume that the hash value of an hash table can be obtained first by summing up the key values of the elements in the hash table and then hashing the sum. Write the function for array implementation. Assume also that linear probing is used as the collision strategy. Do not use any class or external methods except hashFunction.

int hashFunctionItSelf()

5. Write a static method that takes an array of integers as a parameter and checks if the array contains any duplicate elements. Your method should run in $O(N)$ time, where N is the size of the array. You are allowed to use any methods and external data structures we learned in the class.

boolean anyDuplicate(**int**[] array)

CHAPTER 6. HASHING

6. Write a method that simplifies a hash table by creating a new hash table containing elements from the original hash table, where
 - For single occurrence of a value, copy that value to the new table
 - For multiple occurrences of that value, copy that value only once to the new table

Write the function both array and linked list implementations. You are allowed to use linked list and hashing methods.

`Hash simplify()`

7. Write the method

`int numberOfClusters()`

that finds the number of clusters in hash table. A cluster is a contiguous group of non-null elements in the array.

8. Write the method

`boolean perfectMap()`

that returns true if the hash table contains one node at maximum per linked list in separate chaining, otherwise it returns false.

9. Write a static method in Hash class

`int numberOfExtras(int[] array)`

that takes an array of integers as a parameter and counts the number of extra elements in the array. **Your method should run in $\mathcal{O}(N)$ time, where N is the size of the array.** Use hashing.

1 4 2 5 2 4 3 4 \rightarrow 3 extras (two 4, one 2)

2 1 2 1 2 3 1 2 1 2 \rightarrow 7 extras (four 2, three 1)

1 1 1 1 1 1 \rightarrow 5 extras (five 1)

10. Write a static method

`int[] sortByHashing(int[] array)`

that takes an array of integers as a parameter (which contains distinct numbers less than $2N$, where N is the number of elements in the array)

and returns the sorted version of that array. Your method should run in $\mathcal{O}(N)$ time. Do not use any external data structures or arrays except the resulting array and hash table. Hint: Find the maximum number in the array, the sorted array should contain only numbers less than that.

11. Write the static method

```
int [] intersection (int [] list1 , int [] list2 )
```

to find the intersection of the elements in two arrays and return a new array. Your method should run in $\mathcal{O}(N)$ time, where N is the number of elements in the arrays. Do not use any external data structures or arrays except the resulting array and hash table. The intersection array should contain only that many items not more not less. Hint: How can you search an element from the first list in the second list in $\mathcal{O}(1)$?

12. Write the static method in Hash class

```
boolean sumOfTwoK(int[] array, int K)
```

that takes an array of integers as a parameter and returns true if the sum of two elements in the array is K . Your method should run in $\mathcal{O}(N)$ time. Do not use any external data structures or arrays except the external hash table.

13. Write the method in Hash class linked list implementation

```
boolean isValid()
```

that checks if the hash table is valid or not. An hash table is invalid if it contains the same number twice. Your method should run in $\mathcal{O}(N)$ time. Do not use external data structures or hash tables.

14. Write the static method in Hash class

```
boolean sumOfFourK(int[] array, int K)
```

that takes an array of integers as a parameter and returns true if the sum of four elements in the array is K . Your method should run in $\mathcal{O}(N^2)$ time. You are only allowed to use an external array and an external hash table.

15. Write a static method

CHAPTER 6. HASHING

```
int [] union(int [] list1 , int [] list2 )
```

to find the union of the elements in two arrays and return a new array. The union array should contain only that many items not more not less. Your method should run in $O(N)$ time, where N is the number of elements in the arrays. Do not use any external data structures or arrays except the resulting array and an external hash table.

6.10. PROBLEMS

Heap

The heap data structure is a binary tree structure consisting of N elements. It shows the basic properties of the binary tree data structure. The heap has a root node and each node of it has at most two child nodes (left and right). The root node of the tree is shown at the top, and its branches grow not to up but to down manner.

Until this point, the heap data structure seems to be similar to the binary tree structure, but from this point on it will differ from the binary search tree. The main differences between two structures are:

- The binary search tree is a data structure designed to search efficiently, whereas the heap structure is designed to find a maximum priority element from a group of elements and delete that element from that group efficiently.
- In the binary search tree each node is larger than the nodes in its left subtree and smaller than the nodes in its right subtree, whereas in the heap, each node is larger than all of its descendants, that is all of the nodes both in its left and right subtrees. There is no large-small relationship between its left and right subtrees.
- In the binary search tree, each node can have left or right children, or no children. On the other hand, in the heap data structure the elements are placed level by level. One can not have an element in a lower level before the upper level is full. Therefore, the nodes in all levels except the lowest level has two children.

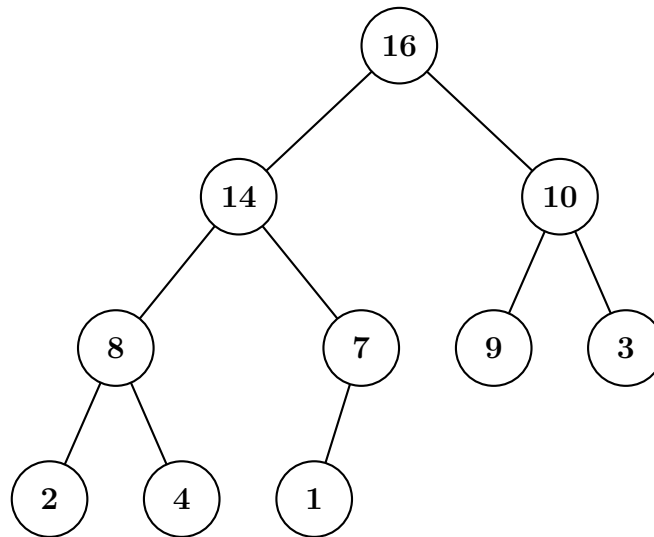


Figure 7.1: An example max-heap

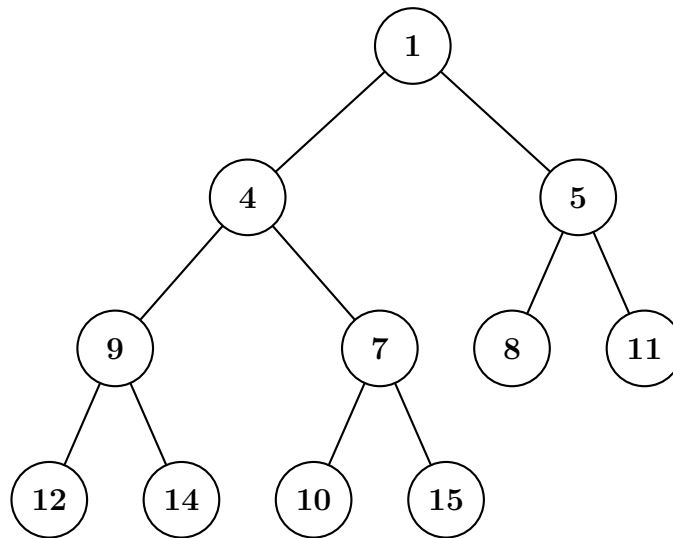


Figure 7.2: An example min-heap

In a heap, if the maximum element is in the root node and all nodes are smaller than its descendants, then this heap is called max-heap, if the minimum element is in the root node and all nodes are larger than its descendants,

then this heap is called min-heap.

Figure 7.1 shows an example max-heap containing 10 elements. The root node of the tree is 16 and larger than all other elements. Also all other nodes are larger than their descendants. For example, 14 is larger than its descendants 8, 2, 4, 7, and 1, 10 is larger than its descendants 9 and 3. The first three levels of this tree is full, and there are three elements in the fourth level.

Figure 7.2 shows an example min-heap containing 11 elements. The root node of the tree is 1 and smaller than all other elements. Also other nodes are smaller than their descendants. For example, 4 is smaller than its descendants 9, 12, 14, 7, 10, 15, 5 is smaller than its descendants 8 and 11.

7.1 Definition

In max-heap the value of each node is larger than the values of its left and right child nodes, in min-heap the value of each node is smaller than the values of its left and right child nodes.

Table 7.1: Definition of elements stored in the heap

<pre> 1 class HeapNode{ 2 private: 3 int data; 4 int name; 5 public: 6 HeapNode(int data, int name) 7 } 8 HeapNode::HeapNode(int data, int name){ 9 this->data = data; 10 this->name = name; 11 }</pre>	<pre> public class HeapNode{ int data; int name; public HeapNode(int data, int name){ this.data = data; this.name = name; } }</pre>
---	---

Table 7.1 shows the definition of each element in the heap structure. **data** field represents the value of each element. **name** field represents the name of the element and used when we want to retrieve the element not from its value but from its name. In many heap applications, the fields **data** and **name** are the same, but as we will see in the shortest path application in Chapter 9, in some applications we need both the names and values of the elements.

7.1. DEFINITION

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Figure 7.3: Array representation of an example heap

Table 7.2: Definition of heap

<pre> 1 class Heap{ 2 private: 3 HeapNode *array; 4 int count; 5 public: 6 Heap(int N); 7 ~Heap(); 8 bool isEmpty(); 9 } 10 Heap::Heap(int N){ 11 array = new HeapNode[N]; 12 count = 0; 13 } 14 Heap::~Heap(){ 15 delete [] array; 16 } 17 bool Heap::isEmpty(){ 18 return count == 0; 19 } </pre>	<pre> public class Heap{ HeapNode array[]; int count; public Heap(int N){ array = new HeapNode[N]; count = 0; } boolean isEmpty(){ return count == 0; } } </pre>
---	---

One important point of this structure is that, each level except the last level are full. This is the most important property of the heap structure and due to this property, the heap can be represented with an array. What we need to do is to place all elements in all levels into an array starting from the root node. The root node will be placed to position 0, left child to position 1, right child to position 2, etc. The array representation of the example max-heap given above is shown in Figure 7.3. 16 in the root node is placed to the zeroth position of the array, the left child of the root node 14 is placed to the first position of the array, the right child of the root node 10 is placed to the second position of the array, etc.

CHAPTER 7. HEAP

When we compare the index values of the left and right children with the index value of the parent, we see there is a relationship between them. Let x represent the index of an element, then the index of the left child will be $2x + 1$, and the index of the right child will be $2x + 2$. For example, the left child of the element in the third position is in the $3 \times 2 + 1 =$ seventh position, the right child is in the $3 \times 2 + 2 =$ eighth position. Or if we say reverse, let x represent the index of a child then $\lceil \frac{x-1}{2} \rceil$ will represent the index of the parent. Here $\lceil \cdot \rceil$ is the ceiling function. For example, the parent of the element 1 in the ninth position is 7 and is in the position $\lceil \frac{9-1}{2} \rceil = 4$.

Three important operations are defined on the heap structure: Extracting the largest element and deleting it, adding a new element to the structure, and changing the value of an element in the structure. Before examining these operations, we will see the definition of the heap structure in C and Java languages in Table 7.2.

The heap is structure consisting of an array field **array** of elements whose definition are given in Table 7.1 and a integer field **count** representing the number of elements in that array. When a new element is inserted into the heap, field **count** will be incremented by one, when the element with the highest priority is deleted, field **count** will be decremented by one. The function *isEmpty* checks whether the heap is empty or not. If there are no elements in the heap, the field **count** will be 0 and the heap will be empty.

7.2 Heap Operations

7.2.1 Deletion

The largest element in a max-heap is the zeroth element of the array, in other words the root node of the tree. When this element is deleted, the array must be restructured to its old max-heap state. When the largest element in the root of tree is deleted, the last element in the tree is placed to the root of the tree. But in this case, the property, that the largest element is at the top, is corrupted. In order to restore this property, the root node and one of its children must switch places. But which child, left or right? Since in the heap, each node in the tree must be larger than its left and right children, the element that will switch place with the root element must be larger than the other child. In that case, what needs to be done is, to choose the largest

7.2. HEAP OPERATIONS

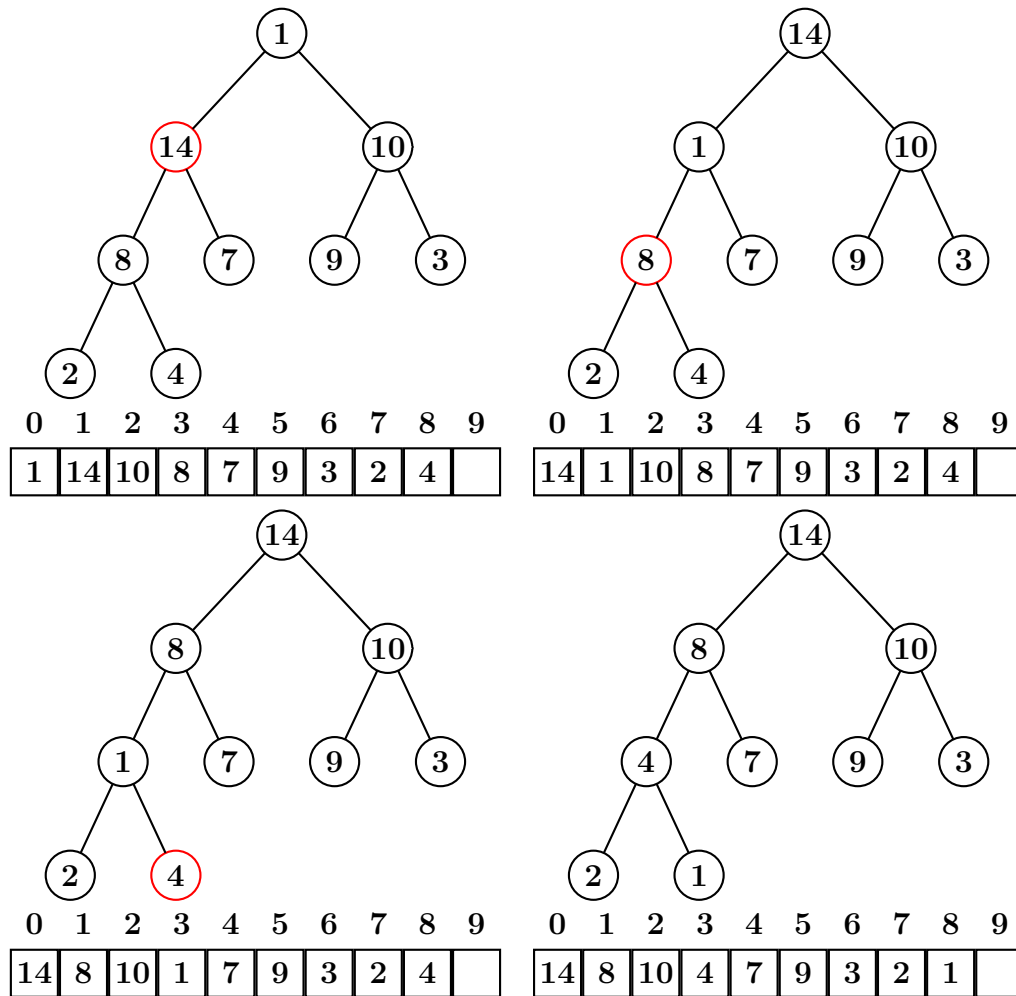


Figure 7.4: Restoring the max-heap property after removing the first element of the heap

of left and right children, and place that element to the root. In this way, the place of the element, that is placed to the root, is emptied. If we proceed similarly, we will place the largest of that element's children to that element's place and continue to the lowest level of the tree.

Figure 7.1 shows the restoring of the max-heap property after removing the first element from the heap.

CHAPTER 7. HEAP

Table 7.3: Percolating down from a node of the heap to restore the max-heap property (C++)

```
1 void Heap::swapNode(int index1, int index2){
2     HeapNode tmpNode;
3     tmpNode = array[index1];
4     array[index1] = array[index2];
5     array[index2] = tmpNode;
6 }
7 void Heap::percolateDown(int no){
8     int left, right;
9     left = 2 * no + 1;
10    right = 2 * no + 2;
11    while ((left < count && array[no].data < array[left].data) ||
12           (right < count && array[no].data < array[right].data)){
13        if (right >= count || array[left].data > array[right].data){
14            swapNode(no, left);
15            no = left;
16        }
17        else{
18            swapNode(no, right);
19            no = right;
20        }
21        left = 2 * no + 1;
22        right = 2 * no + 2;
23    }
24 }
```

- (a) The last element of the heap 1 is selected.
- (b) 1 is placed to the place of 16, the largest one of 1's left and right children, 14 is selected to be the new root node.
- (c) 14 switch place with 1, the largest one of 14's left and right children, 8 is selected to switch place with 1.
- (d) 8 switch place with 1, the largest one of 8's left and right children, 4 is selected to switch place with 1.
- (e) 4 switch place with 1.

Table 7.4 shows the function to percolate down from a node of the tree to restore the max-heap property. For this purpose, left (Line 3) or right

7.2. HEAP OPERATIONS

Table 7.4: Percolating down from a node of the heap to restore the max-heap property (Java)

```
1 void swapNode(int index1, int index2){
2     HeapNode tmpNode;
3     tmpNode = array[index1];
4     array[index1] = array[index2];
5     array[index2] = tmpNode;
6 }
7 void percolateDown(int no){
8     int left, right;
9     left = 2 * no + 1;
10    right = 2 * no + 2;
11    while ((left < count && array[no].data < array[left].data) ||
12           (right < count && array[no].data < array[right].data)){
13        if (right >= count || array[left].data > array[right].data){
14            swapNode(no, left);
15            no = left;
16        }
17        else{
18            swapNode(no, right);
19            no = right;
20        }
21        left = 2 * no + 1;
22        right = 2 * no + 2;
23    }
24 }
```

children (Line 4) are checked, if one of them is larger than the current element of the heap, the iteration continues. The iteration is, determining the largest one of the node's children and switching that node's value with the current node's value (Line 6, 10). We need to check if current node's left and right children exist or not. These checks are done with `left < heap->count` for the left child and with `right < heap->count` for the right child.

Table 7.5 shows restoring the max-heap property after deleting the first element of the heap. The function will return the first element, therefore the first element is stored in the variable `tmp` (Line 3), and the first element of the heap is set to the last element of the heap (Line 4). After that, in order to restore the max-heap property, we percolate down the tree using the function `percolateDown` (Line 5). As a last step, the number of element in the heap is decremented by one (Line 6).

CHAPTER 7. HEAP

Table 7.5: Restoring max-heap property after deleting the first element of the heap

1	<code>HeapNode Heap::deleteMax(){</code>	<code>HeapNode deleteMax(){</code>
2	<code>HeapNode tmp;</code>	<code>HeapNode tmp;</code>
3	<code>tmp = array[0];</code>	<code>tmp = array[0];</code>
4	<code>array[0] = array[count-1];</code>	<code>array[0] = array[count-1];</code>
5	<code>percolateDown(0);</code>	<code>percolateDown(0);</code>
6	<code>count--;</code>	<code>count--;</code>
7	<code>return tmp;</code>	<code>return tmp;</code>
8	<code>}</code>	<code>}</code>

7.2.2 Insertion

The insertion of a new element to the heap, contrary to the removal of the largest element, proceeds from the leaf nodes to the root node. First the new element is added to the end of the heap, then as long as the max-heap property is corrupted, the new element switches place with its parent.

In Figure 7.5 we add 12 to the heap from which we delete the zeroth element. The new element is added to the end of the heap, but in this case, the new element will be larger than its parent 7. What we should do is to switch 12 with its parent. After this switch, the new parent of 12 will be 8. Since $8 < 12$, we need again switch 12 with its parent. In the last phase, the parent of 12 is 14. Now the max-heap property of the heap is restored.

Table 7.6: The algorithm that add a new element to the heap

1	<code>void Heap::insert (HeapNode node){</code>	1	<code>void insert(HeapNode node){</code>
2	<code>count++;</code>	2	<code>count++;</code>
3	<code>array[count - 1] = node;</code>	3	<code>array[count - 1] = node;</code>
4	<code>percolateUp(count - 1);</code>	4	<code>percolateUp(count - 1);</code>
5	<code>}</code>	5	<code>}</code>

Table 7.6 shows the algorithm that restores the max-heap property after adding a new element. First, the number of elements in the heap is incremented by one (Line 3). The new element is added to the end of the heap (Line 4) and the max-heap property of the heap is restored.

Table 7.7 shows the algorithm that percolates up from a node of the tree to restore the max-heap property. As long as the max-heap property is

7.2. HEAP OPERATIONS

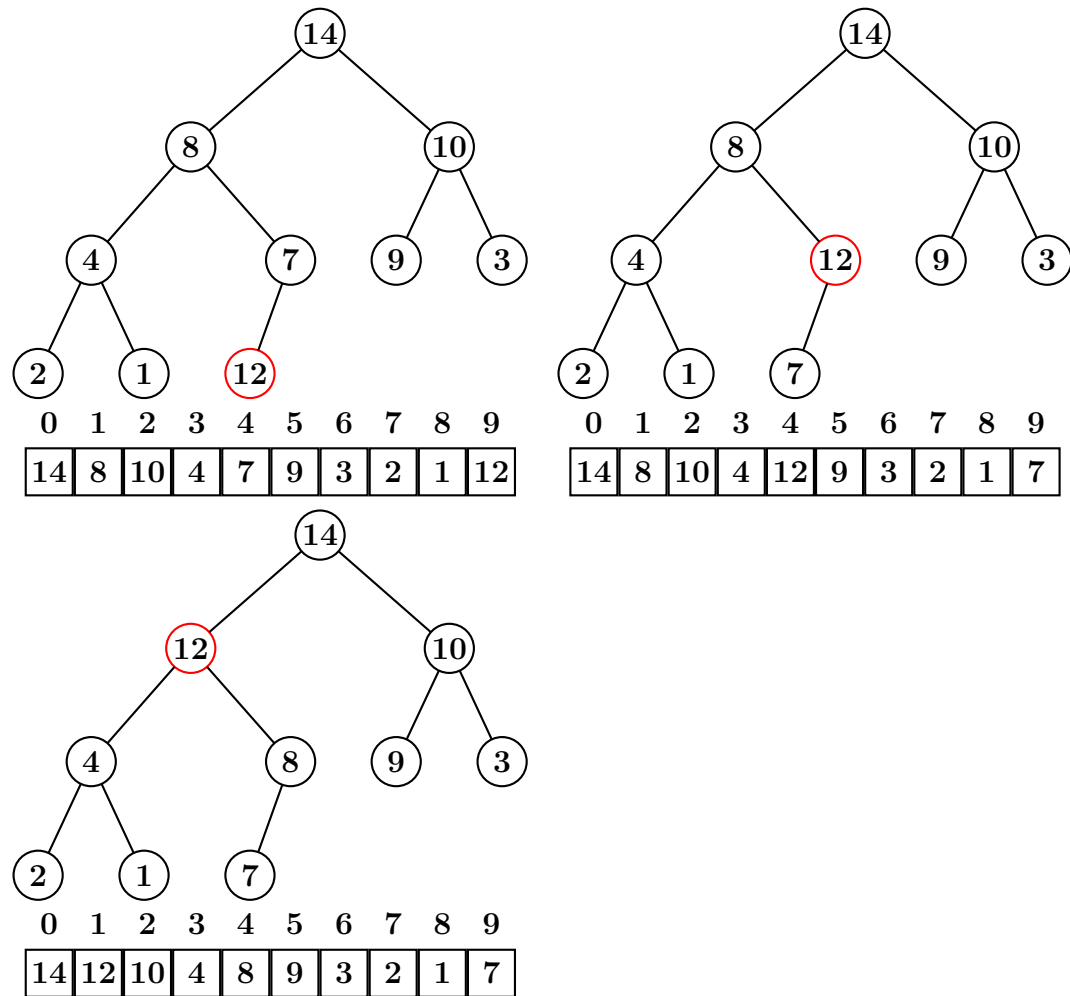


Figure 7.5: Restructring the heap after adding a new element

corrupted (Line 4), the parent (Line 7) and its child (Line 6) switch their values (Line 5). We need to pay attention that, the calculated index of the parent must be a valid number. In other words, while going upper levels, we need to see that we can not go up if we are at the root of the tree.

CHAPTER 7. HEAP

Table 7.7: Percolating up from a node of the heap to restore the max-heap property

```
1 void Heap::percolateUp(int no){
2     int parent;
3     parent = (no - 1) / 2;
4     while (parent >= 0 && array[parent].data < array[no].data){
5         swapNode(parent, no);
6         no = parent;
7         parent = (no - 1) / 2;
8     }
9 }
```

```
1 void percolateUp(int no){
2     int parent;
3     parent = (no - 1) / 2;
4     while (parent >= 0 && array[parent].data < array[no].data){
5         swapNode(parent, no);
6         no = parent;
7         parent = (no - 1) / 2;
8     }
9 }
```

7.2.3 Search

In some applications, we can search a specific element in the heap. We can search an element with respect to its value or its name. In most of the applications, since the names and values are the same, it will be more meaningful to search with respect to its name. In this way, we would also search in applications, where the name and values are different.

Table 7.8: Searching a specific element in the heap

```
1 int Heap::search(int name){
2     int i;
3     for (i = 0; i < count; i++)
4         if (array[i].name == name)
5             return i;
6     return -1;
7 }
```

```
int search(int name){
    int i;
    for (i = 0; i < count; i++)
        if (array[i].name == name)
            return i;
    return -1;
}
```

7.2. HEAP OPERATIONS

The heap data structure is not designed to search a specific element efficiently, but is designed to get the maximum priority element as fast as possible. Therefore, we can find the element we search just by looking all elements of the heap one by one. Table 7.8 shows the algorithm that searches an element with respect to its name. The name we search for is compared with the name of every element of the heap (Line 4), if there is any element whose name is equal to the name we search for, the function will return the index of that element (Line 5). If the element we search for does not exist in the heap, the function returns -1 (Line 6).

7.2.4 Update

When the value of an element in a heap is changed, there are two possibilities. The value is increased or decreased. For both cases, the new value may not have corrupted the max-heap property of the heap. In that case, there is no need to change the structure of the heap. Otherwise, we need to restore the max-heap property of the heap. Let's see these two cases separately.

In the first case, let say the value of the element was increased. As long as the value of the element is larger than the value of its parent, it must switch place with its parent. Also, when the element switch place with its parent, the element will go up one level and its parent will change. Figure 7.6 shows an example heap and the value of 14 is changed to 18. Since the value of the element is larger than the value of its parent, 16, it must switch place with its parent. The last state of the heap is given in the last part of the Figure.

In the second case, let say the value of the element is decreased. As long as the value of the element is smaller than one or two of its children, the element must switch place with the largest child of its own. Figure 7.7 shows an example heap, and the value 14 in the second level is changed to 3. This value is smaller than the values of its children (8, 7). What we should do is to switch the element, whose value is decreased, with the largest one (8). After this switch, the value of the element (3) is still smaller than the value of one of its children (4). Similarly, current element switch place with this element.

Table 7.9 shows the algorithm that restructures the heap after an update to an element. First the value of the given position is updated with the new value (Line 3), then according to the new and old values, in order to restore the max-heap property, we either percolate down (Line 5) or up (Line 7).

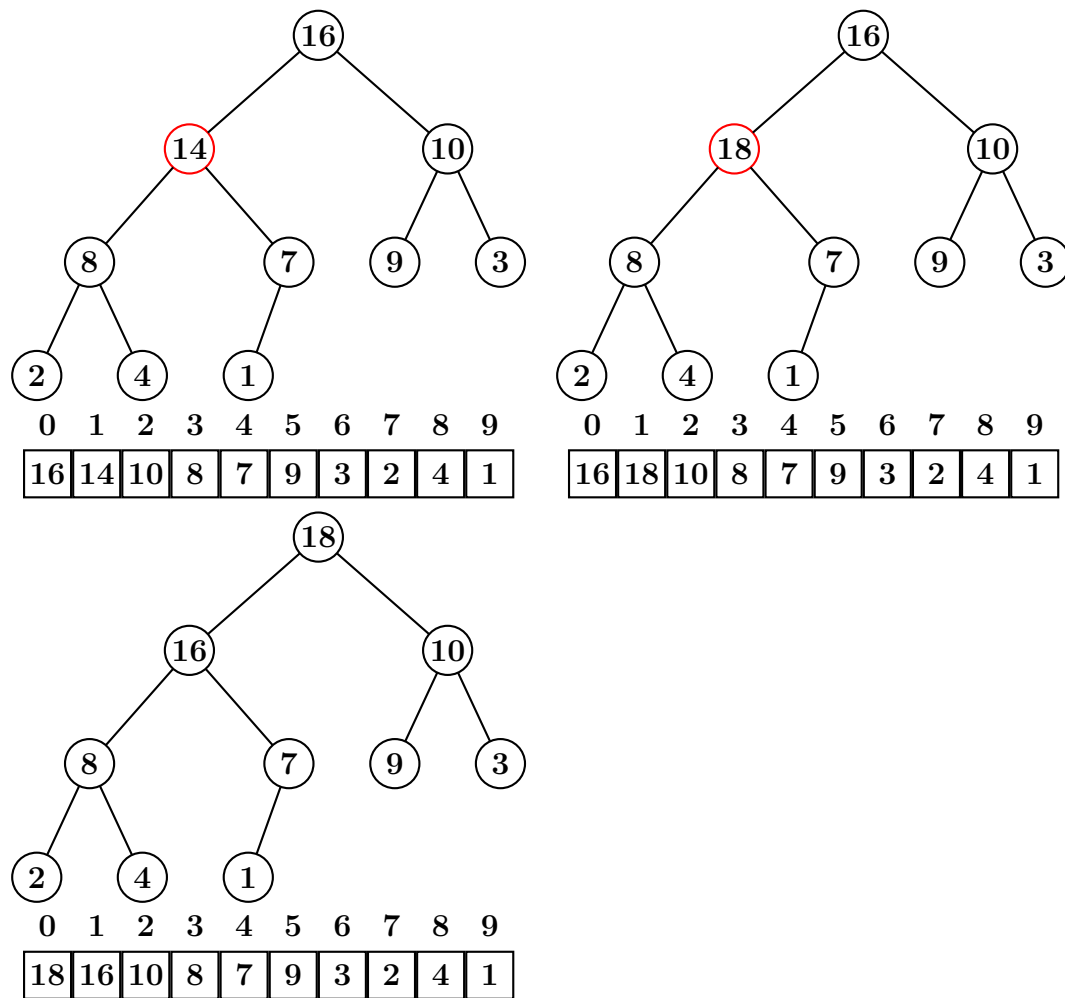


Figure 7.6: Changing the value of an element from 14 to 18

Heap Operations	
Insertion:	$\mathcal{O}(\log N)$
Deletion:	$\mathcal{O}(\log N)$
Update:	$\mathcal{O}(\log N)$
Search:	$\mathcal{O}(N)$

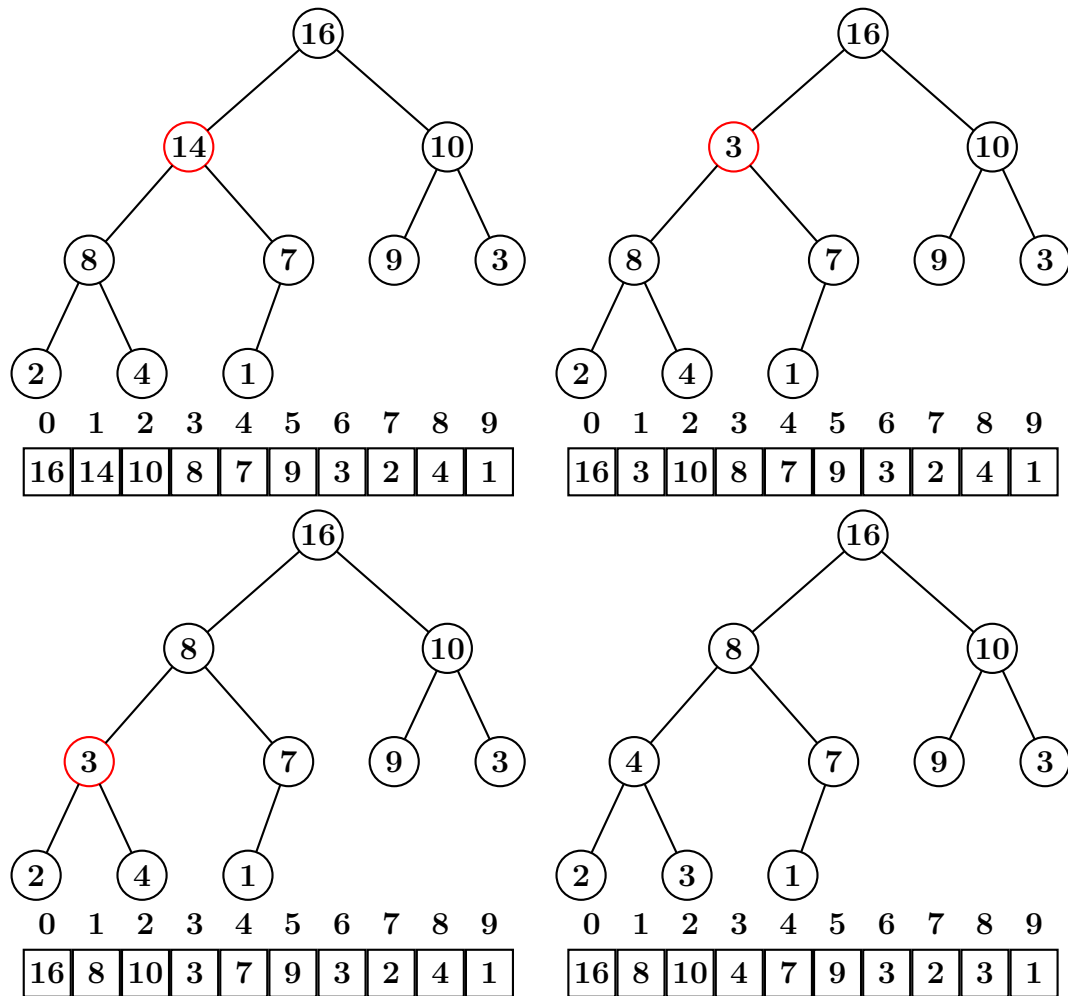


Figure 7.7: Changing the value of an element from 14 to 3

7.3 *d*-ary Heap

It is possible to generalize the binary max-heap where the properties (i) the value of each element is larger than the values of its children (for the min-heap the value of each element is smaller than the values of its children) and (ii) all levels except the last level are full, but each node has not two

Table 7.9: Updating the value of an element in the heap

<pre> 1 void Heap::update(int k, int newValue){ 2 int oldValue = array[k].data; 3 array[k].data = newValue; 4 if (oldValue > newValue) 5 percolateDown(k); 6 else 7 percolateUp(k); 8 } </pre>	<pre> void update(int k, int newValue){ int oldValue = array[k].data; array[k].data = newValue; if (oldValue > newValue) percolateDown(k); else percolateUp(k); } </pre>
---	---

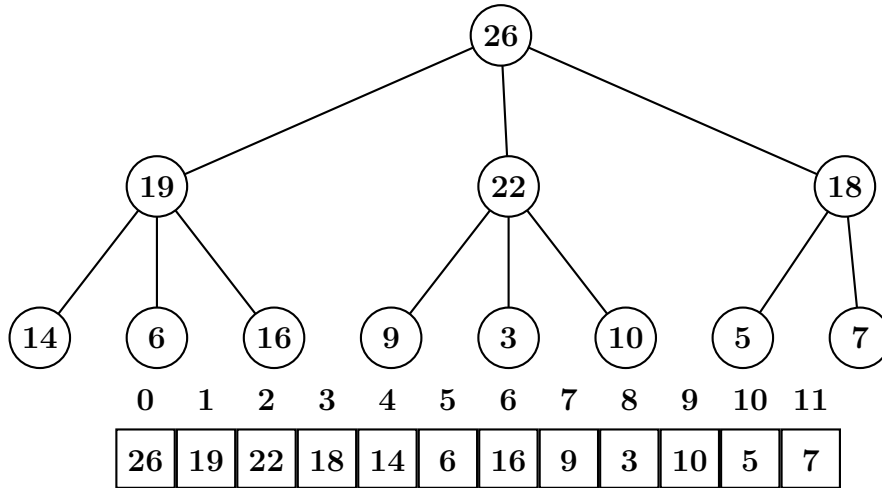


Figure 7.8: An example 3-ary heap

but d children. This type of heap is called d -ary heap in the computer science literature and the heap we have covered in the beginning of this chapter is a special case of this d -ary heap ($d = 2$).

Figure 7.8 shows an example 3-ary ($d = 3$) heap consisting of 12 elements and the array corresponding to this array. The root node of the tree is 26 and is larger than all elements. Also each node is larger than all its descendants. If we look the index of a node and the indexes of its children, we see a relationship. Let x represent the index of an element, then the indexes of its children are $3x + 1$, $3x + 2$, and $3x + 3$ respectively. For example, children of 22 in the second level are in the positions of $3 \times 2 + 1 = 7$, $3 \times 2 + 2 = 8$, and $3 \times 2 + 3 = 9$. Or if we say reverse, let x represent the index of a child

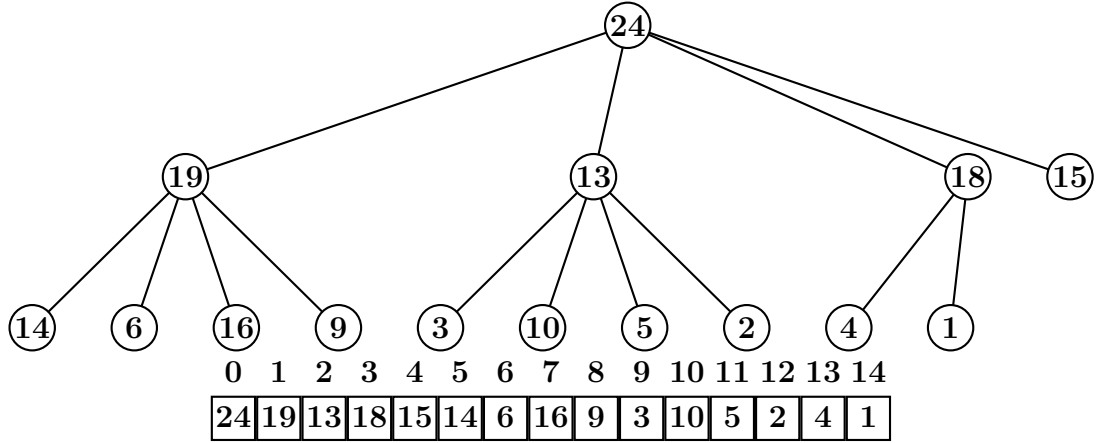


Figure 7.9: An example 4-ary heap

node, then the index of its parent is $\lceil \frac{x-1}{3} \rceil$. For example, the parent of 6 in the fifth position is 19 and its index is $\lceil \frac{5-1}{3} \rceil = 1$.

Figure 7.9 shows another example of d -ary heap. In this case, 15 elements are stored in a 4-ary max-heap. If we look the index of a node and the indexes of its children, we again see a relationship. Let x denote the index of an element, then the indexes of its children are $4x + 1$, $4x + 2$, $4x + 3$, and $4x + 4$. For example, the indexes of the children of 22 in the second level are $4 \times 2 + 1 = 9$, $4 \times 2 + 2 = 10$, $4 \times 2 + 3 = 11$, and $4 \times 2 + 4 = 12$ respectively. The relationship can be expressed in the reverse way. Let x represent the index of any node, then $\lceil \frac{x-1}{4} \rceil$ will represent the index of its parent node. For example, the parent of 9 in the eighth position is 19 and its index is $\lceil \frac{8-1}{4} \rceil = 1$.

If we generalize the relationship we see in both d -ary heap and in the normal heap, Let x denote the index of an element, then the indexes of its children are $d \times x + 1$, $d \times x + 2$, \dots , $d \times x + d$ respectively. Similarly, $\lceil \frac{x-1}{d} \rceil$ represent the index of the parent of the node whose index is x .

The definition of d -ary heap with these properties are given in Table 7.10. As in the previous heap definition, a d -ary heap consists of an array field (**array**) representing array of elements and an integer field (**count**) representing the number of elements in that array. Also the parameter d of the d -ary heap is stored in the structure.

Table 7.10: Definition of d -ary heap

<pre> 1 class DHeap{ 2 private: 3 HeapNode *array; 4 int count; 5 int d; 6 public: 7 DHeap(int N, int d); 8 ~DHeap(); 9 } 10 DHeap::DHeap(int N, int d){ 11 array = new HeapNode[N]; 12 count = 0; 13 this.d = d; 14 } 15 DHeap::~DHeap(){ 16 delete [] array; 17 } </pre>	<pre> public class DHeap{ HeapNode array[]; int count; int d; public DHeap(int N, int d){ array = new HeapNode[N]; count = 0; this.d = d; } } </pre>
---	--

7.3.1 Deletion from d -ary Heap

As in the binary heap, in d -ary heap the maximum priority element is the zeroth element of the array and also the root node of the corresponding tree. When this element is deleted, similar to the binary heap, the array must be restructured to restore its max-heap property.

If we look the deletion functions in the binary heap, we will see that, `deleteMax` does not contain any code including d or the children of the root node. The function `percolateDown` is actually containing/modifying d and the children of the root node. What we should do is, to modify `percolateDown` function, written for the binary heap, to work also for the d -ary heap and use the original `deleteMax` function in the d -ary heap.

Table 7.11 shows the function to percolate down from a node of the d -ary heap to restore the max-heap property. Starting from the root node, at each step, we need to find out which child has the largest value. Let `no` represent the index of a node, then the indexes of the children of that node are $d \times no + 1$, $d \times no + 2$, \dots , $d \times no + d$ respectively. First we are checking if there are children at these indexes (Line 5), if there are, the value of the children is compared with the current maximum value (Line 7). If the value of the child is larger than the current maximum value, the variable

Table 7.11: Percolating down from a node of the *d*-ary heap to restore the max-heap property

```

1 void DHeap::percolateDown(int no){
2     int i, child, largestChild = -1;
3     int value;
4     do{
5         value = array[no].data;
6         for (i = 1; i <= d && d * no + i < count; i++){
7             child = d * no + i;
8             if (value < array[child].data){
9                 largestChild = child;
10                value = array[child].data;
11            }
12        }
13        if (value != array[no].data){
14            swapNode(no, largestChild);
15            no = largestChild;
16        }
17    }while (value != array[no].data);
18 }

```

```

1 void percolateDown(int no){
2     int i, child, largestChild = -1;
3     int value;
4     do{
5         value = array[no].data;
6         for (i = 1; i <= d && d * no + i < count; i++){
7             child = d * no + i;
8             if (value < array[child].data){
9                 largestChild = child;
10                value = array[child].data;
11            }
12        }
13        if (value != array[no].data){
14            swapNode(no, largestChild);
15            no = largestChild;
16        }
17    }while (value != array[no].data);
18 }

```

representing the maximum value (**value**) and the variable representing the maximum value child (**largestChild**) is updated (Lines 8, 9). We continue the

CHAPTER 7. HEAP

process by switching places of the maximum value child and the node at the position numbered `no` (Lines 14, 15). If none of values of the children of the node at the position numbered `no` is larger than the value of the node at the position numbered `no`, the max-heap property of the d -ary heap has been restored (Line 17).

Figure 7.10 shows the restoring the max-heap property of the heap given in Figure 7.8, after removing its first element.

- The last element of the heap 7 is selected.
- 7 replaces 26, and the largest child of 7, namely 22 is selected as the new root node.
- 22 replaces 7, and the largest child of 7, namely 10, is selected to replace 7.
- 10 replaces 7.

Figure 7.11 shows the restoring the max-heap property of the heap given in Figure 7.9, after removing its first element.

- The last element of the heap, 1 is selected.
- 1 replaces 24, the largest child of 1, namely 19, is selected to replace the root node.
- 19 replaces 1, the largest child of 1, namely 16, is selected to replace 1.
- 16 replaces 1.

7.3.2 Insertion into d -ary Heap

The insertion of an element to a d -ary heap, similar to the binary heap, starts from the a leaf node and proceeds up to the root node. First the new element is added to the end of the heap, as long as the max-heap property of d -ary heap is not restored, the parent node and the child node switch places. If we look the insertion functions in the binary heap, we will see that, `insert` does not contain any code including d or the parent of that node. The function `percolateUp` is actually containing/modifying d and the parent of that node. What we should do is, to modify `percolateUp` function, written for the binary

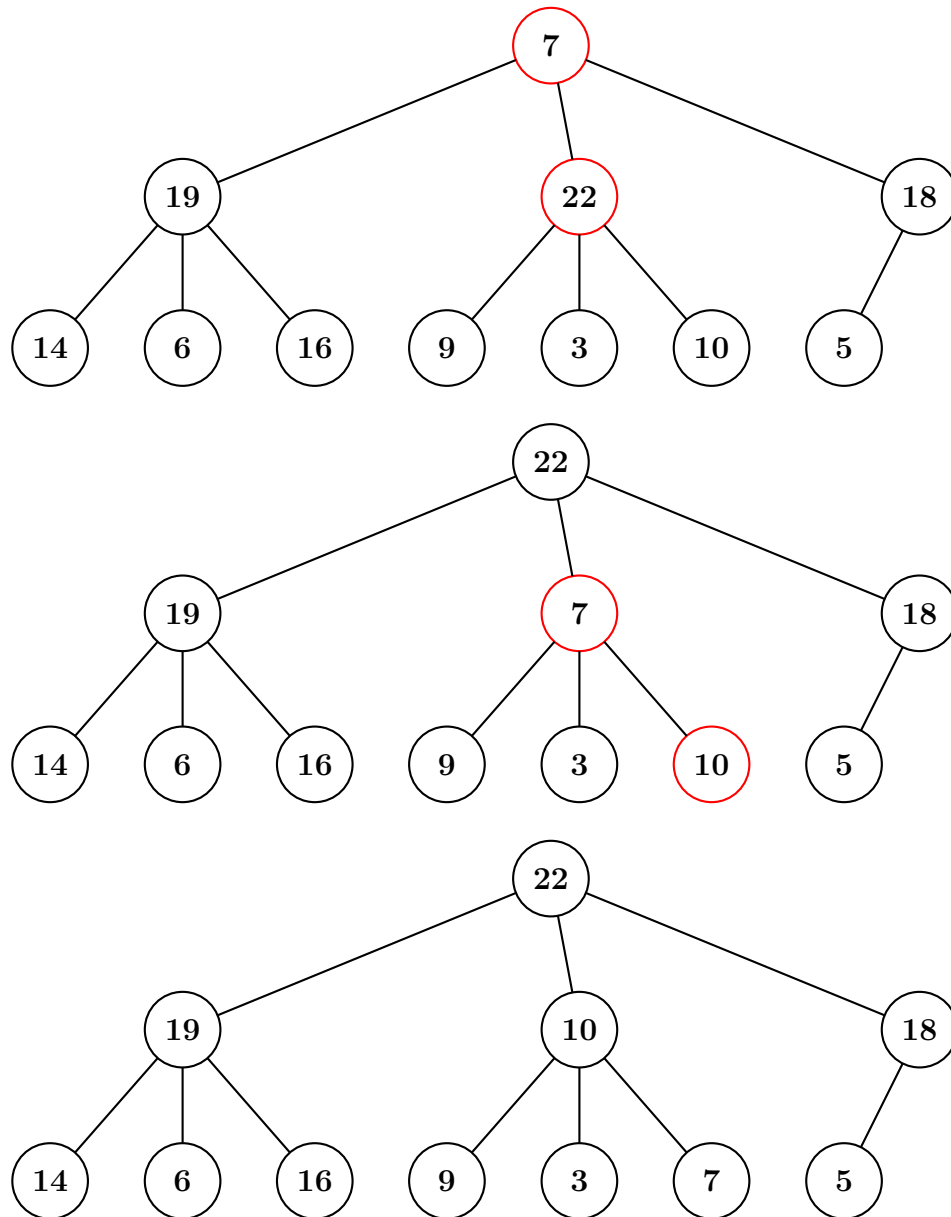


Figure 7.10: Restoring the max-heap property of the example 3-ary heap given above after removing its first element

heap, to work also for the d -ary heap and use the original `insert` function in

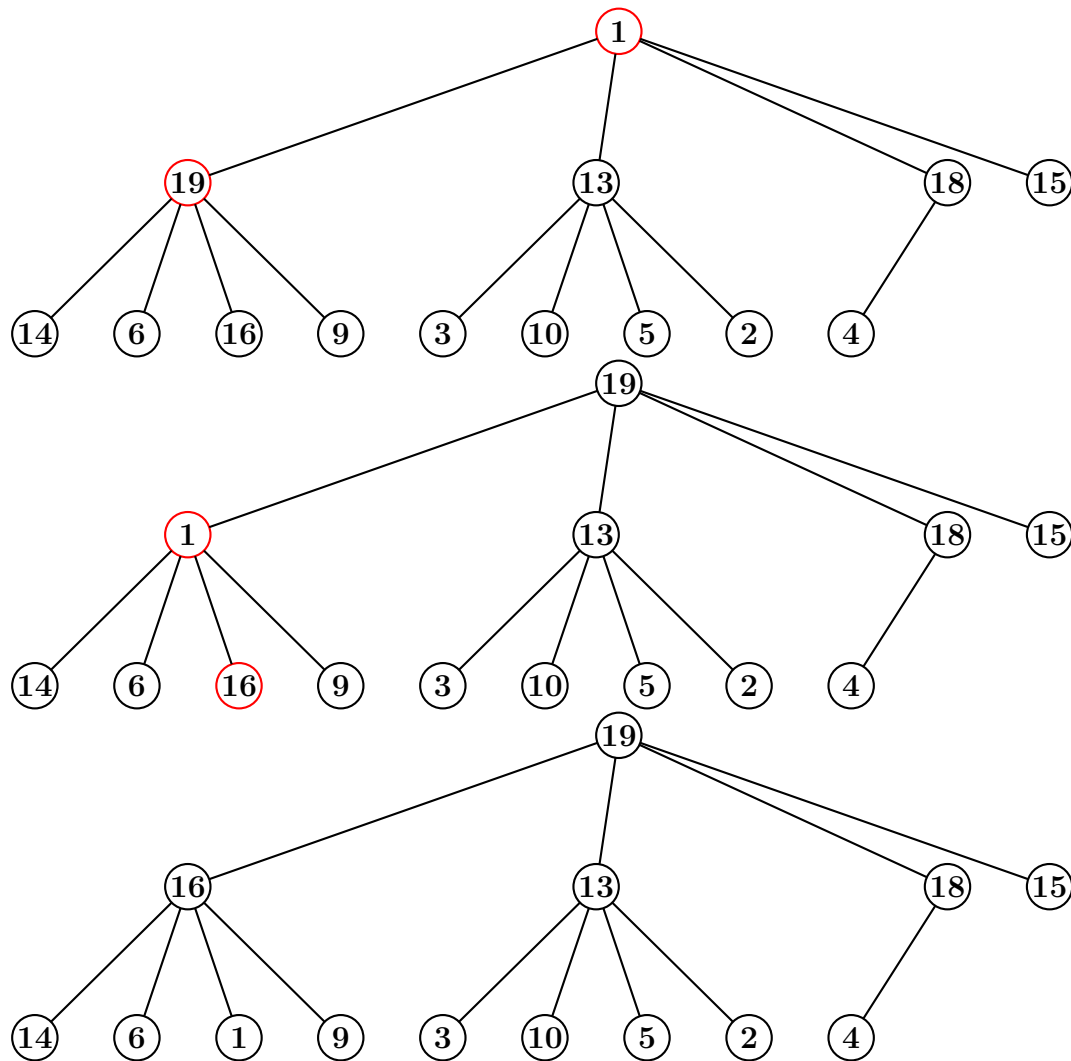


Figure 7.11: Restoring the max-heap property of the example 4-ary heap given above after removing its first element

the d -ary heap.

Table 7.12 shows the algorithm that percolates up from a node of the d -ary heap to restore the max-heap property. The only difference between the previous `percolateUp` function and this function appears in the calculation of the index of the parent node. In the binary heap, the index of the parent

Table 7.12: Percolating up from a node of the *d*-ary heap to restore the max-heap property

```

1 void DHeap::percolateUp(int no){
2     int parent;
3     parent = (no - 1) / d;
4     while (parent >= 0 && array[parent].data < array[no].data){
5         swapNode(parent, no);
6         no = parent;
7         parent = (no - 1) / d;
8     }
9 }

1 void percolateUp(int no){
2     int parent;
3     parent = (no - 1) / d;
4     while (parent >= 0 && array[parent].data < array[no].data){
5         swapNode(parent, no);
6         no = parent;
7         parent = (no - 1) / d;
8     }
9 }

```

node is calculated with $\frac{no-1}{2}$, whereas in the *d*-ary heap the index of the parent node is calculated with $\frac{no-1}{d}$ (Lines 3, 7).

In Figure 7.12, we insert 20 to the 3-ary heap given in Figure 7.8. This element is added to the end of the heap, but in this case, its parent is larger than 18. What should be done is to switch place 20 with its parent. After this switch, the new parent of 20 will be 26. Since $20 < 26$, there is no need to switch place 20 with its parent. Max-heap property of the *d*-ary heap is restored.

In Figure 7.13, we insert 20 to the 4-ary heap given in Figure 7.9. This element is added to the end of the heap, but in this case, its parent is larger than 18. What should be done is to switch place 20 with its parent. After this switch, the new parent of 20 will be 24. Since $20 < 24$, there is no need to switch place 20 with its parent. Max-heap property of the *d*-ary heap is restored.

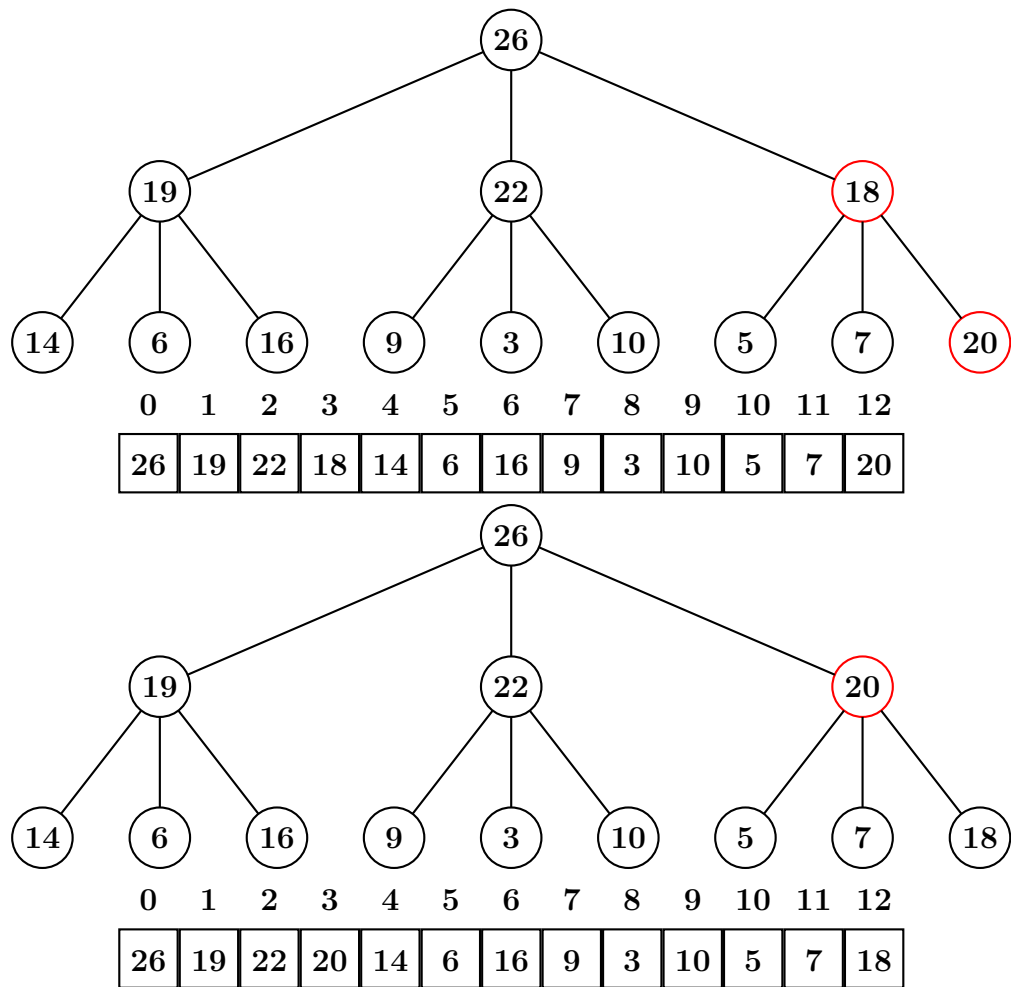






Figure 7.12: Restructuring the 3-ary heap after inserting a new element

d-ary Heap Operations	
	Insertion: $\mathcal{O}(\log_d N)$
	Deletion: $\mathcal{O}(d \log_d N)$
	Update: $\mathcal{O}(\log_d N)$
	Search: $\mathcal{O}(N)$

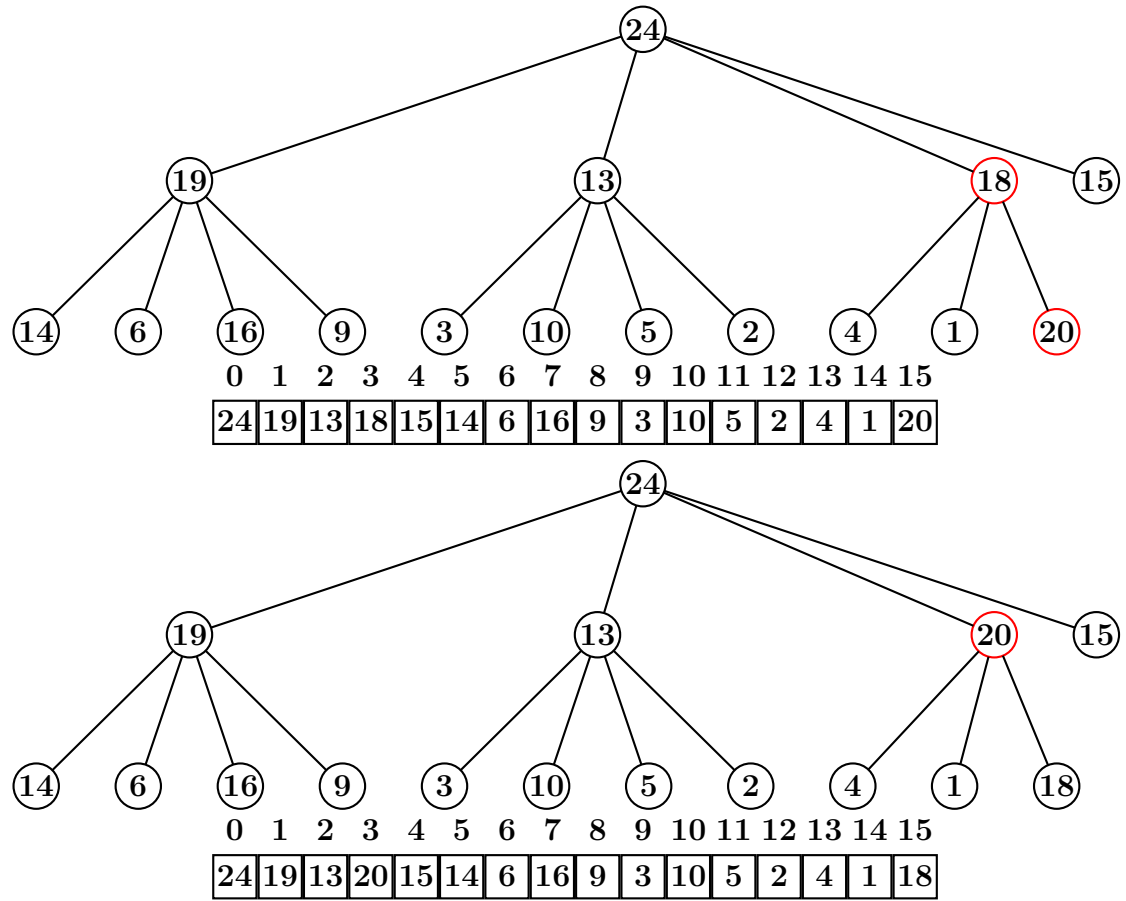


Figure 7.13: Restructuring the 4-ary heap after inserting a new element

7.4 Application: Dart

In Chapter 4 we see breadth first search algorithm to solve the dart problem. As we remember, in breadth first search algorithm, first we examine the states that can be accessed with one shot from the initial state (0), then the states that can be accessed with two shots, then the states that can be accessed with three shots, etc. and if one of these states is a final state, we will finish the search.

For example, in the Dart game given in Figure 4.7, with one shot we can

CHAPTER 7. HEAP

move from the beginning state (0) to the states 11, 21, 27, 33, and 36. On the other hand, with two shots, we can move to the states 22 ($11 + 11$), 32 ($11 + 21$), 38 ($11 + 27$), 42 ($21 + 21$), 44 ($11 + 33$), 47 ($11 + 36$), 48 ($21 + 27$), 54 ($27 + 27$ or $21 + 33$), 57 ($21 + 36$), 60 ($27 + 33$), 63 ($27 + 36$), 66 ($33 + 33$), 69 ($33 + 36$), and 72 ($36 + 36$). Figure 4.8 shows the states that can be accessed with two shots in a state graph.

In Chapter 6 we see that we can access the same state in more than one way in the dart problem and instead of adding all states to the queue, we only insert the unvisited states to the queue. In order to accomplish this, we stored the visited states in an hash table. With the help of the hash table, we can search currently accessed states and decide if they are visited or not in constant time and if current state is a new state (not visited) we can also insert this new state to the hash table in constant time.

Although using the hash table we can get rid of inserting unnecessary states to the queue and therefore solve the dart problem a lot faster, we can insert the states to the queue not in a given order but in the order of they appear in the dart board. For example, if the numbers in the dart board are 11, 21, 27, 33, and 36 respectively, we first insert the states that can be accessed with one dart from 11, then the states that can be accessed with one dart from 21, then the states that can be accessed with one dart from 27,

Similarly when we remove states from the queue to process, due to the nature of the queue, they are removed in the order of they are inserted. For example, if the numbers in the dart board are 11, 21, 27, 33, 36 respectively, first we remove and process the states that can be accessed with one dart from 11, then the states that can be accessed with one dart from 21, then the states that can be accessed with one dart from 27,

However, if we can process the states whose sums are near to 100, we can attain the final state more quickly. This can only be possible in the heap data structure, where the states have priorities with respect to each other. Instead of inserting the states to an ordinary queue but to a priority queue (heap), and remove it from the heap when we want to process, we will not see the disadvantages of the ordinary queue.

The solution to the Dart game problem using breadth first search which also holds the unprocessed states in a heap is given in Table 7.14. As the first step, the beginning state, the empty board state (Line 8), is added as the first element to the heap (Line 9). Since this state is already visited, it is also inserted into the hash table (Line 12). At each stage, one state is

Table 7.13: Solving the Dart game problem using breadth first search (3)
(C++)

```

bool dartGame(int* board){
    int i, t;
    HeapNode e;
    Heap y;
    Node *o;
    Hash kt;
    String a;
    e = HeapNode(State(0, ""), 0);
    y = Heap(100);
    y.insert(e);
    o = new Node(0);
    kt = Hash(1000);
    kt.insert(o);
    while (!y.isEmpty()){
        e = y.deleteMax();
        if (e.data.total == 100)
            return true;
        for (i = 0; i < 5; i++){
            t = e.data.total + board[i];
            if (t <= 100)
                if (kt.search(t) != nullptr){
                    a = e.data.darts;
                    a = a + "," + board[i];
                    e = HeapNode(State(t, a), t);
                    y.insert(e);
                    o = new Node(t);
                    kt.insert(o);
                }
        }
    }
    return false;
}

```

removed from the heap (Line 14), if the sum of the shots of this state is 100 (Line 15), the function returns the shots list of this state (Line 16). For each state retrieved from the heap, the states that can be reached from that state are analyzed (Line 17). If the reachable state

- is a defined state, that is, the sum of its shots is less than or equal to

CHAPTER 7. HEAP

Table 7.14: Solving the Dart game problem using breadth first search (3)
(Java)

```
static boolean dartGame(int[] board){
    int i, t;
    HeapNode e;
    Heap y;
    Node o;
    Hash kt;
    String a;
    e = new HeapNode(new State(0, ""), 0);
    y = new Heap(100);
    y.insert(e);
    o = new Node(0);
    kt = new Hash(1000);
    kt.insert(o);
    while (!y.isEmpty()){
        e = y.deleteMax();
        if (e.data.total == 100)
            return true;
        for (i = 0; i < 5; i++){
            t = e.data.total + board[i];
            if (t <= 100)
                if (kt.search(t) != null){
                    a = e.data.darts;
                    a = a + "-" + board[i];
                    e = new HeapNode(new State(t, a), t);
                    y.insert(e);
                    o = new Node(t);
                    kt.insert(o);
                }
        }
    }
    return false;
}
```

100 (Line 19)

- and we haven't visited the state yet, in other words if this state does not exist in the hash table (Line 20)

new state is added to the heap (Lines 21-22) and the hash table (Lines 23-24).

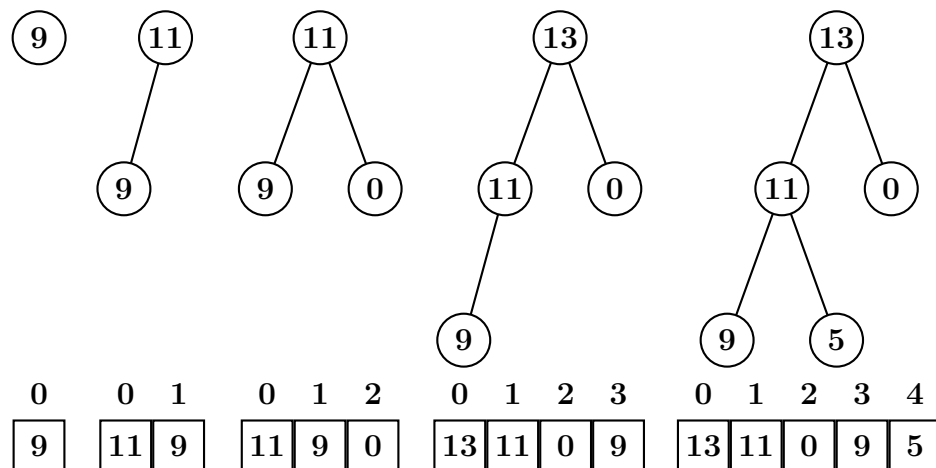
7.5 Notes

Heap structure is first defined in [16]. The linear time heap construction algorithm is taken from [5].

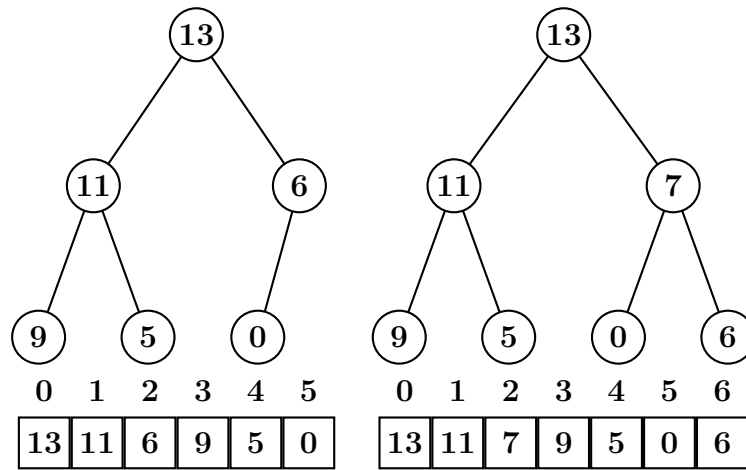
7.6 Solved Exercises

1. Show the result of inserting 9, 11, 0, 13, 5, 6, 7 into an initially empty binary heap.

The heap structure after 9, 11, 0, 13, 5, 6, 7 inserting 9, 11, 0, 13, 5, 6, 7 into an initially empty binary heap is shown below.



CHAPTER 7. HEAP



2. Write a function that finds the minimum item in a binary heap.

```
int smallest()
```

The function that finds the minimum item in a binary heap is given below. In a max heap, the minimum element is always in the second part of the corresponding array structure. That is because, each element in the first part of the array has a child and by the definition of the max-heap, the children of a node are always smaller than that node. The function uses this idea and simply find the minimum element in the second part of the array. The index of the middle element of the array can be found with `s->tane / 2` (Line 3). The rest of the array is compared with this element (Lines 4-5) and if a smaller element is found the new minimum element will be this smaller element (Line 6).

```
int smallest(){
    int i;
    int smallest;
    smallest = array[count / 2].data;
    for (i = count / 2 + 1; i < count; i++)
        if (array[i].data < smallest)
            smallest = array[i].data;
    return smallest;
}
```

3. Write a function that finds the k 'th largest element of an array using heap.

```
int kthLargest(int [] A, int N, int k)
```

The function that finds the k 'th largest element of an array using heap is given below. First all elements in the array (Line 5) are added to the heap (Lines 6, 7). When the function that returns the largest element is called once, it will return us the largest element, if it is called twice it will return us the second largest element, ..., if it is called k times it will return us the k th largest element (Lines 9, 10).

```

1  T kthLargest(T[] A, int N, int k){
2      Heap<T> y;
3      HeapNode<T> e = null;
4      int i;
5      y = new Heap<T>(N, comparator);
6      for (i = 0; i < N; i++){
7          e = new HeapNode<T>(A[i], i);
8          y.insert(e);
9      }
10     for (i = 0; i < k; i++)
11         e = y.deleteMax();
12     return e.data;
13 }
```

7.7 Exercises

1. Show the result of performing two deleteMax operations in the heap of solved exercise 1.
2. Write a function that prints all nodes in a binary heap larger than a given value X .

```
void printLargerThanX(int x)
```

3. Implement heap data structure using explicit links for children.
4. If the heap is implemented using explicit links, write a function that finds the node at position k .

```
Node getKth(int i)
```

5. Write a function that finds the k 'th maximum element in a binary heap.

CHAPTER 7. HEAP

`HeapNode kthMaximum(int k)`

6. Given an array A , write a function that determines if this array satisfies heap-order property.

`boolean heapOrder(int[] A, int N)`

7. Given the index of a heap node, write a function that returns the maximum child of this node.

`HeapNode largestChild(int index)`

8. Given the index of a heap node, write a function that swaps this node with its smallest child.

`void swapWithSmallest(int index)`

9. Given the index of a heap node, write a function that obtains the depth of this node. The depth of root node is 0, the depth of the node in the second level are 1, etc.

`int depth(int index)`

10. Given the index of a heap node, write a function that obtains the height of this node. The height of a node is the length of the longest path from that node to a leaf.

`int height(int index)`

11. Write a function that returns the index of the maximum valued grand-child (children of children) of a heap node given its index.

`int maxGrandChild(int no)`

12. Write a function that returns the index of a heap node's grandparent (parent of parent) given its index.

`int grandParent(int no)`

7.8 Problems

1. Given the index of a heap node in a max-heap, write a function that returns the minimum descendant of this node.

`HeapNode minDescendant(int index)`

- Given the index of a heap node in a max-heap, write a recursive function that returns the sum of all descendants of this node.

`double sumOfDescendants(int index)`

- Given the index of a d-heap node, write a method that returns the number of descendants (children, grandchildren, grandgrandchildren, etc.) of that heap node. Do not use any class or external methods.

`int descendants(int no)`

- Given the index of a heap node in a max-heap, write a method that determines if the subtree rooted from the node with index satisfies the heap-order property. Do not use any class or external methods.

`boolean heapOrder(int index)`

- Given index of a d-heap node and a level l , write a recursive function that returns the number of descendants at level l of that heap node. Level 1 corresponds to children, Level 2 corresponds grandchildren, Level 3 corresponds grand grand children of that heap node.

`int descendants(int no, int level)`

- Given an array of N integers, find the k 'th maximum of those integers in $\mathcal{O}(N \log K)$ time. (Hint: Use a min-heap to store K largest elements at a time, in that case the removeMin will return the k 'th maximum).

`int kthMaximum(int[] array, int k)`

- Given the index of a heap node in a binary max heap, write the method in **MaxHeap** class

`int* ascendants(int index)`

that returns the indexes of its ascendants (parent, grandparent, grand-grandparent, ...) of this node. The array should contain only that many items not more not less.

- Write the method in **MinDHeap** class

`int numberOfPlacesToReplace(int key)`

that finds the number of places in the heap, which can be replaced with the given value key.

CHAPTER 7. HEAP

9. Write the method in **MinHeap** class

int howManyChildrenCanBeSwapped()

that returns the number of nodes in the heap whose children can be swapped without hurting the heap property. Your method should run in $\mathcal{O}(N)$ time.

10. Write the method in **MaxDHeap** class

int third()

that returns the third maximum number in the heap. Your method should run in $\mathcal{O}(d^2)$ time.

11. Write the method in **MaxHeap** class

int shortestDistanceBetweenNodes(**int** index1, **int** index2)

that returns the shortest distance between two nodes in the heap with indexes index1 and index2. Generate the ascendant lists of the nodes with index1 and index2 (You can assume the tree depth is smaller than 100). Compare those lists to solve the problem.

12. Write the method in **MinDHeap** class

int sumOfMaxChange(**int** min, **int** max)

that returns the sum of the lengths of the change intervals of each node. A change interval of a node N is defined with the minimum and maximum values that can be attained by N without changing any other node. For root node, take the minimum as min . For leaf nodes, take the maximum as max .

13. Write the method in **MinHeap** class

bool isLargestLeftMost()

which returns true if the largest value appears on the leftmost node. You may not use any additional data structures.

14. Write the method in **MaxDHeap** class

int howManyPairCanBeSwapped()

that returns the number of node pairs in the heap which can be swapped without hurting the heap property. Your method should run in $\mathcal{O}(N^2)$ time. You may not use any additional data structures.

7.8. PROBLEMS

Disjoint Set

Some applications require to represent N elements as sets. In the first stage of the program, each set contains one element, as the program progresses the sets of the elements with similar properties are merged. In these applications, the sets are always disjoint because an element belongs to one and only one set. Since each element belongs to one set, the operation where we search the set of the element is an important operation defined for the disjoint set data structure.

8.1 Definition

If we represent each disjoint set with a tree structure, all sets will be represented by a forest. Each tree in this forest will represent a set. Each node of the tree will represent an element and each node (except the root node) will have a parent that is one step closer to the root node of the tree.

Figure 8.1 shows an example disjoint set structure which consists of 8 elements and 3 disjoint sets. The first set consists of elements 0, 2, 3, and 6; second set consists of 1, 5, and 7 and third set consists of only 4. In the first set, the parent of 6 is 3, the parents of 3 and 0 is 2. In the second set, the parent of 7 is 5, the parent of 5 is 1. Another important point is that the numbers in the root nodes of the trees are also the parents of themselves. This is basically shown by drawing an arc to themselves. For example, in the first set 2, in the second set 1 and in the third set 4 is the parent of itself.

Then with which data structure will we represent these sets? Since each

8.1. DEFINITION

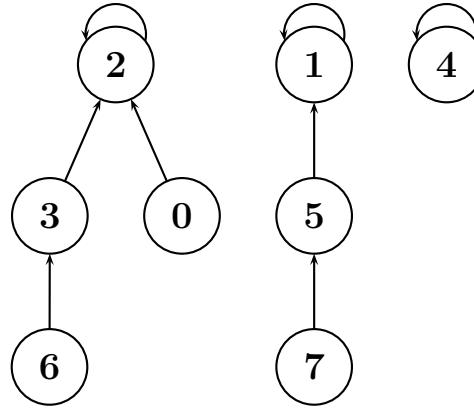


Figure 8.1: An example disjoint set data structure containing 8 elements

	0	1	2	3	4	5	6	7
Data	0	1	2	3	4	5	6	7
Parent	2	1	2	2	4	1	3	5
Depth	1	3	3	2	1	2	1	1

Figure 8.2: The array representation of a disjoint set structure containing 8 elements and 3 sets

number has a parent, a value, and a depth, it will be more suitable to represent each number in the data structure as a separate structure. The index information of the elements in the array will help us to represent the parent information of those elements. Figure 8.2 shows the array representation of the example disjoint set we have given before. In the Figure, **data** field shows the value of current element, **parent** field shows the index of the parent of the current element in the array, **depth** shows the depth of the subtree rooted the current element. For example,

- 3 is stored in the position 3, its parent is in position 2 (number 2) and

Table 8.1: The definition of a set

<pre> class Set{ int data; int parent; int depth; Set(int data, int index); } Set::Set(int data, int index){ this->data = data; parent = index; depth = 1; } </pre>	<pre> public class Set{ int data; int parent; int depth; public Set(int data, int index){ this.data = data; parent = index; depth = 1; } } </pre>
--	---

its depth is 2.

- 1 is stored in the position 1, its parent is in position 1 (itself) and its depth is 3.
- 4 is stored in the position 4, its parent is in position 4 (itself) and its depth is 1.

In that case, there are three fields in the structure (data, position of the parent, and depth). Table 8.1 defines this data structure.

The depth of a given set is the longest path from the representative root node of that set to its leaves. For example, in Figure 8.1, the depth of the sets with representatives 1 and 2, is 3, whereas the depth of the set with representative 4 is 1. That is because, the lengths of the paths from the set with representative 1, to its leaves (6 and 0) are 3 and 2 respectively. The longest one of these paths is 3 and therefore the depth of the set with representative 1 is 3. Since the set with representative 4 has only one leaf node and is also itself, its depth is 1.

We can store all sets in an array. The index information of sets will help us in representing the parents of those sets. Table 8.2 shows the disjoint set data structure where the sets are stored. `disjointSet` structure contains an array where each element is a set (sets). The number of elements in the array `sets` is given with the field `count`. The sets are defined with values between 0 and $N - 1$ and stored also in the addresses between 0 and $N - 1$.

The operations defined for the disjoint set data structure are, finding the set of an element and merging two disjoint sets.

Table 8.2: The definition of the disjoint set data structure which has a set as its own element

```

class DisjointSet {
    Set *sets;
    int count;
    DisjointSet (int* elements, int count);
}
DisjointSet :: DisjointSet (int* elements, int count){
    int i;
    sets = new Set[count];
    for (i = 0; i < count; i++)
        sets[i] = Set(elements[i], i);
}
}

```

```

public class DisjointSet{
    Set sets [];
    int count;
    public DisjointSet(int[] elements, int count){
        int i;
        sets = new Set[count];
        for (i = 0; i < count; i++)
            sets[i] = new Set(elements[i], i);
    }
}

```

8.2 Disjoint Set Operations

8.2.1 Finding Set of An Element

In this operation, the aim is to find the set of an element with a given index. Each element belong to one disjoint set. In each disjoint set, there can be one or more elements. These elements are connected to each other with the parent relationship. The element whose parent equals to itself is selected as the set index of that disjoint set. For example, in Figure 8.1 the set of element 3 is 2, the set of 5 and 7 is 1. When we are given an element and if we are asked what is the set of this element, we first check if that element is the representative of the set it belongs to. If it is, we have already found the set. If it is not, we continue with the parent of that element and

CHAPTER 8. DISJOINT SET

then the grandparent of that element until we find an element who is the representative of that disjoint set.

Table 8.3: The function that returns the index of a set of an element with a given index

<pre>int DisjointSet :: findSet(int index){ if (sets[index].parent != index) return findSet(sets[index].parent); return sets[index].parent; }</pre>	<pre>int findSet(int index){ if (sets[index].parent != index) return findSet(sets[index].parent); return sets[index].parent; }</pre>
--	---

Table 8.3 shows the operation explained above. The function is a recursive function. If the index of the parent of the current element is not equal to the index of current element (Line 2) the function calls itself with the index of the parent of current element (Line 3). If they are the same, we have found the index of the set, the function will return the index of the parent (Line 4).

8.2.2 Merging Two Disjoint Sets

The aim is to find the indexes of the sets of two elements with the given indexes, then merge these two sets. When we merge two sets to construct the resulting set, the parent of one set will be the parent of the second set. Which set will be the parent set depends on the depth of the two sets. There are two possible cases:

- The set with lower depth will be the child of the set with higher depth. For example, on the left hand side of Figure 8.3 the set with index 4 is the child of the set with index 2. In this case, the depth of the resulting set will not be changed.
- The set with higher depth will be the child of the set with lower depth. For example, on the right hand side of Figure 8.3 the set with index 2 is the child of the set with index 4. In this case, the depth of the resulting set will be increased by 1.

Naturally, not changing the depth is preferred. Otherwise, the balanced property of the tree will be corrupted and the time complexity of finding the set of an element will not be logarithmic but linear.

8.2. DISJOINT SET OPERATIONS

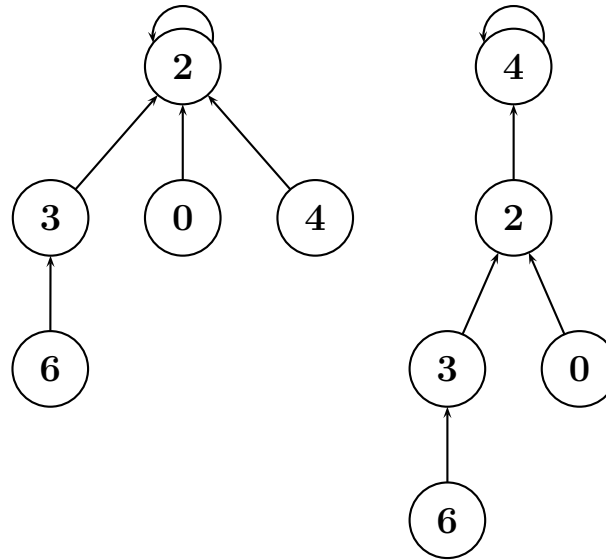


Figure 8.3: Merging two sets with indexes 2 and 4 in Figure 8.1 in two different ways

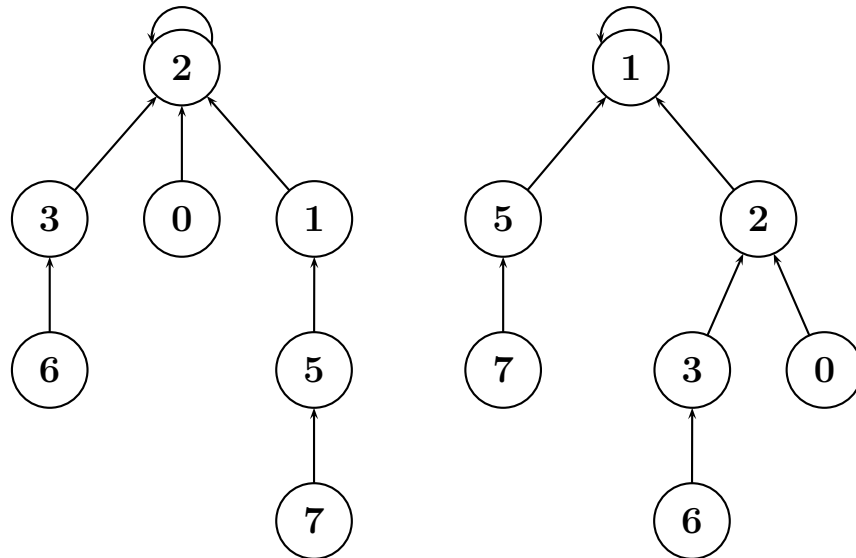


Figure 8.4: Merging two sets with indexes 1 and 2 in Figure 8.1 in two different ways

CHAPTER 8. DISJOINT SET



Table 8.4: The function that merges two sets given their indexes

<pre> void DisjointSet :: union(int index1, int index2){ int x, y; x = findSet(index1); y = findSet(index2); if (sets[x].depth < sets[y].depth) sets[x].parent = y; else{ sets[y].parent = x; if (sets[x].depth == sets[y].depth) sets[x].depth++; } } </pre>	<pre> void union(int index1, int index2){ int x, y; x = findSet(index1); y = findSet(index2); if (sets[x].depth < sets[y].depth) sets[x].parent = y; else{ sets[y].parent = x; if (sets[x].depth == sets[y].depth) sets[x].depth++; } } </pre>
---	--

Table 8.4 shows the function that merges the sets of two elements. The function first finds the sets of those two elements (Lines 3-4), then checks the depths of those two elements (Lines 5, 7). The set of lower depth will be the child of the set of higher depth (Lines 6, 8). If the depths of the two sets are equal to each other (Line 9), when we merge two sets, the depth of the parent set will be increased by one (Line 10).

If the depth of the two sets are equal to each other, whatever we do, the depth of the resulting set will increase by 1. In this case, it does not make a difference which set is the parent of which set. On the left hand side of Figure 8.4, we see merging two sets where the set with index 1 is the child of the set with index 2, on the right hand side of Figure 8.4, we see merging two sets where the set with index 2 is the child of the set with index 1. Since the depths of both sets are 3, the depth of the resulting set will be one more, that is, 4.

Disjoint Set Operations

-  Find Set: $\mathcal{O}(\log N)$
-  Merge Sets: $\mathcal{O}(\log N)$

8.3 Application: Cracking the Code

In this application, there are two strings consisting of uppercase and lowercase letters. In these strings, the uppercase letters are encrypted, and must be replaced by one of the lowercase letters. The aim of this application is to replace each variable (uppercase letter) with such a lowercase letter that both strings will be equal. For example, if the two strings are

```
AbcDbcE  
gbchbcj
```

then in order to be those two strings equal, variable A must take the value g, the variable D must take the value h and the variable E must take the value j. On the other hand, it is possible to see more complex cases. For example, if the two strings are

```
ABCEGFE  
BCdFHGa
```

then, we understand

- from the first character that the variables A and B are equal
- from the second character that the variables B and C are equal, therefore the variables A, B, and C are all equal
- from the third character that the value of variable C is d, therefore the values of the variables A, B, and C are d

On the other hand, we see

- from the fourth character that the variables E and F are equal
- from the fifth character that the variables G and H are equal
- from the sixth character that the variables F and G are equal, therefore the variables E, F, G, and H are all equal
- from the seventh character that the value of variable E is a, therefore the values of the variables E, F, G, and H are a

CHAPTER 8. DISJOINT SET

After we pass one time through two strings, we should check if there is an anomalous case to equalize two strings by using the equalities between the uppercase letters. What kind of anomalous case can happen? For example, if the value of the variable A is b and the variables A and B are equal, the variables B and C are equal, and if the value of the variable C is c, then the value of the variable A will be both b and c. In other words, if one of the equal variables is equal to one lowercase letter, and the other one of the equal variables is equal to another lowercase letter, those two strings can not be equal to each other. An easy way to understand this is to assume each variable as a set and merge the sets of the equal variables. After merging all equal variables, all sets are checked if the values of the variables inside each set is equal or not. If the values of two elements in a set are not equal, those two strings are not equal to each other. If this is not the case, we can decide that those two strings are equal to each other.

Table 8.5 shows the solution to the cracking code application. The characters of each string is compared one by one (Line 5). The first letter of the first string is compared with the first letter of the second string, the second letter of the first string is compared with the second letter of the second string, etc. Three cases can occur in those comparisons:

1. Both letters can be lowercase (Line 9).
2. One letter can be uppercase and the other lowercase (Lines 12, 20).
3. Both letters can be uppercase (Line 26).

If both letters are lowercase, these two letters are equal or not (Line 10). If the letters are not equal, those two strings are not equal (Line 11). If the letters are equal, those two strings may be equal.

If one of the letters is uppercase and the other is lowercase (Line 12, 20), the value of the uppercase letter must equal to the lowercase letter. If the variable represented by the uppercase letter is not used before, in other words, its value is not equalized to a lowercase letter, there is no problem (Lines 16, 24), the value of the uppercase letter is set to the lowercase letter (Line 17, 25). If the variable represented by the uppercase letter is used before and set to another lowercase letter, since a variable can not be equal to two different lowercase letters, these two strings can not be the same (Lines 15, 23). If the variable represented by the uppercase letter is used before and set to the same lowercase letter, there is no problem, nothing will be done.

8.3. APPLICATION: CRACKING THE CODE

Table 8.5: The application of crack the code (Java)

```
boolean crackCode(String first, String second){
    int[] chars = new int[26];
    for (int i = 0; i < 26; i++){
        chars[i] = -1;
    }
    DisjointSet a = new DisjointSet(chars);
    for (int i = 0; i < first.length(); i++){
        int set1 = a.findSet( first.charAt(i) - 'A');
        int set2 = a.findSet(second.charAt(i) - 'A');
        if ( first.charAt(i) >= 'a' && first.charAt(i) <= 'z')
            if (second.charAt(i) >= 'a' && second.charAt(i) <= 'z')
                if (first.charAt(i) != second.charAt(i))
                    return false;
            else{
                if (a.sets[set2].data != -1){
                    if (a.sets[set2].data != first.charAt(i) - 'a')
                        return false;
                }else
                    a.sets[set2].data = first.charAt(i) - 'a';
            }
        else
            if (second.charAt(i) >= 'a' && second.charAt(i) <= 'z'){
                if (a.sets[set1].data != -1){
                    if (a.sets[set1].data != second.charAt(i) - 'a')
                        return false;
                }else
                    a.sets[set1].data = second.charAt(i) - 'a';
            }else{
                a.union(set1, set2);
                if (a.sets[set1].data != a.sets[set2].data)
                    if (a.sets[set1].data == -1)
                        a.sets[set1].data = a.sets[set2].data;
                    else
                        if (a.sets[set2].data == -1)
                            a.sets[set2].data = a.sets[set1].data;
                        else
                            return false;
            }
    }
    return true;
}
```

If both letters are uppercase letters, the lowercase letter values of those two variables must be the same. If one of those two variables are set to a lowercase letter before (Lines 29, 32), the value of the variable, which is not set, is set to the value of the variable, which was set to a value before (Lines 30, 33). If the values of both variables are set before and those values are different, then two strings can not be the same (Line 35).

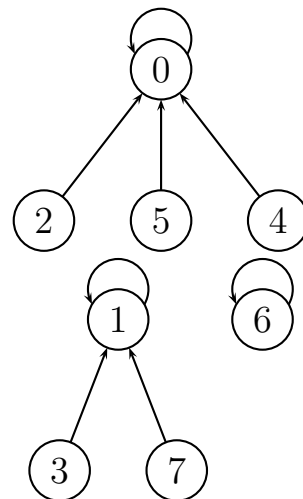
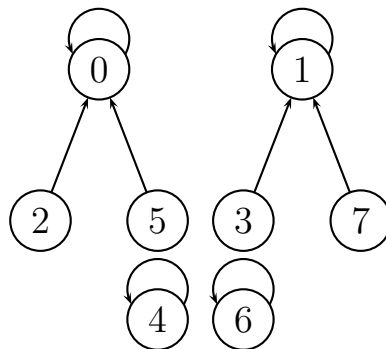
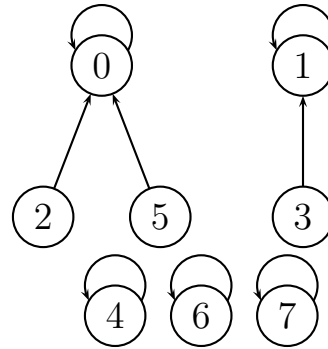
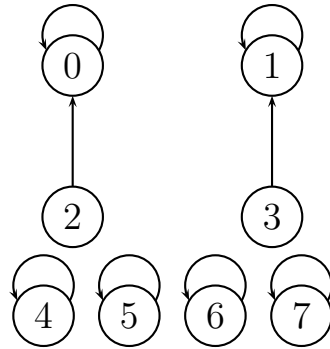
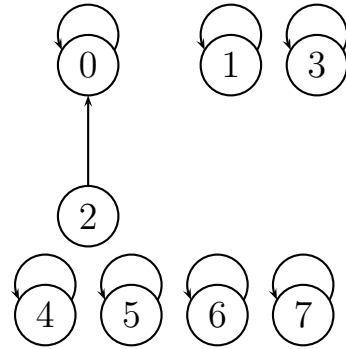
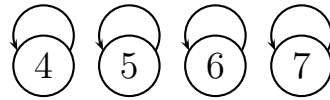
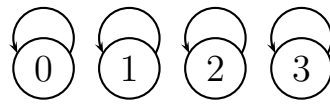
8.4 Solved Exercises

1. Show the result of the following sequence of operations. The unions are performed by depth.

```
for (int i = 0; i < 8; i++)
    numbers[i] = i;
DisjointSet k = new DisjointSet(numbers);
k.union(0, 2);
k.union(1, 3);
k.union(2, 5);
k.union(1, 7);
k.union(4, 0);
```

The states of the disjoint sets after each operation is given in the Figure below. At the first step each number is itself a disjoint set, in the second step set 0 and 2 are merged, in the third step sets 1 and 3 are merged. In the fourth step, the sets of the elements 5 and 2 will be merged. Since the depth of the set of 2 is larger than the depth of the set of 5, 2 is set as the parent of 5. Similarly, in the fifth step, when the sets of 1 and 7 are merged, 1 is set as the parent of 7. In the sixth and last step, since the depth of the set of 4 is larger, it is set as the parent of 0.

8.4. SOLVED EXERCISES



2. Given the index of an element, find the height of that element. For

CHAPTER 8. DISJOINT SET

example, the heights of the nodes h, g, a, d, f, c, b, e in Figure 8.1 are 3, 3, 2, 2, 2, 1, 1, 1 respectively.

```
int height(int index)
```

The function, that finds the height of an element given its index, is shown below. The height of an element is calculated by the number of nodes visited when we traverse the tree starting from that set and continue through the parent links until the top. Starting from the element with index `index` (Line 3), until we get to the top (Line 5), we go up using the parent links (Line 6). If the parent index of an element is equal to the index of that element, that element is at the top of the disjoint set tree.

```
int depth(int index){
    int i, d;
    i = index;
    d = 1;
    while (sets[i].parent != i){
        i = sets[i].parent;
        d++;
    }
    return d;
}
```

3. In this chapter, the sets are merged according to their depths. Another approach could be to merge sets according to their size. In this strategy, the disjoint set with smaller number of elements will be the subtree of the disjoint set with larger number of elements. Implement this idea in a new disjoint set data structure.

The modified definition of an element of a disjoint set is given below. Instead of the field **depth** showing the depth of an element, we add field **count** showing the number of elements in the subtree whose root is that element.

```

public class Set{
    int data;
    int parent;
    int count;
}
public class DisjointSet{
    Set sets [];
    int count;
}

```

When we merge the disjoint sets, not the depths of the disjoint sets, but the number of elements in the disjoint sets will be important. The function `union` that merges the disjoint sets according to this property is given below. First the indexes of the sets of those two elements are identified (Lines 3-4), then the set which has less number of elements will be the child of the set which has more number of elements (Lines 6, 9). According to this change, the number of elements are corrected (Lines 7, 10).

```

void union(int index1, int index2){
    int x, y;
    x = findSet(index1);
    y = findSet(index2);
    if (sets[x].count > sets[y].count){
        sets[y].parent = x;
        sets[x].count += sets[y].count;
    } else{
        sets[x].parent = y;
        sets[y].count += sets[x].count;
    }
}

```

8.5 Exercises

1. Write a function that computes the number of disjoint sets in a disjoint set structure.

```
int numberOfSets()
```

2. Given the index of a set, write a function that finds the depth of that

CHAPTER 8. DISJOINT SET

set. The depth of root node is 0, the depth of the node in the second level are 1, etc.

`int depth(int index)`

3. Given the index of a set, write a function that determines the number of children of that set.

`int numberOfChildren(int index)`

4. Given the index of a set, write a function that computes the sum of all ancestors (parent, grandparent, etc.).

`int sumOfAncestors(int index)`

5. Given the index of a set, write a function that returns the indexes of children of that set.

`LinkedList getChildren(int index)`

6. Write a function that returns the indexes of disjoint sets in a disjoint set structure.

`LinkedList getSets()`

8.6 Problems

1. Write a function that calculates the number of singleton disjoint sets in a disjoint set structure. A disjoint set is singleton, if the number of sets in that disjoint set is 1.

`int numberOfSingletons()`

2. Write a function that calculates the number of pair disjoint sets in a disjoint set structure. A disjoint set is pair, if the number of sets in that disjoint set is 2.

`int numberOfPairs()`

3. Given the index of a set S , write a method that unmerges (creates disjoint sets of) it. After unmerging, the direct children of S and S itself will be disjoint sets themselves. You don't need to modify the depths. Do not use any class or external methods.

```
void unmerge(int index)
```

4. Given the index of a set, write a method that returns the indexes of its grandchildren as a linked list. Do not use any class or external methods.

```
LinkedList grandChildren(int index)
```

5. Write a function that merges three sets given their indexes. You can use *findSet* method, but not the original *union* method. Merge the sets such that the resulting merged set will have the minimum depth. Update also the depth if needed.

```
void union(int index1, int index2, int index3)
```

6. You are given a set of equalities such as

```
0=9
1=2
3=5
5=7
9=4
5=4
6=8
```

where numbers correspond to variables. When the equalities are combined, we get

```
0=9=4=3=5=7
1=2
6=8
```

3 equalities. Write the function that finds the number of equalities when combined where N represents the number of variables, left and right represent the left and right parts of the equalities.

```
int combine(int N, int [] left, int [] right)
```

7. Write the method in **DisjointSet** class

```
int* getSetWithIndex(int index)
```

which returns the indexes of all sets in the disjoint set where a set with index *index* is in that set.

CHAPTER 8. DISJOINT SET

8. Write the method in **DisjointSet** class

```
void unionOfSets(int* indexList , int N)
```

that merges N sets given their indexes in the indexList. You should use findSet and the original unionOfSets method. Merge the sets such that the resulting merged set will have the minimum depth. Use an algorithm that sorts the sets according to their depths.

9. Write the method

```
int* numberOfDescendants(int index)
```

which returns the descendants (children, grandchildren, grandgrandchildren, etc.) of set with index *index*. Your method should run in $\mathcal{O}(N)$ time. The size of the returning array should be as much as needed.

10. Rewrite constructor in **DisjointSet** class

```
DisjointSet (int count)
```

such that all numbers with a common factor other than 1 will be in the same set. The numbers are from 0 to count - 1.

11. Given the index of a set, write a function that returns the ancestors (itself, parent, grandparent, etc.).

```
int* ascendants(int index)
```

The size of the returning array should be as much as needed.

12. Write a method

```
bool isValid ()
```

that returns true when the given disjoint set is valid, that is from every node n , when the ascendants are traversed, no circularity is observed (that is you do not encounter the node n again).

13. Write the method in **DisjointSet** class

```
int numberOfTriplets()
```

that calculates the number of triplet disjoint sets in a disjoint set structure. A disjoint set is a triplet, if the number of sets in that disjoint set is 3. Do not use any class or external methods.

8.6. PROBLEMS

A graph $G = (V, E)$ consists of one or more vertices and the lines (edges) connecting these vertices. The vertices in the graph are represented with the set V , and the edges are represented with the set E . A graph may not contain any edges but must contain at least one vertex.

Figure 9.1 shows an example directed graph. In this graph, there are six vertices numbered from 1 to 6 and there are seven directed edges. There are two edges from vertex numbered 1 to vertices numbered 2 and 4, one edge from vertex numbered 5 to vertex numbered 4, two edges from vertex numbered 3 to vertices numbered 5 and 6, one edge from vertex numbered 2 to vertex numbered 5, and one edge from vertex numbered 4 to vertex numbered 2.

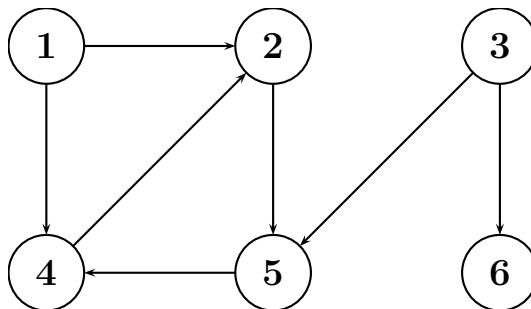


Figure 9.1: An example directed graph consisting of six vertices and seven edges

A graph can be undirected as well. In an undirected graph, the edges also connect vertices but in these type of graphs, since the edges are bidirectional, the directions are not shown. Figure 9.2 shows an example undirected graph. In this graph, there are five vertices numbered from 1 to 5 and there are four edges. There is one edge from vertex numbered 1 to vertex numbered 2, one edge from vertex numbered 2 to vertex numbered 5, one edge from vertex numbered 4 to vertex numbered 5 and one edge from vertex numbered 3 to vertex numbered 5.

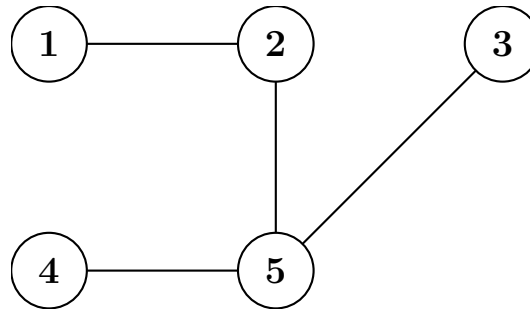


Figure 9.2: An example undirected graph consisting of five vertices and four edges

Another type of graph is the weighted graph. In a weighted graph, each edge, which connect two vertices, has a specific weight. This weight is defined by the relationship between the vertices it connects. Let say, we want to represent the cities and the highways between the cities in a country as a graph. Each city is represented as a vertex in a graph, each highway between two cities is represented as an edge, then the length of this highway is represented with the weight of this edge.

Figure 9.3 shows an example weighted graph. In this graph, there are five vertices numbered from 1 to 5 and there are five weighted edges. There is an edge from vertex numbered 4 to vertex numbered 1 of weight 7, an edge from vertex numbered 1 to vertex numbered 5 of weight 4, two edges from vertex numbered 5 to vertices numbered 4 and 2 of weights 1 and 2 respectively, and an edge from vertex numbered 2 to vertex numbered 3 of weight 3.

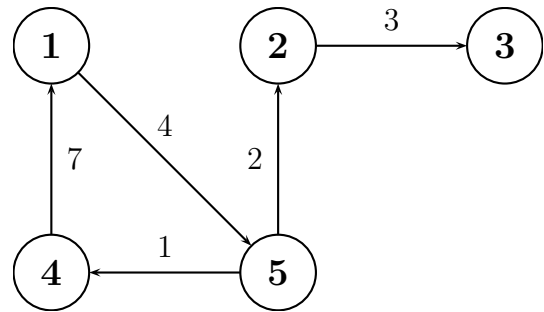


Figure 9.3: An example weighted graph consisting of five vertices and five edges

(a)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	0

(b)

	1	2	3	4	5
1	0	1	0	0	0
2	1	0	0	0	1
3	0	0	0	0	1
4	0	0	0	0	1
5	0	1	1	1	0

Figure 9.4: Adjacency matrix representation of the graphs shown in Figures 9.1 and 9.2

9.1 Definition

9.1.1 Adjacency Matrix Representation

There are two well known representations of graphs: Adjacency list representation and adjacency matrix representation. In the adjacency matrix

Table 9.1: Definition of a graph using adjacency matrix representation

<pre> class Graph{ private: int** edges; int vertexCount; public: Graph(int vertexCount); ~Graph(); } Graph::Graph(int vertexCount){ this.vertexCount = vertexCount; edges = new int*[vertexCount]; for (int i = 0; i < vertexCount; i++){ edges[i] = new int[vertexCount]; } for (i = 0; i < vertexCount; i++) for (j = 0; j < vertexCount; j++) edges[i][j] = 0; } Graph::~~Graph(){ for (int i = 0; i < vertexCount; i++){ delete [] edges[i]; } delete [] edges; } </pre>	<pre> public class Graph{ int [] edges; int vertexCount; public Graph(int vertexCount){ this.vertexCount = vertexCount; edges = new int[vertexCount][vertexCount]; for (i = 0; i < vertexCount; i++) for (j = 0; j < vertexCount; j++) edges[i][j] = 0; } } </pre>
---	---

representation, the graph G is represented via a matrix of $|N| \times |N|$ elements. Each element of the matrix, namely a_{ij} , takes the value 1, if there is an edge from vertex numbered i to vertex numbered j , otherwise takes value 0. The disadvantage of the adjacency matrix representation is, it requires N^2 cells of memory irrespective of the number of edges in the corresponding graph.

Figure 9.4(a) shows the adjacency matrix representation of the graph shown in Figure 9.1. Since there are two edges from vertex numbered 1 to the vertices numbered 2 and 4, the elements a_{01} and a_{03} of the adjacency matrix are 1. Similarly, since there is an edge from vertex numbered 5 to vertex numbered 4, the element a_{43} of the adjacency matrix is 1.

Figure 9.4(b) shows the adjacency matrix representation of the graph shown in Figure 9.2. Since the graph is undirected, the matrix is symmetric. For example, since there is an edge connecting the vertices numbered 2 and

5, both elements a_{14} and a_{41} of the adjacency matrix are 1. Similarly, since there is an edge connecting the vertices numbered 1 and 2, both elements a_{01} and a_{10} of the adjacency matrix are 1.

We use two dimensional array **edges** of size N to define adjacency matrix in a graph (Table 9.1). In order to allocate memory for the two dimensional array, we first allocate memory for the rows (first dimension). Later we allocate memory for the elements in each row (second dimension). Since there are no edges in the graph at the beginning, all elements of array **edges** are initialized to 0.

9.1.2 Adjacency List Representation

In the adjacency list representation, there is a linked list for each vertex. In the linked list of vertex v , there is a node for each edge outgoing from v . The data in that node is the index of the incoming vertex of that corresponding edge. If there are no outgoing edges from a vertex, its linked list is empty.

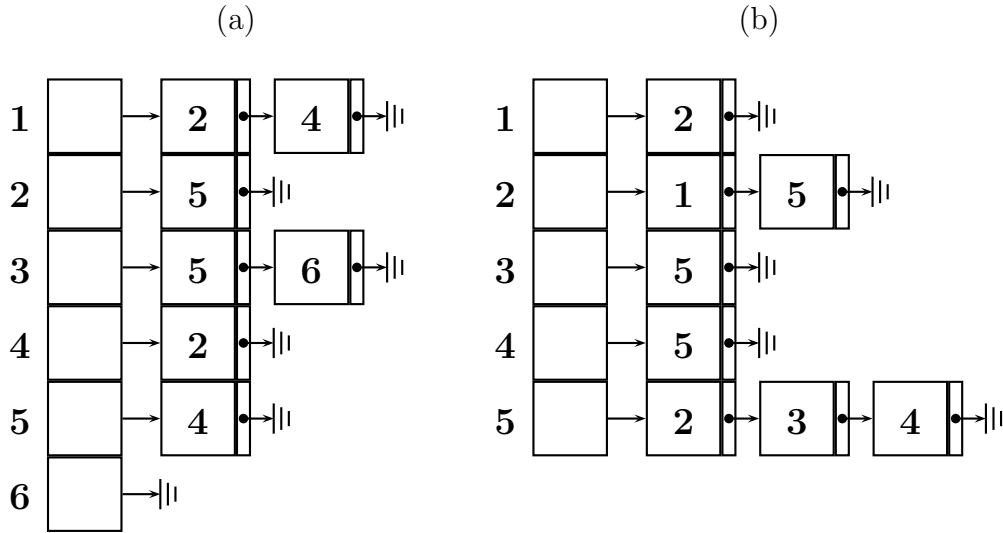


Figure 9.5: Adjacency list representation of the graphs shown in Figures 9.1 and 9.2

Figure 9.5(a) shows the adjacency list representation of the graph given in Figure 9.1. As can be seen, since the edges outgoing from vertex 1 are connected to the vertices 2 and 4, the linked list of 1 contains 2 and 4; since the edge outgoing from vertex 4 is connected to 2, the linked list of 4 contains

Table 9.2: Definition of edge for adjacency list representation

<pre> class Edge{ private: int from; int to; int weight; Edge* next; public: Edge(int from, int to, int weight); } Edge::Edge(int from, int to, int weight){ this->from = from; this->to = to; this->weight = weight; next = nullptr; } </pre>	<pre> public class Edge{ int from; int to; int weight; Edge next; public Edge(int from, int to, int weight){ this.from = from; this.to = to; this.weight = weight; this.next = null; } } </pre>
---	---

2; and since there are no edges outgoing from vertex 6, there are no nodes in the linked list of 6.

Figure 9.5(b) shows the adjacency list representation of the graph given in Figure 9.2. Since the graph is undirected, the edge connecting two vertices are enlisted in both linked lists. For example, since there is an endue between vertex 1 and vertex 2, there is 2 in the linked list of 1 and there is 1 in the linked list of 2. Similarly, since there is an edge between vertex 3 and vertex 5, there is 5 in the linked list of 3 and there is 3 in the linked list of 5.

Table 9.4 shows the adjacency list representation of a graph. Since there is a linked list for each vertex, there is an array of size N `edges` containing linked lists as elements. Therefore, `edges[0]` represents the linked list of vertex 1, `edges[1]` represents the linked list of vertex 2, etc.

9.2 Add Edge

In the previous section, we saw how we can define a graph using adjacency matrix or adjacency list representations. In both representations, there are no edges in the graph in the beginning. In this section, we will discuss how we can add a new edge into a graph defined previously.

CHAPTER 9. GRAPH

Table 9.3: Definition of edge list for adjacency list representation

<pre> class EdgeList{ private: Edge* head; Edge* tail; public: EdgeList(); void insert (Edge* newEdge); } EdgeList::EdgeList(){ head = nullptr; tail = nullptr; } EdgeList::insert (Edge* newEdge){ if (head == nullptr) { head = newEdge; } else tail ->setNext(newEdge); tail = newEdge; } </pre>	<pre> public class EdgeList{ Edge head; Edge tail; public EdgeList(){ head = null; tail = null; } public insert(Edge newEdge){ if (head == null) { head = newEdge; } else { tail.setNext(newEdge); } tail = newEdge; } } </pre>
--	---

Table 9.4: Definition of graph using adjacency list representation

<pre> class Graph{ private: EdgeList* edges; int vertexCount; public: Graph(int vertexCount); ~Graph(); } Graph::Graph(int vertexCount){ this.vertexCount = vertexCount; edges = new EdgeList[vertexCount]; for (i = 0; i < vertexCount; i++) edges[i] = EdgeList(); } Graph::~~Graph(){ delete [] edges; } </pre>	<pre> public class Graph{ EdgeList[] edges; int vertexCount; public Graph(int vertexCount){ this.vertexCount = vertexCount; edges = new EdgeList[vertexCount]; for (i = 0; i < vertexCount; i++) edges[i] = new EdgeList(); } } </pre>
---	---

Table 9.5: Adding an edge to a graph with adjacency matrix representation

```

1 void Graph::addEdge(int from, int to){
2     edges[from][to] = 1;
3 }
4 void Graph::addEdge(int from, int to, int weight){
5     edges[from][to] = weight;
6 }

```

```

1 void addEdge(int from, int to){
2     edges[from][to] = 1;
3 }
4 void addEdge(int from, int to, int weight){
5     edges[from][to] = weight;
6 }

```

Table 9.6: Adding an edge to a graph with adjacency list representation

```

1 void Graph::addEdge(int from, int to){
2     Edge* edge = new Edge(from, to, 1);
3     edges[from].insert (node);
4 }
5 void Graph::addEdge(int from, int to, int weight){
6     Edge* edge = new Edge(from, to, weight);
7     edges[from].insert (node);
8 }

```

```

1 void addEdge(int from, int to){
2     Edge edge = new Edge(from, to, 1);
3     edges[from].insert (node);
4 }
5 void addEdge(int from, int to, int weight){
6     Edge edge = new Edge(from, to, weight);
7     edges[from].insert (node);
8 }

```

In order to add an edge into a graph, we need to know (i) the starting vertex, (ii) the ending vertex and if exists (iii) the weight of the edge. According to these information, Table 9.5 shows the algorithm that adds a new edge to a graph with adjacency matrix representation. In the adjacency ma-

trix representation, if there is an edge from vertex i to vertex j , the element with index (i, j) of the array `edges` will have the value 1 (or if the graph is a weighted graph then the weight of that edge), otherwise 0. While the parameter `from` representing the starting vertex, the parameter `to` representing the ending vertex and the parameter `weight` representing the weight of the edge, the function converts the value of the element in the position `(from,to)` of the matrix `edges` from 0 to `weight` (Line 2).

On the other hand, Table 9.6 shows the algorithm that adds an edge to a graph with adjacency list representation. In the adjacency list representation, if there is an edge from vertex i to vertex j , we will add a new element j to the linked list of vertex i (`edges[i]`). While the parameter `from` representing the starting vertex and `to` representing the ending vertex, the function first encapsulates the value `to` to a node (Line 3), then adds this new node to the linked list with index `from` (Line 4).

9.3 Application: Connected Components

If the vertices in a graph G can be divided into K disjoint sets, all vertices in each disjoint set are connected to each other directly or indirectly and any two vertices in two separate disjoint sets are not connected to each other, then G is said to have K connected components. Each connected component is also called a subgraph.

Figure 9.6 shows a graph consisting of 7 vertices (countries). In this graph, there are two connected components. In the first connected component corresponding vertices are Turkey, Greece, Bulgaria, and Albania, in the second connected component corresponding vertices are U.S.A., Canada, and Mexico. It is possible to reach from vertices Turkey, Greece, Bulgaria, and Albania to each other, whereas we can not reach from any of these vertices to the vertices U.S.A., Canada, or Mexico. Similarly, it is possible to reach from vertices U.S.A., Canada, and Mexico to each other, on the other hand we can not reach from any of these vertices to the vertices Turkey, Greece, Bulgaria, or Albania.

We can use three different approaches (disjoint set, depth first search, breadth first search) to find the connected components in a given graph G .

9.3. APPLICATION: CONNECTED COMPONENTS

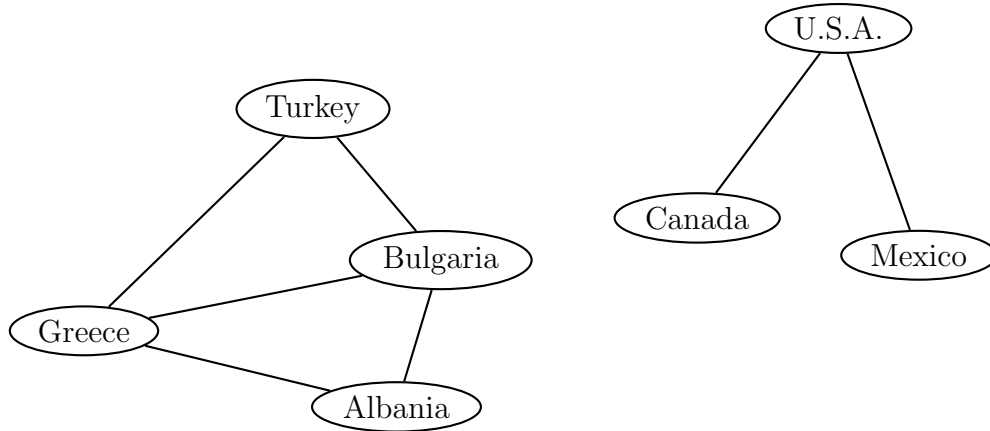


Figure 9.6: An example seven country map

9.3.1 Disjoint Set Approach

Table 9.7 shows the disjoint set approach to determine the connected components in a graph G . In the algorithm, N represents the number of vertices in the graph, $e[v_1, v_2]$ represents the edge connecting the vertices v_1 and v_2 . First, we define a disjoint set for each vertex in the graph (Line 4). If there are no edges in the graph, in that case, each vertex will be a connected component. If there is at least one edge, for each vertex v_1 we traverse all edges outgoing from that vertex one by one (Lines 5-7, 11). For each edge e that connects vertices v_1 and v_2 , the disjoint set containing the vertex v_1 is merged with the disjoint set containing the vertex v_2 (Line 10). If the vertices belong to the same disjoint set, there is nothing to do (Line 9). When the algorithm is finished, the number of disjoint sets will give us the number of connected components in the graph.

Figure 9.7 shows the extraction of the connected components for the graph given in Figure 9.6 using the disjoint set approach. In the first phase of the algorithm, each country alone forms up a disjoint set (Figure 9.7 (a)). For the first vertex (U.S.A.), the disjoint set of the vertex U.S.A. is merged with the disjoint sets of the vertices, which are directly connected with an edge to the vertex U.S.A. (First U.S.A. is merged with Canada then U.S.A.

CHAPTER 9. GRAPH

Table 9.7: Disjoint set algorithm that finds connected components in a graph

```
1 void connectedComponentsDisjointSet(){
2     Edge* edge;
3     int toNode;
4     DisjointSet sets = DisjointSet(vertexCount);
5     for (int fromNode = 0; fromNode < vertexCount; fromNode++){
6         edge = edges[fromNode].getHead();
7         while (edge != nullptr){
8             toNode = edge->getTo();
9             if (sets.findSetRecursive(fromNode) != sets.findSetRecursive(toNode)){
10                 sets.unionOfSets(fromNode, toNode);
11             }
12             edge = edge->getNext();
13         }
14     }
15 }
```

```
1 void connectedComponentsDisjointSet(){
2     Edge edge;
3     int toNode;
4     DisjointSet sets = new DisjointSet(vertexCount);
5     for (int fromNode = 0; fromNode < vertexCount; fromNode++){
6         edge = edges[fromNode].getHead();
7         while (edge != null){
8             toNode = edge.getTo();
9             if (sets.findSetRecursive(fromNode) != sets.findSetRecursive(toNode)){
10                 sets.unionOfSets(fromNode, toNode);
11             }
12             edge = edge.getNext();
13         }
14     }
15 }
```

with Mexico). The disjoint set of the second vertex Albania is merged with the disjoint sets of Bulgaria and Greece, which are connected with an edge to Albania (Figures 9.7 (d) and (e)). As a last step, the disjoint set of the third vertex Bulgaria is merged with the disjoint set of the Albania (Figure 9.7 (f)).

9.3. APPLICATION: CONNECTED COMPONENTS

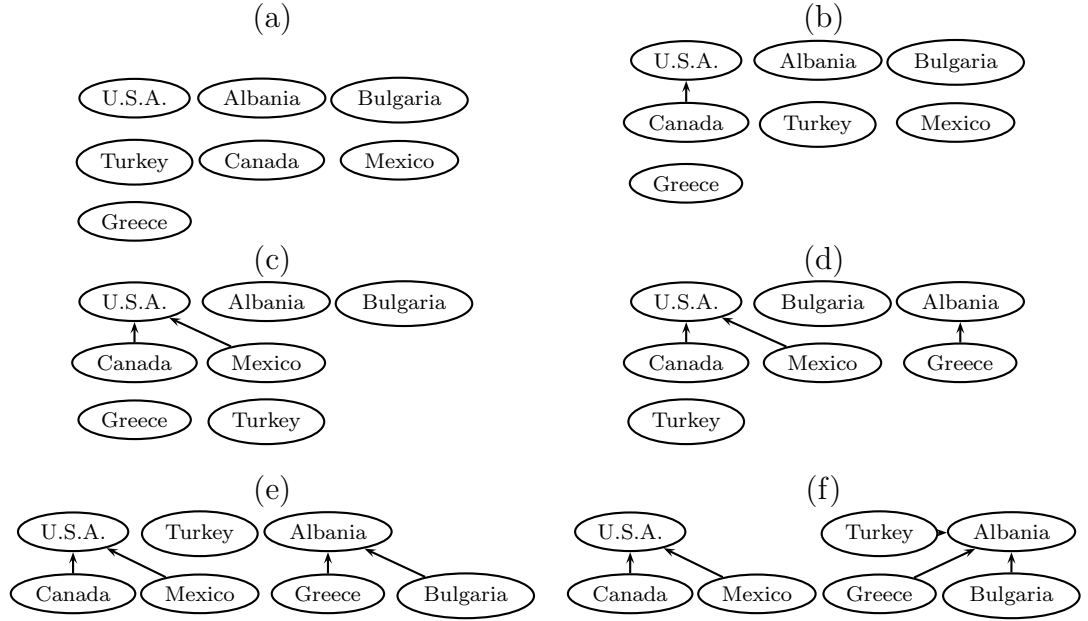


Figure 9.7: Finding the connected components using the disjoint set approach in the graph given in Figure 9.6

9.3.2 Depth First Search Approach

Another algorithm that find the connected components in graph G is the depth first search. In this method, while we are traversing the graph with the depth first search, we are extracting the connected components. In the depth first search approach, we traverse one vertex starting from the initial vertex. Before continuing the traversal with the adjacent vertices of the initial vertex, we continue the traversal recursively by taking the recent visited vertex as the initial vertex. The depth stems from the fact that, in the graph we proceed in depth from the first vertex to the second vertex, then from that vertex to the third vertex, from the third vertex to the fourth vertex, etc.

Table 9.9 shows the algorithm that determines the connected components in a given graph using the depth first search approach. The **visited** array represents if each vertex is currently visited or not. In this array, there is an element corresponding to each vertex in the graph. If for a given vertex i , **visited**[i] is 1, this will mean we have already reached this vertex i using the

CHAPTER 9. GRAPH

depth first approach, if it is 0, this will mean we haven't reached it yet. First, the function `connectedComponentsDfs` initializes the values of all vertices in the visited array to 0 (establishing that those vertices are not reached yet) (Lines 3-4), then for each vertex in turn, if that vertex is not visited yet (Line 6), we call the function `depthFirstSearch` (Line 8).

The function `depthFirstSearch` is a recursive function. What it does is, starting from a vertex x , it visits all vertices which are in the same connected component of that vertex x . First, for all vertices y adjacent to the vertex x (Lines 16-18, 23), if the vertex y is not visited yet (Lines 19), it visits y (Line 20) and calls itself with the vertex y again (Line 21). When the function `depthFirstSearch` updated all values of the vertices, which are reachable from the vertex x , in the visited array (Line 20), for those vertices, the function `depthFirstSearch` will not be called by the main function again. So, each call of the `depthFirstSearch` from the main function will correspond to finding a new connected component (Line 9).

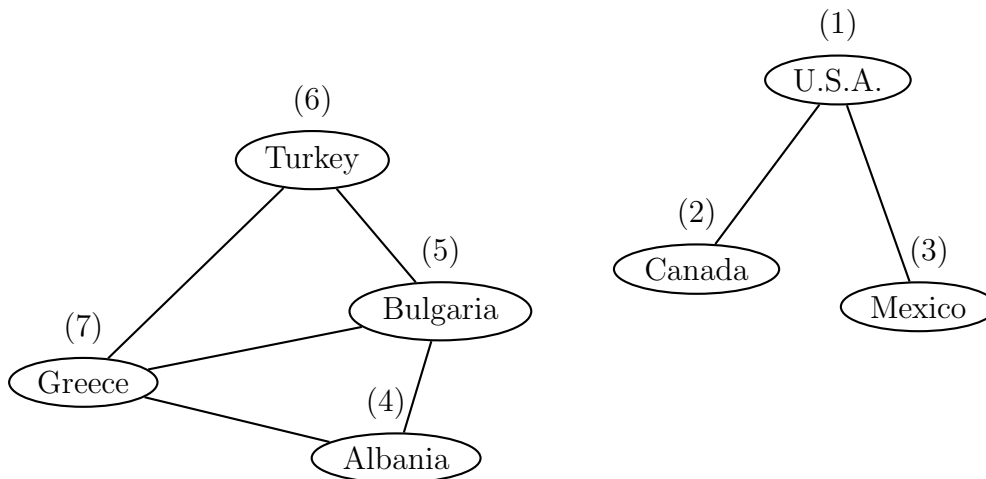


Figure 9.8: Finding the connected components in the graph in Figure 9.6 using the depth first search approach

Figure 9.8 shows the extraction of the connected components from the graph given in Figure 9.6. The number above a vertices shows the order of visit of that vertex using the depth first search.

9.3. APPLICATION: CONNECTED COMPONENTS

Table 9.8: Depth first search algorithm that finds the connected components in a given graph (C++)

```
int connectedComponentsDfs(){
    int component = 0;
    bool* visited = new bool[vertexCount];
    for (int vertex = 0; vertex < vertexCount; vertex++){
        if (!visited [vertex]){
            visited [vertex] = true;
            depthFirstSearch( visited , vertex );
            component++;
        }
    }
    return component;
}

void depthFirstSearch(bool* visited , int fromNode){
    Edge* edge;
    int toNode;
    edge = edges[fromNode].getHead();
    while (edge != nullptr){
        toNode = edge->getTo();
        if (!visited [toNode]){
            visited [toNode] = true;
            depthFirstSearch( visited , toNode);
        }
        edge = edge->getNext();
    }
}
```

- (1) First we visit the vertex U.S.A then we call the function **depthFirstSearch** with this vertex.
- (2) We visit vertex Canada as the first adjacent vertex of U.S.A. Then we call the function **depthFirstSearch** with vertex Canada but since all adjacent vertices of Canada are visited, this function will end.
- (3) We visit vertex Mexico as the second adjacent vertex of U.S.A. Then we call the function **depthFirstSearch** with vertex Mexico but since all adjacent vertices of Mexico are visited, this function will end. In this stage, the function **depthFirstSearch** called with U.S.A. will end and the number of connected components will be incremented by one.

CHAPTER 9. GRAPH

Table 9.9: Depth first search algorithm that finds the connected components in a given graph (Java)

```
int connectedComponentsDfs(){
    int component = 0;
    boolean[] visited = new boolean[vertexCount];
    for (int vertex = 0; vertex < vertexCount; vertex++){
        if (! visited [vertex]){
            visited [vertex] = true;
            depthFirstSearch(visited, vertex);
            component++;
        }
    }
    return component;
}

void depthFirstSearch(boolean[] visited, int fromNode){
    Edge edge;
    int toNode;
    edge = edges[fromNode].getHead();
    while (edge != null){
        toNode = edge.getTo();
        if (! visited [toNode]){
            visited [toNode] = true;
            depthFirstSearch(visited, toNode);
        }
        edge = edge.getNext();
    }
}
```

- (4) Since we haven't reached Albania yet, in the function `connectedComponentsDfs`, Albania is visited and then we call the function `depthFirstSearch` with this vertex.
- (5) We visit vertex Bulgaria as the first adjacent vertex of Albania. Then we call the function `depthFirstSearch` with vertex Bulgaria.
- (6) The first of the adjacent vertices of Bulgaria, Albania is already visited. The first unvisited vertex Turkey is now visited at this stage and we call the function `depthFirstSearch` with vertex Turkey.
- (7) The first of the adjacent vertices of Turkey, Bulgaria is already visited. The first unvisited vertex Greece is now visited at this stage and we call

9.3. APPLICATION: CONNECTED COMPONENTS

the function `depthFirstSearch` with vertex Greece. But since all adjacent vertices of Greece are visited, this function will end. One after another the recursive functions belonging to Turkey, Bulgaria and Albania will end and the number of connected components will be incremented by one.

9.3.3 Breadth First Search Approach

Yet another algorithm that finds the connected components in a graph G is the breadth first approach. In this approach, while we are traversing the graph with the breadth first search, we are extracting the connected components. Contrary to the depth first search, in the breadth first search, we visit first the vertices whose distance to the initial vertex is 1, then the vertices with distance 2, then the vertices with distance 3, etc. For this reason, this search is called breadth first search.

Table 9.11 shows the algorithm that finds the connected components in a given graph using the breadth first search approach. The `visited` array represents if each vertex is currently visited or not. In this array, there is an element corresponding to each vertex in the graph. If for a given vertex i , `visited[i]` is 1, this will mean we have already reached this vertex i using the breadth first approach, if it is 0, this will mean we haven't reached it yet. First, the function `connectedComponentsBfs` initializes the values of all vertices in the `visited` array to 0 (establishing that those vertices are not reached yet) (Lines 3-4), then for each vertex in turn, if that vertex is not visited yet (Line 6), we call the function `breadthFirstSearch` (Line 8).

`breadthFirstSearch` function will reach the vertices whose distance to vertex x is 1, then the vertices whose distance is 2, then the vertices whose distance is 3, etc. First vertex x is added to the queue k (Lines 18-20) and since it is the single vertex in the queue, it is dequeued immediately (Lines 22-23). Then the adjacent vertices of x , which are not visited yet (Lines 24-26, 33), are added to the queue k (Lines 28-31). These vertices form up the vertex set which are one edge distant from the vertex x . Then the adjacent vertices of x are dequeued in turn (Lines 22-23), then the adjacent (Lines 24-26, 33) and not visited vertices of those adjacent vertices are added to the queue k (Lines 28-31). These vertices form up the vertex set which are two edges distant from the vertex x . The function continues until there are no vertices left in the queue (Line 21).

Figure 9.9 shows the extraction of connected components from the graph

CHAPTER 9. GRAPH

Table 9.10: Breadth first search algorithm that finds the connected components in a given graph (C++)

```
int connectedComponentsBfs(){
    int component = 0;
    bool* visited = new bool[vertexCount];
    for (int vertex = 0; vertex < vertexCount; vertex++){
        if (!visited [vertex]){
            visited [vertex] = true;
            breadthFirstSearch(visited, vertex);
            component++;
        }
    }
    return component;
}

void breadthFirstSearch(boolean[] visited, int fromNode){
    Edge* edge;
    int fromNode, toNode;
    Queue queue = new Queue();
    queue.enqueue(new Node(startNode));
    while (!queue.isEmpty()){
        fromNode = queue.dequeue()->getData();
        edge = edges[fromNode].getHead();
        while (edge != nullptr) {
            toNode = edge->getTo();
            if (!visited [toNode]){
                visited [toNode] = true;
                queue.enqueue(new Node(toNode));
            }
            edge = edge->getNext();
        }
    }
}
```

given in Figure 9.6. The number above a vertices shows the order of visit of that vertex using the breadth first search.

- (1) First we visit the vertex U.S.A then we call the function **breadthFirstSearch** with this vertex.
- (2) The vertices Canada and Mexico, which are adjacent to the vertex U.S.A., are added to the queue.

9.3. APPLICATION: CONNECTED COMPONENTS

Table 9.11: Breadth first search algorithm that finds the connected components in a given graph (Java)

```
int connectedComponentsBfs(){
    int component = 0;
    boolean[] visited = new boolean[vertexCount];
    for (int vertex = 0; vertex < vertexCount; vertex++){
        if (!visited [vertex]){
            visited [vertex] = true;
            breadthFirstSearch(visited, vertex);
            component++;
        }
    }
    return component;
}

void breadthFirstSearch(boolean[] visited, int fromNode){
    Edge edge;
    int fromNode, toNode;
    Queue queue = new Queue();
    queue.enqueue(new Node(startNode));
    while (!queue.isEmpty()){
        fromNode = queue.dequeue().getData();
        edge = edges[fromNode].getHead();
        while (edge != null) {
            toNode = edge.getTo();
            if (!visited [toNode]){
                visited [toNode] = true;
                queue.enqueue(new Node(toNode));
            }
            edge = edge.getNext();
        }
    }
}
```

- (3) Vertex Canada is removed from the queue and the vertices that are adjacent to it are tried but since all adjacent vertices of Canada are visited, there will be no change in the queue.
- (4) Vertex Mexico is removed from the queue and the vertices that are adjacent to it are tried but since all adjacent vertices of Mexico are visited, there will be no change in the queue. Since there are no elements left in the queue, **breadthFirstSearch** function is ended and the number

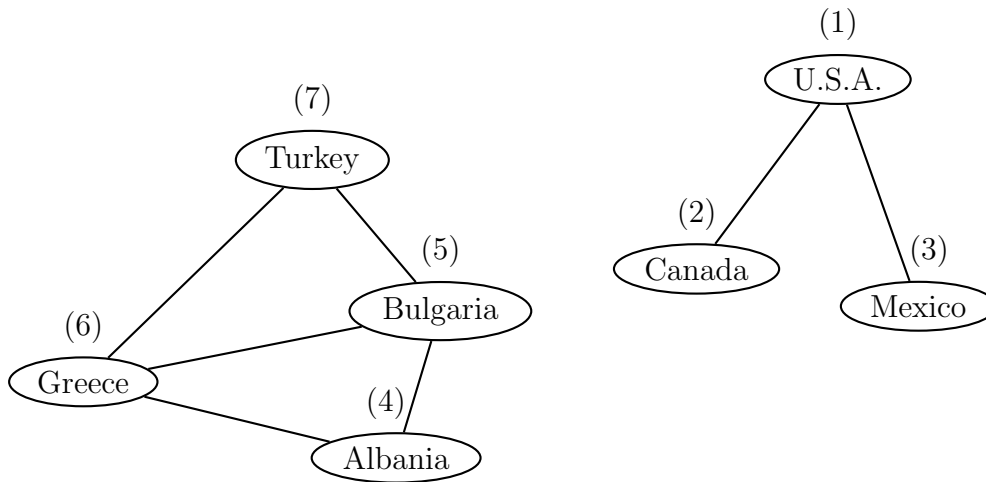


Figure 9.9: Finding the connected components in the graph in Figure 9.6 using the breadth first search approach

of connected components will be incremented by one.

- (5) Since we haven't reached Albania yet, in the function `connectedComponentsBfs`, Albania is visited and then we call the function `breadthFirstSearch` with this vertex.
- (6) The vertices Bulgaria and Greece, which are adjacent to the vertex Albania, are added to the queue.
- (7) Vertex Bulgaria is removed from the queue and vertex Turkey, which is adjacent to it, is added to the queue.
- (8) Vertex Greece is removed from the queue and the vertices that are adjacent to it are tried but since all adjacent vertices of Greece are visited, there will be no change in the queue.
- (9) Vertex Turkey is removed from the queue and the vertices that are adjacent to it are tried but since all adjacent vertices of Turkey are visited, there will be no change in the queue. Since there are no elements left in the queue, `breadthFirstSearch` function is ended and the number of connected components will be incremented by one.

9.4 Application: Shortest Path

The shortest path problem can be summarized as finding the shortest path from a given vertex u to a given vertex v in a given graph G . Graph $G(V, E)$ is composed of the vertices (set V), and the edges connecting these vertices (set E). The length of the path from vertex u to vertex v is the sum of the weights of the edges in that path. If one can not reach from vertex u to vertex v , the result of the shortest path problem is ∞ . Let's examine the shortest path problem in two main topics as, finding the shortest paths of all paths from vertex u to all other vertices and finding the shortest paths from all vertices to all vertices.

9.4.1 Bellman-Ford Algorithm

Bellman-Ford algorithm (Table 9.12) finds the length of the shortest paths from an initial vertex to all other vertices in a graph G . The algorithm returns two arrays as output. In the first array (array d) there is one number $d[i]$ corresponding to each vertex i . This number represents the length of the shortest path of that vertex to the initial vertex. In the second array (array **prev**) there is one vertex **once**[i] corresponding to each vertex i . **prev**[i] is the last vertex we visit if we want to go to vertex i with the shortest path. Using the array **prev**, one can determine the shortest path to a given vertex i . What we should do is, equalize vertex i to the value **prev**[i] until **prev**[i] is the initial vertex. If we store the vertex we visit at each stage, we will determine the shortest path.

The first for in the algorithm initializes these two arrays (Lines 5-8). Each element of the array **d** except the initial vertex is initialized to value ∞ (Line 6), each element of the array **prev** is initialized to value -1 (Line 7). As we have mentioned before, if we can not reach to a point from the initial vertex, with this initialization the length of that vertex to the initial vertex will be ∞ .

The main part of the algorithm is where two nested loops are working together (Line 10-17). In this part, the distance array is tried to be updated. Let the length of the shortest path to a vertex u be represented by $d[u]$. If the current shortest path of vertex v is larger (Line 13) than the sum of the length of the edge containing the vertices u, v and the length of the shortest

CHAPTER 9. GRAPH

Table 9.12: Bellman-Ford algorithm that finds the shortest paths from an initial vertex to all other vertices in a graph

```
Path* bellmanFord(int source){
    Path* paths = new Path[vertexCount];
    for (int i = 0; i < vertexCount; i++){
        paths[i] = Path(INT_MAX, -1);
    }
    paths[source].setDistance(0);
    for (int i = 0; i < vertexCount - 1; i++){
        for (int from = 0; from < vertexCount; from++){
            for (int to = 0; to < vertexCount; to++){
                int newDistance = paths[from].getDistance() + edges[from][to];
                if (newDistance < paths[to].getDistance()){
                    paths[to].setDistance(newDistance);
                    paths[to].setPrevious(from);
                }
            }
        }
    }
    return paths;
}
```

```
Path[] bellmanFord(int source){
    Path[] paths = new Path[vertexCount];
    for (int i = 0; i < vertexCount; i++){
        paths[i] = new Path(Integer.MAX_VALUE, -1);
    }
    paths[source].setDistance(0);
    for (int i = 0; i < vertexCount - 1; i++){
        for (int from = 0; from < vertexCount; from++){
            for (int to = 0; to < vertexCount; to++){
                int newDistance = paths[from].getDistance() + edges[from][to];
                if (newDistance < paths[to].getDistance()){
                    paths[to].setDistance(newDistance);
                    paths[to].setPrevious(from);
                }
            }
        }
    }
    return shortestPaths;
}
```

9.4. APPLICATION: SHORTEST PATH

path of the vertex u (Line 13), then this will mean there is another shorter path to v passing through vertex u . Therefore, the **prev** vertex of the vertex v will be u (Line 15).

Figure 9.10 shows a graph consisting of 5 vertices (t, u, v, y, z). The aim is to find the length of the shortest paths from the initial vertex u to other 4 vertices.

- (I) First stage of the graph is given. The shortest path length of the initial vertex is set to 0, path lengths of other vertices are ∞ .
- (II) The edges represented by red color (3, 5, 11) are used in determining the new shortest path lengths.

$$\begin{aligned}d[u] + 3 = 3 &< d[t] = \infty \\d[u] + 5 = 5 &< d[v] = \infty \\d[u] + 11 = 11 &< d[z] = \infty\end{aligned}$$

- (III) Looking from vertex t , the edges of lengths 6 and 1 determine the new shortest path lengths.

$$\begin{aligned}d[t] + 1 = 4 &< d[v] = 5 \\d[t] + 6 = 9 &< d[y] = \infty\end{aligned}$$

- (IV) Looking from vertex v , the edges of lengths 4 and 6 determine the new shortest path lengths.

$$\begin{aligned}d[v] + 4 = 8 &< d[y] = 9 \\d[v] + 6 = 10 &< d[z] = 11\end{aligned}$$

- (V) Looking from vertex y , the edge of length 1 determines the shortest path length of z .

$$d[y] + 1 = 9 < d[z] = 10$$

9.4.2 Dijkstra Algorithm

Similar to the Bellman-Ford algorithm, the Dijkstra algorithm also finds the shortest paths from an initial vertex to all other vertices. For this, the length

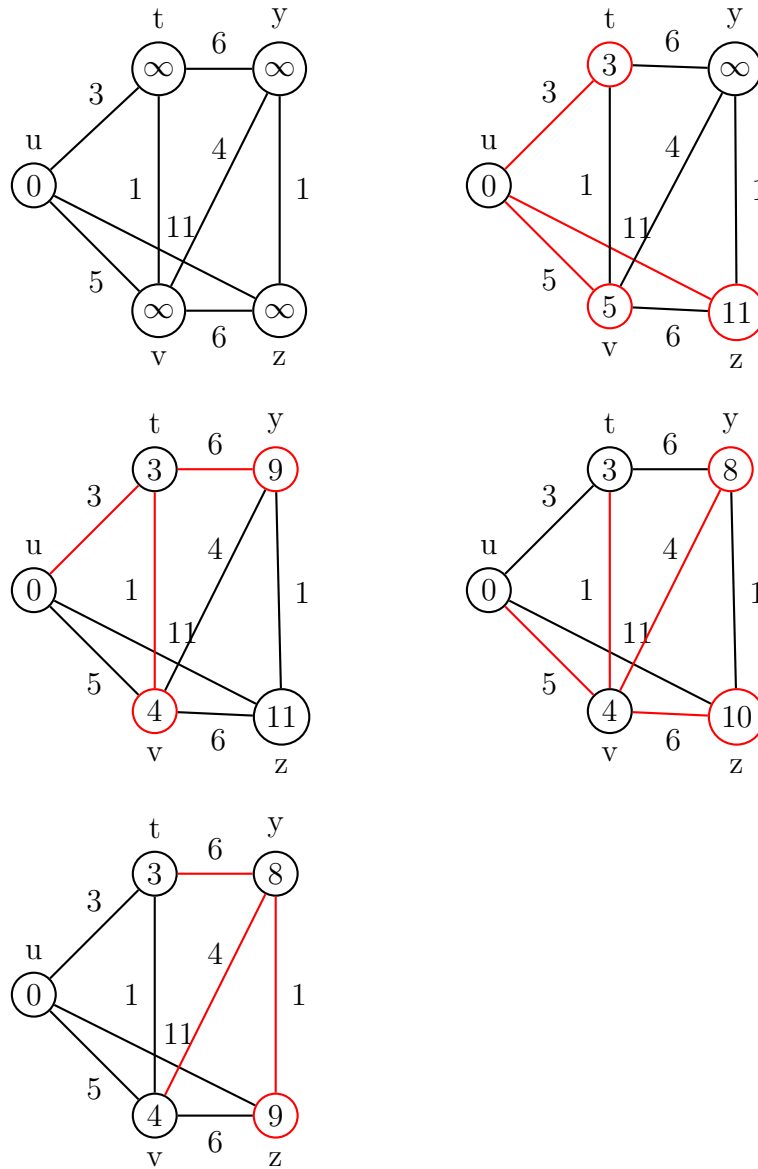


Figure 9.10: Application of Bellman-Ford algorithm on a graph of 5 vertices

of the current shortest path from each vertex v_i to the initial vertex is stored in a min-heap. Table 9.14 shows the code of the Dijkstra algorithm. In the

9.4. APPLICATION: SHORTEST PATH

first stage, similar to the Bellman-Ford algorithm, the path lengths are set to their initial values (Lines 7) and all these values are added to the heap T (Lines 11-13).

Table 9.13: Dijkstra algorithm that finds the shortest paths from an initial vertex to all other vertices in a graph (C++)

```
Path* dijkstra(int source){
    Path* paths = new Path[vertexCount];
    for (int i = 0; i < vertexCount; i++){
        paths[i] = Path(INT_MAX, -1);
    }
    paths[source].setDistance(0);
    Heap heap = Heap(vertexCount);
    for (int i = 0; i < vertexCount; i++){
        heap.insert(HeapNode(shortestPaths[i].getDistance(), i));
    }
    while (!heap.isEmpty()){
        HeapNode node = heap.deleteMin();
        int fromNode = node.getName();
        for (int toNode = 0; toNode < vertexCount; toNode++){
            int newDistance = paths[fromNode].getDistance() + edges[fromNode][toNode];
            if (newDistance < paths[toNode].getDistance()){
                int position = heap.search(toNode);
                heap.update(position, newDistance);
                paths[toNode].setDistance(newDistance);
                paths[toNode].setPrevious(fromNode);
            }
        }
    }
    return shortestPaths;
}
```

At this stage, the algorithm chooses the minimum element of the heap v_i (the nearest vertex to the initial vertex in the graph). This selection is realized by the function `deleteMin` (Line 15). If we can reach the adjacent vertices v_j of the vertex v_i using the vertex v_i in a shorter way (Line 18), we update the shortest path lengths of vertices v_j in the min-heap (Line 19). The for loop inside the while loop, does this control and update operation for all adjacent vertices v_j to vertex v_i (Line 16).

Figure 9.11 shows the application of the Dijkstra algorithm on an example graph of 5 vertices. The stages of the algorithm are

CHAPTER 9. GRAPH

Table 9.14: Dijkstra algorithm that finds the shortest paths from an initial vertex to all other vertices in a graph (Java)

```

Path[] dijkstra(int source){
    Path[] paths = new Path[vertexCount];
    for (int i = 0; i < vertexCount; i++){
        paths[i] = new Path(Integer.MAX_VALUE, -1);
    }
    paths[source].setDistance(0);
    Heap heap = new Heap(vertexCount);
    for (int i = 0; i < vertexCount; i++){
        heap.insert(new HeapNode(shortestPaths[i].getDistance(), i));
    }
    while (!heap.isEmpty()){
        HeapNode node = heap.deleteMin();
        int fromNode = node.getName();
        for (int toNode = 0; toNode < vertexCount; toNode++){
            int newDistance = paths[fromNode].getDistance() + edges[fromNode][toNode];
            if (newDistance < paths[toNode].getDistance()){
                int position = heap.search(toNode);
                heap.update(position, newDistance);
                paths[toNode].setDistance(newDistance);
                paths[toNode].setPrevious(fromNode);
            }
        }
    }
    return shortestPaths;
}

```

- (I) The elements of the min-heap T are $\{u, t, v, y, z\}$. The shortest path lengths are $\{0, \infty, \infty, \infty, \infty\}$ respectively.
- (II) Since the minimum element of the heap is u , it is selected. The adjacent vertices of u are the vertices t, v , and z . Since the shortest path lengths of all these vertices are ∞ , they will be updated. The elements of the new heap are $\{t, v, z, y\}$ and their shortest path lengths are $\{3, 5, 11, \infty\}$ respectively.
- (III) Since the minimum element is t , in this stage vertex t is selected. We can reach vertices u, v , and y from vertex t . Since $d[u] = 0 < d[t] + d[ut] = 3 + 3$, the shortest path length of vertex u will not be changed

9.4. APPLICATION: SHORTEST PATH

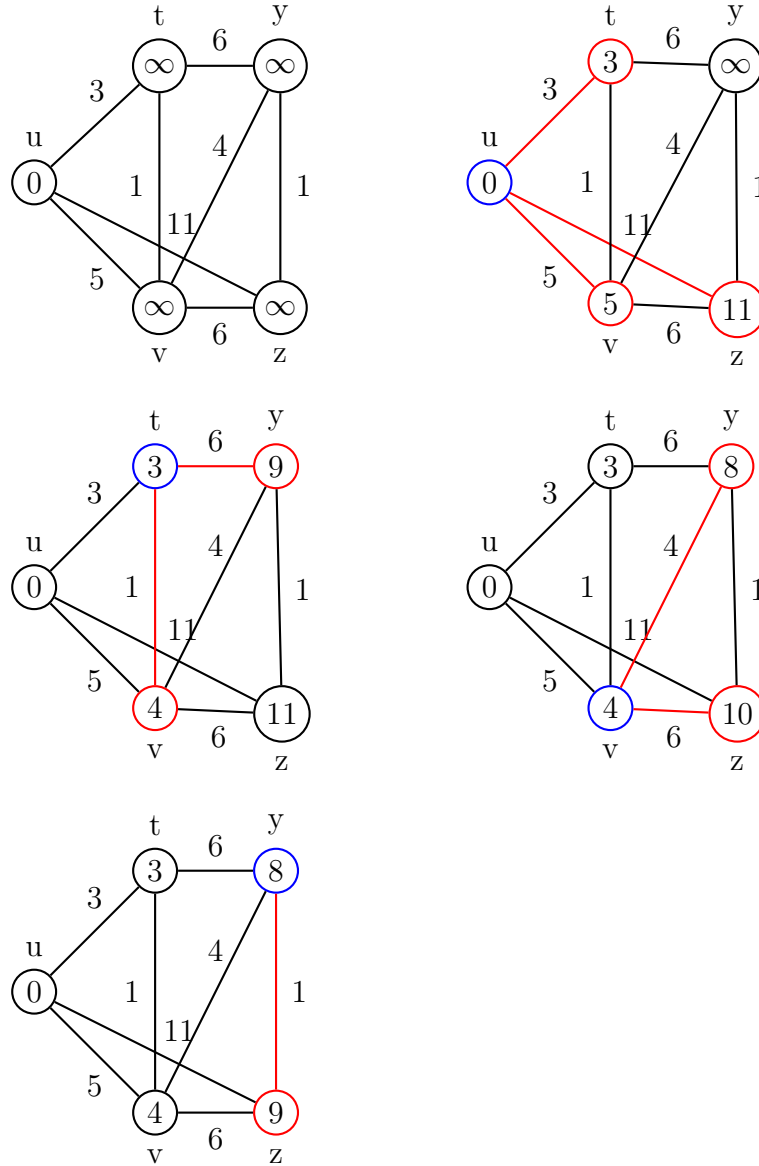


Figure 9.11: Application of Dijkstra algorithm on a graph of 5 vertices

but on the other hand since $d[v] = 5 > d[t] + d[vt] = 3 + 1$, $d[y] = \infty > d[t] + d[vy] = 3 + 6$, the shortest path lengths of vertices v and y will be

CHAPTER 9. GRAPH

changed. The elements of the new heap are $\{v, y, z\}$ and their shortest path lengths are $\{4, 9, 11\}$.

- (IV) The vertices that are reachable from vertex v are y, z, t , and u . From these vertices, only the shortest path lengths of y and z will be updated ($d[y] = 9 > d[v] + d[v y] = 4 + 4$, $d[z] = 11 > d[v] + d[v z] = 4 + 6$). The elements of the new heap are $\{y, z\}$ and their shortest path lengths are $\{8, 10\}$ respectively.
- (V) At the last step, vertex y is selected from the heap. There is a shorter path to z passing through y ($d[z] = 10 > d[y] + d[y z] = 8 + 1$).

9.4.3 Floyd-Warshall Algorithm

The algorithms Bellman-Ford and Dijkstra we examined in the previous sections were finding the shortest paths from one vertex to all other vertices. In some applications, we are not only required to find the shortest paths from one vertex but from all vertices to all vertices. In these problems, we can either run for each vertex Bellman-Ford or Dijkstra algorithms or use Floyd Warshall algorithm in general.

Floyd-Warshall algorithm finds all shortest paths from all vertices which use only one intermediate vertex, then finds all shortest paths which use two intermediate vertices, ..., finds all shortest paths which use $V - 1$ intermediate vertices. The logic behind Floyd-Warshall algorithm is Dynamic Programming . If we have found the shortest paths using k intermediate vertices, we can determine the shortest paths using $k + 1$ intermediate vertices by adding one more vertex. Since the shortest paths in a graph can not use one vertex more than once, the shortest path which uses the maximum number of vertices will use $V - 1$ vertices.

Table 9.15 shows the Floyd-Warshall algorithm. Here the external for loop corresponds to the number of intermediate vertices used. If we express in another way, when the external loop takes the value $k = 1$ (Line 9), we will find the shortest paths from all vertices to all vertices using the first l vertices. The two for loops on the other hand (Lines 10-11) checks if we can find a shorter path from vertex v_i to vertex v_j using the intermediate vertex v_k (Lines 12-13). If there is no shorter path, the shortest path will remain the same, but if exists the shortest path will be replaced with the new shorter path (Line 14). Note that, the transition from the matrix d^{k-1} to the matrix

9.4. APPLICATION: SHORTEST PATH

Table 9.15: Floyd-Warshall algorithm that finds the shortest paths from all vertices to all vertices in a given graph

```

int** floydWarshall(){
    int** distances = new int*[vertexCount];
    for (int i = 0; i < vertexCount; i++)
        distances[i] = new int[vertexCount];
        for (int j = 0; j < vertexCount; j++){
            distances[i][j] = edges[i][j];
        }
        for (int k = 0; k < vertexCount; k++)
            for (int i = 0; i < vertexCount; i++)
                for (int j = 0; j < vertexCount; j++)
                    int newDistance = distances[i][k] + distances[k][j];
                    if (newDistance < distances[i][j]){
                        distances[i][j] = newDistance;
                    }
            }
        }
    return distances;
}

```

```

int [][] floydWarshall(){
    int [][] distances = new int[vertexCount][vertexCount];
    for (int i = 0; i < vertexCount; i++){
        for (int j = 0; j < vertexCount; j++){
            distances[i][j] = edges[i][j];
        }
        for (int k = 0; k < vertexCount; k++){
            for (int i = 0; i < vertexCount; i++){
                for (int j = 0; j < vertexCount; j++){
                    int newDistance = distances[i][k] + distances[k][j];
                    if (newDistance < distances[i][j]){
                        distances[i][j] = newDistance;
                    }
                }
            }
        }
    }
    return distances;
}

```

d^k is the same as in dynamic programming. The operation is like in dynamic programming, filling a table of values with the previous table of values.

Figure 9.12 shows the application of the Floyd-Warshall algorithm. The example graph is given in the upper-left corner of the figure. Next to the original graph, the initial version of the D matrix in Floyd-Warshall algorithm, namely D^0 is given. Note that, the i, j 'th element of matrix D^0 is

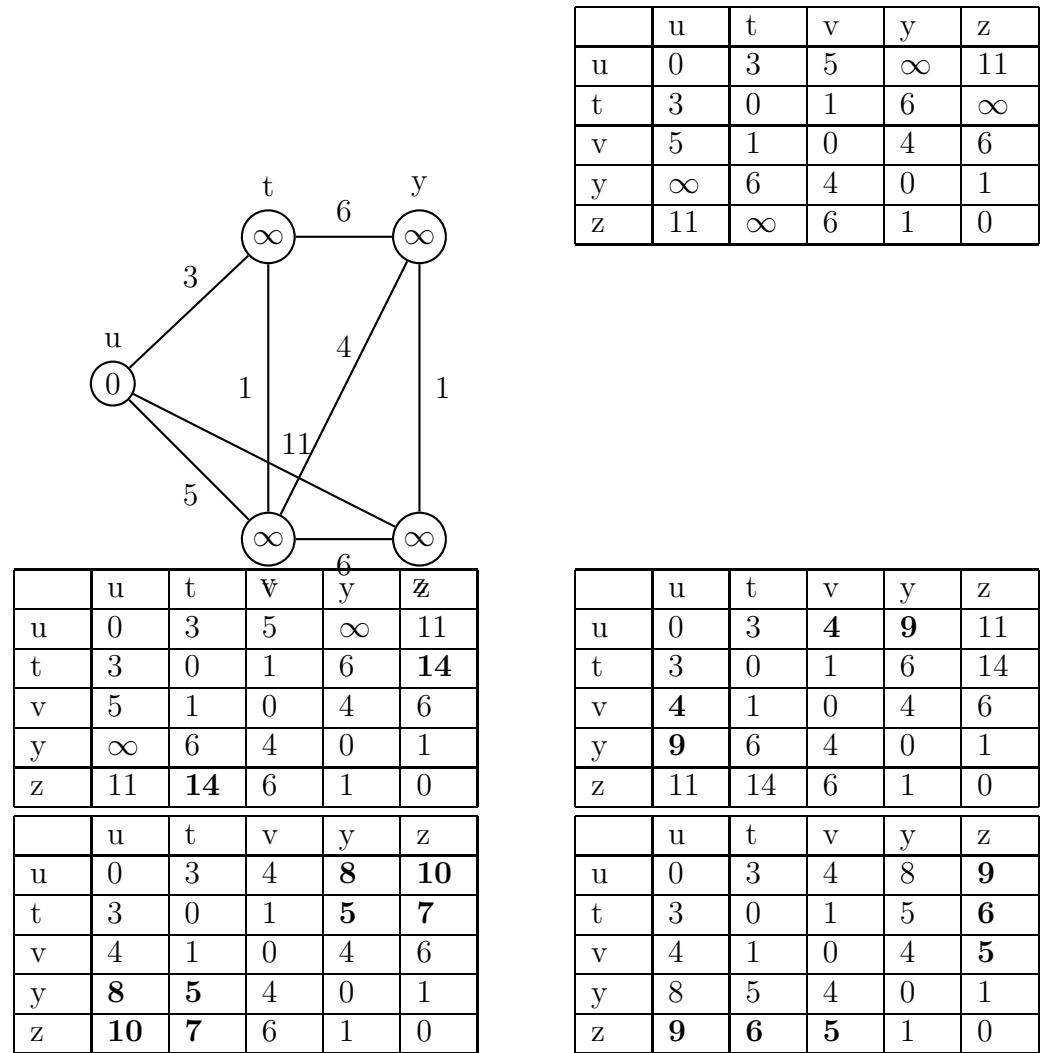


Figure 9.12: Application of 4 steps of Floyd-Warshall algorithm on a graph of 5 vertices

- the weight of an edge if that edge connects the vertex v_i to the vertex v_j ,
- ∞ if there are no such edge,
- 0 if $i = j$ since the shortest path from a vertex to the same vertex is 0.

9.5. APPLICATION: MINIMUM SPANNING TREE

In the next phases;

- (1) The i, j 'th element of the matrix D^1 contains, the length of the shorter path if we can find such a shorter path from vertex i to vertex j using the first vertex (u), otherwise the value of the i, j 'th element of the matrix D^0 . According to this, in the example graph only the length of the path tz gets shorter (shown in red color) and is 14. $tu + uz < tz$, $3 + 11 < \infty$.
- (2) The elements whose value is changed in the matrix D^2 according to the matrix D^1 (the ones for which a shorter path is found using vertex t) are given below.

$$ut + tv < uv, 3 + 1 < 5$$

$$ut + ty < uy, 3 + 6 < \infty$$

- (3) The elements whose value is changed in the matrix D^3 according to the matrix D^2 (the ones for which a shorter path is found using vertex v) are given below.

$$uv + vy < uy, 4 + 4 < 9$$

$$uv + vz < uz, 4 + 6 < 11$$

$$tv + vy < ty, 1 + 4 < 6$$

$$tv + vz < tz, 1 + 6 < 8$$

- (4) The elements whose value is changed in the matrix D^4 according to the matrix D^3 (the ones for which a shorter path is found using vertex y) are given below.

$$uy + yz < uz, 8 + 1 < 10$$

$$ty + yz < tz, 5 + 1 < 7$$

$$vy + yz < vz, 4 + 1 < 6$$

9.5 Application: Minimum Spanning Tree

If we span all vertices of an undirected graph G , which consists of N vertices, by choosing $N - 1$ edges, we will get a structure called tree. In this case span

CHAPTER 9. GRAPH

means, any vertex of the graph is reachable from any vertex by the help of the edges. Being the number of edges $N - 1$, provides that these edges form up a tree. On the other hand, the minimum spanning tree is a tree whose sum of lengths of edges is minimum across all spanning trees. A graph may have a single minimum spanning tree, or may have more than one spanning trees.

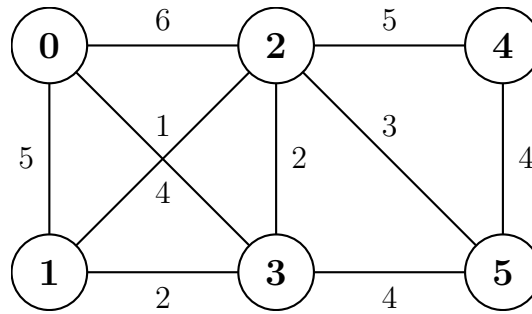


Figure 9.13: A weighted graph consisting of six vertices

Figure 9.13 shows an undirected weighted graph G consisting of six vertices. On the other hand, Figure 9.14 shows two spanning trees which span the same graph G . The tree shown in the Figure 9.14(a) consists of 5 edges (0 - 2, 2 - 4, 4 - 5, 5 - 3, 3 - 1), its total weight is $6 + 5 + 4 + 4 + 2 = 21$. The tree shown in Figure 9.14(b) consists of 5 edges again (0 - 2, 2 - 1, 2 - 3, 2 - 5, 2 - 4), but its total weight is $6 + 1 + 2 + 3 + 5 = 17$. By using both trees it is possible to go from one vertex to another in this graph. For example, in Figure 9.14(a), we can go from vertex 0 to vertex 1 by following path 0 - 2 - 4 - 5 - 3 - 1, in Figure 9.14(b), we can go again from vertex 0 to vertex 2 by following path 0 - 2 - 1.

Figure 9.15 shows two different spanning trees corresponding to the graph in Figure 9.13. As we explained above, there can be more than one spanning tree for a given graph. We see a similar case also in this example. The total weight of both trees are $1 + 2 + 3 + 4 + 4 = 14$, and the edges forming up these trees are nearly the same except one edge on each side. The tree in Figure 9.15(a) has the edge 2-3, but not the edge 1-3; whereas the tree in Figure 9.15(b) has the edge 1-3, but not the edge 2-3. Since both edges have the same weight 2, both trees have the same total weight.

9.5. APPLICATION: MINIMUM SPANNING TREE

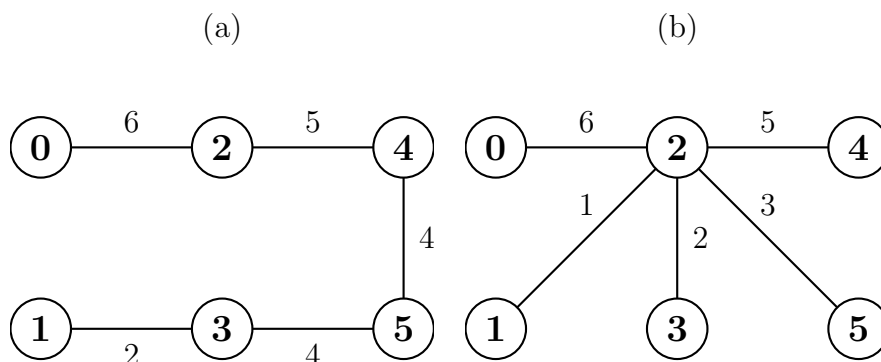


Figure 9.14: Two trees spanning the graph in Figure 9.13

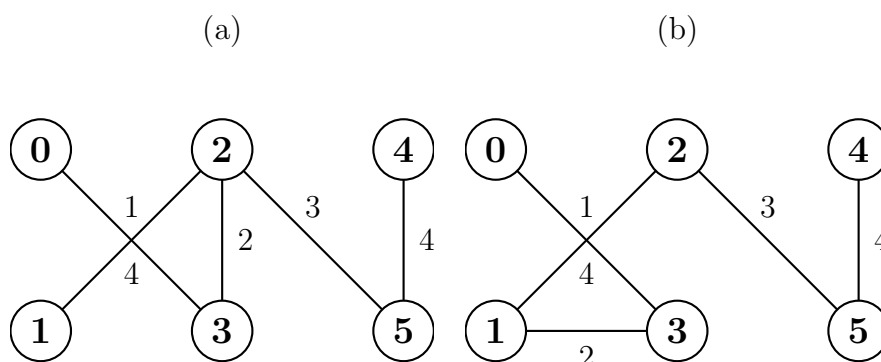


Figure 9.15: Two minimum spanning trees for the graph in Figure 9.13

9.5.1 Kruskal's Algorithm

Kruskal's minimum spanning tree algorithm belongs to the class of greedy algorithms, and starts with an empty graph. At each step, we select the edge with minimum weight as the candidate edge. When the addition of this edge (i) produces a cycle in the graph, this edge is not added to the resulting graph, (ii) does not produce a cycle in the graph, this edge is added to the resulting graph. When we add $N - 1$ edges to the resulting graph, the algorithm is finished. The greediness of the algorithm stems from the fact that, each time the algorithm selects the most advantageous option (the edge with the minimum weight).

CHAPTER 9. GRAPH

Table 9.16: The algorithm that returns all edges in a graph of adjacency list representation as a linked list (C++)

```
1 Edge* edgeList(int& edgeCount){
2     Edge* list;
3     edgeCount = 0;
4     for (int i = 0; i < vertexCount; i++){
5         Edge* edge = edges[i].getHead();
6         while (edge != nullptr){
7             edgeCount++;
8             edge = edge.getNext();
9         }
10    }
11    list = new Edge[edgeCount];
12    int index = 0;
13    for (int i = 0; i < vertexCount; i++){
14        Edge* edge = edges[i].getHead();
15        while (edge != nullptr){
16            list [index] = Edge(edge.getFrom(), edge.getTo(), edge.getWeight());
17            index++;
18            edge = edge.getNext();
19        }
20    }
21    return list;
22 }
```

In the Kruskal algorithm, the edges are processed in increasing order of their weights. For this reason, sorting the edges in increasing order is an important step in determining the time complexity of the algorithm. Neither the adjacency matrix, nor the adjacency list representations are not suitable for sorting the edges. Instead of these structures, the fastest solution would be, to put all edges in a linked list (or array) and then sort the edges in this linked list (or array) with respect to their weights.

Table 9.17 shows the algorithm that returns the edges in a graph of adjacency list representation as a linked list. First the new linked list is defined (Line 5). Then for each vertex (Line 6), we traverse its adjacency list (Lines 7, 11). The aim of traversing the list is to add each element to the new edge list. If the original elements in the adjacency list are added to the new edge list, the links between them will be changed and one can not traverse the original adjacency list. Therefore, in order not to modify the original

9.5. APPLICATION: MINIMUM SPANNING TREE

Table 9.17: The algorithm that returns all edges in a graph of adjacency list representation as a linked list (Java)

```
1 Edge[] edgeList(){
2     Edge[] list ;
3     int edgeCount = 0;
4     for (int i = 0; i < vertexCount; i++){
5         Edge edge = edges[i].getHead();
6         while (edge != null){
7             edgeCount++;
8             edge = edge.getNext();
9         }
10    }
11    list = new Edge[edgeCount];
12    int index = 0;
13    for (int i = 0; i < vertexCount; i++){
14        Edge edge = edges[i].getHead();
15        while (edge != null){
16            list [index] = new Edge(edge.getFrom(), edge.getTo(), edge.getWeight());
17            index++;
18            edge = edge.getNext();
19        }
20    }
21    return list;
22 }
```

adjacency list, each edge must be recreated (Line 9) and will be added to the linked list that will contain all edges (Line 10). As a last step, the new linked list is returned (Line 14).

In Kruskal algorithm, when the edges are added to the resulting graph, one should check if there exists a cycle or not. Checking the existence of a cycle in a graph is an hard problem, and fastest algorithms have the time complexity of $\mathcal{O}(N^2)$ where N represents the number of vertices in the graph. Therefore, if at each addition of an edge one checks the existence of a cycle, the time complexity of Kruskal algorithm will be $\mathcal{O}(N^3)$.

Instead of checking the existence of a cycle at each time, one can check the existence of a cycle using the disjoint set data structure for that edge only. Table 9.18 shows Kruskal algorithm that uses the disjoint set data structure. First all vertices are stored as sets in a disjoint set structure (Line 5). Before adding each edge, one checks if the starting (Line 11) and ending (Line 12)

CHAPTER 9. GRAPH

Table 9.18: Kruskal algorithm that finds the minimum spanning tree of a given graph

```
1 void kruskal(){
2     int edgeCount = 0, i, count;
3     DisjointSet sets = DisjointSet(vertexCount);
4     Edge* list = edgeList(count);
5     i = 0;
6     while (edgeCount < vertexCount - 1){
7         int fromNode = list[i].getFrom();
8         int toNode = list[i].getTo();
9         if (sets.findSetRecursive(fromNode) != sets.findSetRecursive(toNode)){
10             sets.unionOfSets(fromNode, toNode);
11             edgeCount++;
12         }
13         i++;
14     }
15 }
```

```
1 void kruskal(){
2     int edgeCount = 0, i;
3     DisjointSet sets = new DisjointSet(vertexCount);
4     Edge[] list = edgeList();
5     Arrays.sort(list);
6     i = 0;
7     while (edgeCount < vertexCount - 1){
8         int fromNode = list[i].getFrom();
9         int toNode = list[i].getTo();
10        if (sets.findSetRecursive(fromNode) != sets.findSetRecursive(toNode)){
11            sets.unionOfSets(fromNode, toNode);
12            edgeCount++;
13        }
14        i++;
15    }
16 }
```

vertices are in the same disjoint set or not (Line 13). If both vertices are in the same disjoint set, one can reach from one to the other by a path and adding the corresponding that edge will form a cycle. If both vertices are in separate disjoint sets, both the edge is added to the resulting graph (Line 15) and the disjoint sets of both vertices are merged (Line 14). When the number of added edges is $N - 1$, the algorithm is finished (Line 10).

9.5. APPLICATION: MINIMUM SPANNING TREE

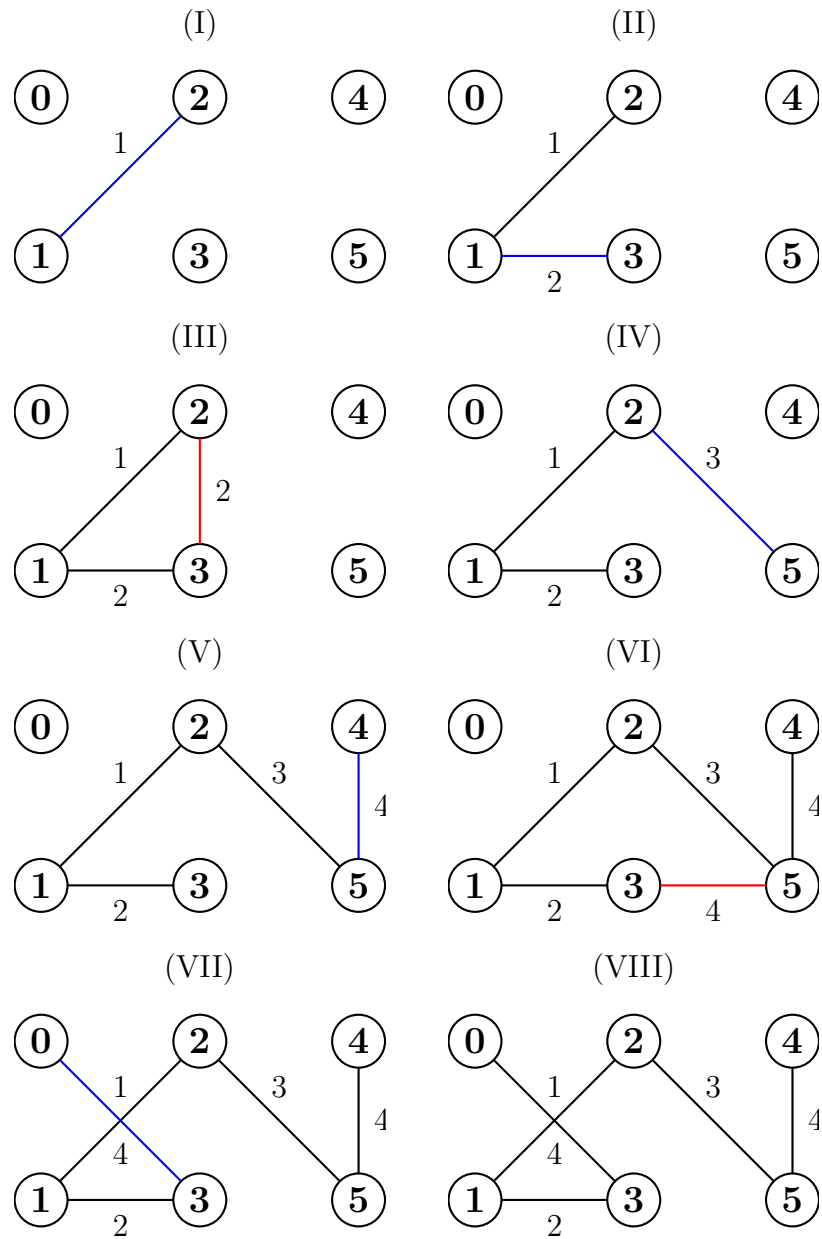


Figure 9.16: Application of Kruskal algorithm to the graph in Figure 9.13

CHAPTER 9. GRAPH

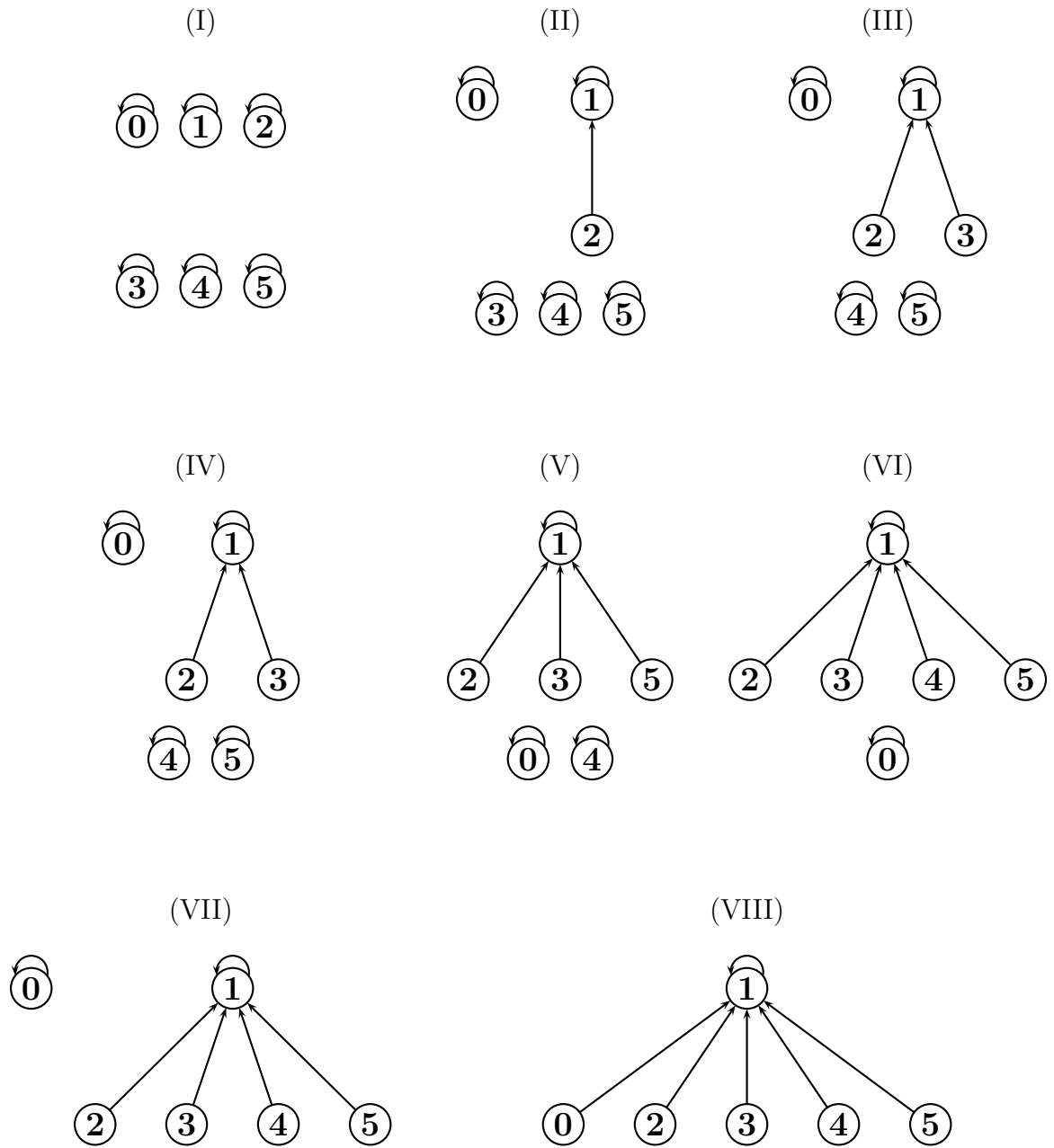


Figure 9.17: The disjoint set structure used in applying Kruskal algorithm to the graph in Figure 9.13

9.5. APPLICATION: MINIMUM SPANNING TREE

Figure 9.16 shows the application of Kruskal algorithm to the graph in Figure 9.13 and Figure 9.17 shows the disjoint set a while applying the same algorithm. The steps of the algorithm are:

- (I) The edge 1-2 has a minimum weight of 1. Since 1 and 2 are in separate disjoint sets, edge 1-2 is added to the resulting graph.
- (II) The edge 1-3 has a minimum weight of 2. Since 1 and 3 are in separate disjoint sets, edge 1-3 is added to the resulting graph.
- (III) The edge 2-3 has a minimum weight of 2. Since 2 and 3 are in the same disjoint sets, this edge can not be added to the resulting graph.
- (IV) The edge 2-5 has a minimum weight of 3. Since 2 and 5 are in separate disjoint sets, edge 2-5 is added to the resulting graph.
- (V) The edge 4-5 has a minimum weight of 4. Since 4 and 5 are in separate disjoint sets, edge 4-5 is added to the resulting graph.
- (VI) The edge 3-5 has a minimum weight of 4. Since 3 and 5 are in the same disjoint sets, this edge can not be added to the resulting graph.
- (VII) The edge 0-3 has a minimum weight of 4. Since 0 and 3 are in separate disjoint sets, edge 0-3 is added to the resulting graph.

9.5.2 Prim's Algorithm

Prim's minimum spanning tree algorithm also belongs to the class of greedy algorithms, and starts with an empty graph. Different than the Kruskal algorithm, at each step of the Prim algorithm we have a tree. Prim's algorithm can start from any vertex and works with two disjoint sets of vertices. The first vertex set K_1 , grows as we add edges to the resulting tree, the second vertex set $K_2 = N - K_1$, shrinks as we add edges to the resulting tree. At each stage, we choose the minimum weight edge from the edges that connect the set K_1 with the set K_2 . The vertex of this edge that belongs to the set K_2 is removed from this set and added to the set K_1 . Yet similar to the Kruskal algorithm, when we add $N - 1$ edges to the resulting tree, the algorithm is finished.

Table 9.20 shows the Prim's minimum spanning tree algorithm. Note that, Prim's algorithm and Dijkstra's shortest path algorithm are very similar

CHAPTER 9. GRAPH

Table 9.19: Prim's algorithm that finds the minimum spanning tree of a given graph (C++)

```
1 void prim(){
2     Path* paths = new Path[vertexCount];
3     for (int i = 0; i < vertexCount; i++){
4         paths[i] = Path(INT_MAX, -1);
5     }
6     paths[source].setDistance(0);
7     Heap heap = Heap(vertexCount);
8     for (int i = 0; i < vertexCount; i++){
9         heap.insert(HeapNode(paths[i].getDistance(), i));
10    }
11    while (!heap.isEmpty()){
12        HeapNode node = heap.deleteMin();
13        int from = node.getName();
14        Edge* edge = edges[from].getHead();
15        while (edge != nullptr){
16            int to = edge->getTo();
17            if (paths[to].getDistance() > edge->getWeight()){
18                int position = heap.search(to);
19                heap.update(position, edge->getWeight());
20                paths[to].setDistance(edge->getWeight());
21                paths[to].setPrevious(from);
22            }
23            edge = edge->getNext();
24        }
25    }
26 }
```

to each other. In the Dijkstra's shortest path algorithm, the shortest distance of each vertex to the initial vertex is stored in a min-heap, whereas in Prim's minimum spanning tree algorithm, the shortest distance of each vertex in set K_2 connecting to set K_1 is stored in a min-heap. At the first stage, the values of all vertices except the initial vertex are ∞ in the min-heap (Line 9) and these values are added to the heap T (Lines 14-17).

The vertex belonging to the set K_2 that is connected with the shortest edge to the set K_1 is the vertex on the top of the heap (u). u is extracted by the `deleteMin` function at each step (Line 19, 20). The adjacent vertices of u can be obtained by traversing the adjacency linked list of u (Line 21, 22, 30). For each adjacent vertex v of u (Line 23), if v belongs to the set K_2 and the

9.5. APPLICATION: MINIMUM SPANNING TREE

Table 9.20: Prim's algorithm that finds the minimum spanning tree of a given graph (Java)

```

1 void prim(){
2   Path[] paths = new Path[vertexCount];
3   for (int i = 0; i < vertexCount; i++){
4     paths[i] = new Path(Integer.MAX_VALUE, -1);
5   }
6   paths[source].setDistance(0);
7   Heap heap = new Heap(vertexCount);
8   for (int i = 0; i < vertexCount; i++){
9     heap.insert(new HeapNode(paths[i].getDistance(), i));
10  }
11  while (!heap.isEmpty()){
12    HeapNode node = heap.deleteMin();
13    int from = node.getName();
14    Edge edge = edges[from].getHead();
15    while (edge != null){
16      int to = edge.getTo();
17      if (paths[to].getDistance() > edge.getWeight()){
18        int position = heap.search(to);
19        heap.update(position, edge.getWeight());
20        paths[to].setDistance(edge.getWeight());
21        paths[to].setPrevious(from);
22      }
23      edge = edge.getNext();
24    }
25  }
26 }

```

length of the edge connecting u and v is shorter than the previous shortest length (Line 24), this shortest length will be updated in the heap (Line 27).

Figure 9.18 shows the application of Prim's algorithm on the graph in Figure 9.13, whereas Figure 9.19 shows the min-heap T while we apply the same algorithm. The steps of the algorithm are

- (I) The initial vertex is 0 and the elements of the min-heap T are $\{3, 1, 2, 4, 5\}$ respectively. The initial distances of these vertices are $\{4, 5, 6, \infty, \infty\}$ respectively.
- (II) Since the minimum element of the heap is 3, it is selected. Vertex 3 is added to the set K_1 . One can reach from vertex 3 to the set K_2 using

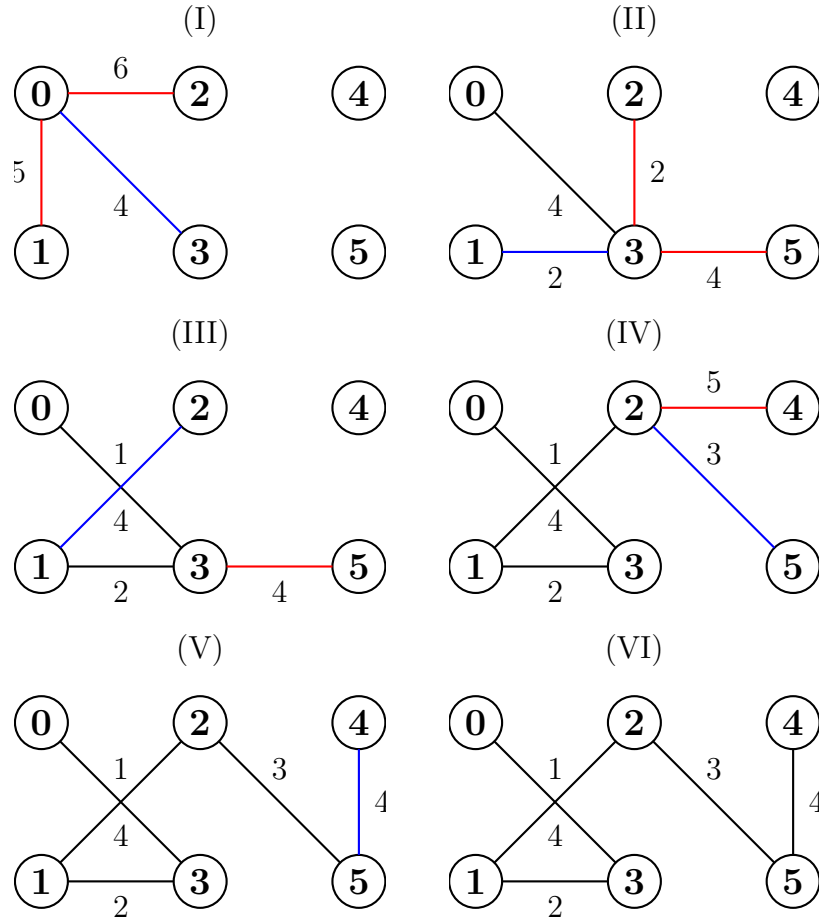


Figure 9.18: Application of Prim's algorithm to the graph in Figure 9.13

vertices 1, 2, and 5. The length of these vertices will be changed and the new elements of the min-heap will be $\{1, 5, 2, 4\}$ and their lengths $\{2, 4, 2, \infty\}$ respectively.

- (III) Since the minimum element of the heap is 1, it is selected. Vertex 1 is added to the set K_1 . One can reach from vertex 1 to the set K_2 using vertex 2 only. The new elements of the min-heap will be $\{2, 5, 4\}$ and their lengths $\{1, 4, \infty\}$ respectively.

- (IV) Since the minimum element of the heap is 2, it is selected. Vertex 2 is

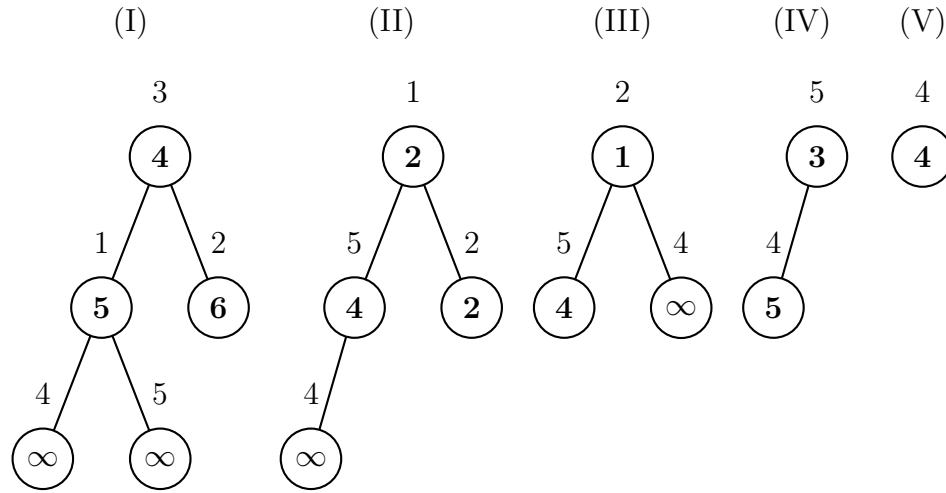


Figure 9.19: Min-heap T in the application of Prim's algorithm in Figure 9.18

added to the set K_1 . One can reach from vertex 1 to the set K_2 using vertices 4 and 5. The new elements of the min-heap will be $\{5, 4\}$ and their lengths $\{3, 5\}$ respectively.

(V) Since the minimum element of the heap is 5, it is selected. Vertex 5 is added to the set K_1 . One can reach from vertex 5 to the set K_2 using vertex 4 only. The new elements of the min-heap will be $\{4\}$ and their lengths $\{4\}$ respectively.

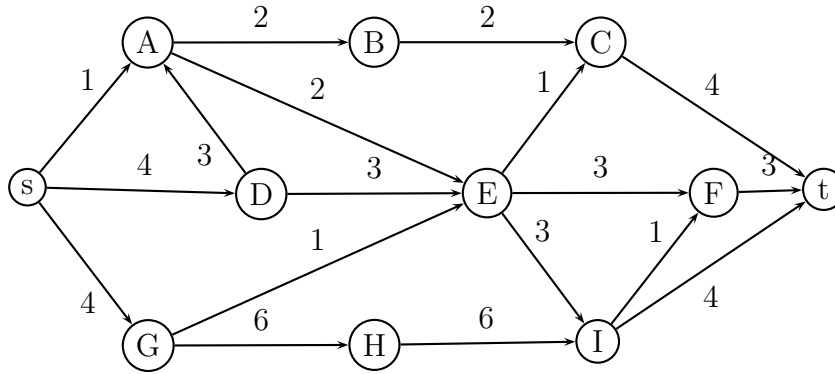
(VI) The last element of the heap 4 is selected and added to the set K_1 .

9.6 Notes

Adjacency list representation in the graphs is first proposed by Hopcroft [8]. Dijkstra's shortest path algorithm is given in [3]. Bellman's shortest path algorithm can be found in [2]. The details of the Floyd-Warshall algorithm can be seen in the works of Floyd [4] and Warshall [15]. Depth first search and some graph applications of that approach can be obtained by Tarjan [14].

9.7 Solved Exercises

- Find the shortest path from s to t in the following graph.



The shortest path lengths after each step and the vertex used at that step (in boldface) is given below.

- 1. Step: **s(0)**, $A(\infty)$, $B(\infty)$, $C(\infty)$, $D(\infty)$, $E(\infty)$, $F(\infty)$, $G(\infty)$, $H(\infty)$, $I(\infty)$, $t(\infty)$. The shortest path lengths of vertices A ($0 + 1 < \infty$), D ($0 + 4 < \infty$), and G ($0 + 6 < \infty$) are changed.
- 2. Step: **A(1)**, $D(4)$, $G(6)$, $B(\infty)$, $C(\infty)$, $E(\infty)$, $F(\infty)$, $H(\infty)$, $I(\infty)$, $t(\infty)$. The shortest path lengths of vertices B ($1 + 2 < \infty$) and E ($1 + 2 < \infty$) are changed.
- 3. Step: **B(3)**, $E(3)$, $D(4)$, $G(6)$, $C(\infty)$, $F(\infty)$, $H(\infty)$, $I(\infty)$, $t(\infty)$. The shortest path length of vertex C ($3 + 2 < \infty$) is changed.
- 4. Step: **E(3)**, $D(4)$, $C(5)$, $G(6)$, $F(\infty)$, $H(\infty)$, $I(\infty)$, $t(\infty)$. The shortest path lengths of vertices C ($3 + 1 < 5$), F ($3 + 3 < \infty$), and I ($3 + 3 < \infty$) are changed.
- 5. Step: **D(4)**, $C(4)$, $G(6)$, $F(6)$, $I(6)$, $H(\infty)$, $t(\infty)$.
- 6. Step: **C(4)**, $G(6)$, $F(6)$, $I(6)$, $H(\infty)$, $t(\infty)$. The shortest path length of vertex t ($4 + 4 < \infty$) is changed.

9.7. SOLVED EXERCISES

- 7. Step: **G(6)**, F(6), I(6), t(8), H(∞). The shortest path length of vertex H ($6 + 6 < \infty$) is changed.
 - 8. Step: **F(6)**, I(6), t(8), H(12).
 - 9. Step: **I(6)**, t(8), H(12).
 - 10. Step: **t(8)**, H(12).
 - 11. Step: **H(12)**.
2. In two coloring problem, the aim is to find out if a graph can be colored with just two colors. To color the graph, each vertex is given one of two colors, and no edge connects identically colored vertices. Write a function that determines if a graph is two-colorable or not.

boolean twoColorable()

In order to solve the problem, we traverse the graph using depth first search and we try to color the adjacent vertices in different colors. If we can not color the two vertices belonging to the same edge in two different colors, the function returns 0. `color[i]` represents the color of vertex i . On the other hand, if `color[i]` is -1, vertex i was not colored before (Lines 3-4).

For each vertex v (Line 5), if that vertex is not colored before (Line 8), vertex v is colored with color 0 and the recursive function **paint** is called (Line 9). **paint** function tries to color the adjacent vertices of v in a different color. If the adjacent vertex y of vertex v is not colored before (Line 18), it is colored in a reverse color of v (Line 19-20), or if it is colored with the same color (Line 24-25) the function will return 0.

CHAPTER 9. GRAPH

```
1  boolean twoColorable(){
2      int v;
3      for (v = 0; v < N; v++)
4          color[v] = -1;
5      for (v = 0; v < N; v++)
6          if (color[v] != -1){
7              color[v] = 0;
8              if (!paint(v))
9                  return false;
10         }
11     return true;
12 }
13 boolean paint(int v){
14     Node<Edge> e;
15     int y;
16     e = edges[v].first ;
17     while (e != null){
18         y = e.data.to;
19         if (color[y] == -1){
20             color[y] = 1 - color[v];
21             if (!paint(y))
22                 return false;
23         } else
24             if (color[y] == color[v])
25                 return false;
26         e = e.next;
27     }
28     return true;
29 }
```

3. For each node n in an undirected graph, let $\text{twodegree}[n]$ be the sum of degrees of n 's neighbors. Write a function that computes the array twodegree .

int [] twoDegree()

The function that computes the array two degree for a given graph is shown below. $d[u]$, represents the sum of degrees of the adjacent vertices of u . First we allocate memory for the array d (Line 5). Then for each vertex u (Line 6), we traverse (Line 8-9, 12) all adjacent vertices v to that vertex (Line 10). The degree of the vertex v is the number of adjacent vertices of v (Line 11). What we should do is to add the length of the adjacency list of v to $d[u]$ (Line 11).

```

1  int[] twoDegree(){
2      int u, v;
3      int[] d;
4      Node<Edge> e;
5      d = new int[N];
6      for (u = 0; u < N; u++){
7          d[u] = 0;
8          e = edges[u].first ;
9          while (e != null){
10             v = e.data.to;
11             d[u] += edges[v].nodeCount();
12             e = e.next;
13         }
14     }
15     return d;
16 }

```

4. The objective of the Kevin Bacon game is to connect a movie actor to Kevin Bacon via shared movie roles. The minimum number of links is an actor's Bacon number. For instance, Tom Hanks has a Bacon number of 1; he was in Apollo 13 with Kevin Bacon. Sally Fields has a Bacon number of 2, because she was in Forrest Gump with Tom Hanks, who was in Apollo 13 with Kevin Bacon. Given that you have an input file where (i) first line contains two integers, namely number of actors N and number of films F (ii) remaining lines form N groups where each group contains the number of actor pairs and the index of actor pairs. Kevin Bacon has an index of 0. Given the input file, write a function that determines the Bacon number of each actor.

In order to solve the problem, first a graph c is constructed (Line 9). In this graph, the vertices correspond to the actors, the edges correspond to the films where both actors are played together. If actor j plays with actor k in the same film, one will add an edge from vertex j to vertex k and another edge from vertex k to vertex j (Lines 17-18). Kevin Bacon number of an actor is equal to the length of the shortest path from that actor to the vertex corresponding to Kevin Bacon (vertex with index 0). What should be done is to run the Dijkstra's shortest path algorithm on the constructed graph where Kevin Bacon vertex is the initial vertex (Line 22).

```

int[] kevinBacon(String inputFile) throws FileNotFoundException{
    Scanner file ;
    Graph c;
    int i, j, k, v, actorCount, filmCount, cast;
    int film [];
    file = new Scanner(new File(inputFile));
    actorCount = file.nextInt ();
    filmCount = file.nextInt ();
    c = new Graph(actorCount);
    for (i = 0; i < filmCount; i++){
        cast = file.nextInt ();
        film = new int[cast];
        for (j = 0; j < cast; j++)
            film[j] = file.nextInt ();
        for (j = 0; j < cast; j++)
            for (k = j + 1; k < cast; k++){
                c.addEdge(j, k, 1);
                c.addEdge(k, j, 1);
            }
    }
    return c.dijkstra();
}

```

9.8 Exercises

1. What would be the shortest path from s to t in the graph given in solved exercise 1 if all edge lengths were 1.
2. Modify Dijkstra's shortest path algorithm such that if there is a shortest path from vertex u to vertex v , it selects the path with minimum number of edges.

void dijkstra ()

3. Write a function that computes the number of edges in a graph. Write functions for both adjacency-matrix and adjacency-list representations.

int edgeCount()

4. In a given graph, the degree of a vertex is the sum of incoming edges and outgoing edges. Given the index of a vertex, write a function

9.8. EXERCISES

that determines the degree of that vertex. Write functions for both adjacency-matrix and adjacency-list representations.

int degree(**int** index)

5. Given a graph, write a function that determines if it is undirected. Write functions for both adjacency-matrix and adjacency-list representations.

boolean isUndirected()

6. Write a function that converts a graph with adjacency matrix representation to a graph with adjacency list representation.

Graph matrixToList()

7. Write a function that converts a graph with adjacency list representation to a graph with adjacency matrix representation.

Graph listToMatrix()

8. Write a function that computes the number of paths between all vertices in a graph.

int [][] pathCount()

9. Write a function that finds the minimum weight in a weighted graph. Write the function for both adjacency matrix and adjacency list representation.

int minimumWeight()

10. Write a function that finds the average weight in a weighted graph. Write the function for both adjacency matrix and adjacency list representation.

double averageWeight()

11. Implement the graph data structure as an array of hash tables. Node i is connected to node j if the i 'th hash table contains a weighted edge j and its weight (for unweighted graphs edge weight is 1). Your implementation must contain the data structure, constructor function, two addEdge functions for unweighted and weighted graphs respectively. You must also implement hash function for the Edge class.

12. Write a function that finds the median weight in a weighted graph. Write the function for adjacency matrix representation.

`int medianWeight()`

13. Given the adjacency list representation of a graph G , find the incoming nodes to a given node i .

`LinkedList incomingNodes(int i)`

9.9 Problems

1. Given the match scores of a volleyball league with N teams and two team indices X and Y , write a function that determines if X is better than Y . X is better than Y with a circular argument if we can find a set teams Z_1, Z_2, \dots, Z_k , where X beats Z_1 , Z_1 beats Z_2 , \dots , Z_{k-1} beats Z_k , and Z_k beats Y .
2. A word can be changed to another word by a 1-character substitution. Given a file of five letter words in English dictionary, write a function to determine if a word A can be transformed to a word B by a series of 1-character substitutions. Note that all intermediate words must be in the dictionary.
3. The reverse of a directed graph is another directed graph on the same vertex set, but with all edges reversed. Write a function that computes the reverse of a graph in adjacency matrix representation.

`Graph reverseGraph()`

4. The reverse of a directed graph is another directed graph on the same vertex set, but with all edges reversed. Write a function that computes the reverse of a graph in adjacency list representation.

`Graph reverseGraph()`

5. A node in a graph is said to be an *island* if there are no incoming edges to it and no outgoing edges from it. Given the adjacency matrix representation of unweighted graph G , write a function that calculates the number of islands in that graph.

`int islands()`

9.9. PROBLEMS

6. A node in a web graph is called a source, if it has no incoming edges. Write a method that finds the number of sources in a graph. Write the function for both adjacency list and adjacency matrix representations.

int numberOfSources()

7. A node in a web graph is called a hub, if it has more incoming edges than outgoing edges. Write a method that finds the number of hubs in a graph. Write the function for both adjacency list and adjacency matrix representations.

int numberOfHubs()

8. A graph represents a ring topology if all the nodes create a circular path. Each node is connected to two others, like points on a circle. Write a class method in Graph class for adjacency list representation which checks if the corresponding graph is circular or not.

boolean isCircular()

9. A bipartite graph is a graph such that vertices of the graph can be partitioned into two subsets such that no edge has both its vertices in the same subset. Write a method for adjacency list representation which checks if the corresponding graph is bipartite or not. Hint: Use Depth or breath first search to traverse the graph.

boolean isBipartite()

10. Write a method that checks if the graph is complete or not. Write the function for both adjacency matrix and adjacency list representations. Do not use any class or external methods.

boolean isComplete()

11. Write a method which checks if two graphs are the same. Assume that, the method is written in the Adjacency list representation of a graph.

1 **boolean** isSame(Graph g)

12. Write a method that checks if the graph is complete bipartite graph or not. Write the function for adjacency matrix representation. A graph (V_1, V_2) is said to be a complete bipartite graph if every vertex in V_1 is connected to every vertex of V_2 .

CHAPTER 9. GRAPH

boolean isCompleteBipartite()

13. Write a function that computes the number of bidirectional edges in a graph. Write the function for adjacency list representation.

int bidirectionalEdges ()

14. Modify the breadth first search **linked list** implementation such that it will store the indexes of the nodes of the shortest paths from the start node in *paths* parameter.

void shortest(**int**** paths, **bool*** visited , **int** start)

At the end of the execution, the indexes in the paths[i] array will show the path visited from node *start* to node *i*. You may assume that the visited array is initialized to false and paths array is already allocated.

15. Modify the breadth first search **linked list** implementation such that it will store the length of the shortest paths from the start node in *lengths* parameter.

void shortest(**int*** lengths, **int** start)

At the end of the execution, lengths[i] will show the shortest path length from node *start* to node *i*. You may assume that the path length elements are initialized to *vertexCount* (number of nodes, which should be larger than any possible shortest path) when you call the function.

16. Write the method in linked list implementation

Graph constructGraphFromNumbers(**int** N)

which constructs a graph from numbers 0, 1, 2, ..., N - 1; where the numbers represent the node indexes and two nodes are connected if they have common divisor other than 1.

17. Write the method in array implementation

int numberOfCompleteSubGraphs()

which returns the number of complete subgraphs in the graph. A complete graph is a graph, in which all vertices are connected to all vertices. Assume that the graph only consists of complete subgraphs of size > 1, there are no extra vertices, which is not in a complete subgraph. You are not allowed to use depth first search or breadth first search. In the graph below (1, 2, 5), (3, 6) and (4, 7) are complete subgraphs.

	1	2	3	4	5	6	7
1	0	1	0	0	1	0	0
2	1	0	0	0	1	0	0
3	0	0	0	0	0	1	0
4	0	0	0	0	0	0	1
5	1	1	0	0	0	0	0
6	0	0	1	0	0	0	0
7	0	0	0	1	0	0	0

18. Write the method in linked list implementation

`int* twoHops(int index)`

which returns the indexes of the nodes which are reachable by two hops, that is, it will consist of all indexes j , where one goes from index node to node i , then from node i to node j . The size of the returning array should be as much as needed. If there are two or more ways to go to a node j , then j must appear that many times in the list (no need to sort or check for duplicates).

19. Write the method in array implementation

`bool isSubGraph(const Graph& g)`

which returns true if g is a subgraph of the current graph, false otherwise. A graph G_1 is a subgraph of graph G_2 if every edge of graph G_1 is also an edge in graph G_2 .

20. Write the method in linked list implementation of **Graph** class

`Graph intersection(const Graph& g2, int v)`

that returns a new graph formed by adding edges which exist both in the original graph and $g2$. You may assume both graphs are unweighted.

21. For a directed weighted graph, write the method in linked list implementation

`int shortestIn2Hops(int index1, int index2)`

which returns the shortest distance between the nodes `index1` and `index2` by two hops, that is, it will return of the shortest of all paths, where one goes from `index1` node to node i , then from node i to node `index2`.

CHAPTER 9. GRAPH

22. For a directed weighted graph, write the method in array implementation

`int lengthOfCircle ()`

which returns length of the circle assuming that the graph is a circular graph. A graph is circular if all the nodes create a circular path. Each node is connected to two others, like points on a circle.

23. Write the method in linked list implementation of **Graph** class

`Graph merge(const Graph& g2, int v)`

that returns a new graph formed by adding edges which exist in the original graph or g2. You may assume both graphs are weighted, if an edge exists in both graphs, add the resulting edge with the sum of their weights. You are not allowed to use any linked list methods.

24. Write the method in array (adjacency matrix) implementation

`bool isStarGraph()`

which returns whether graph is a star graph or not. Star graph is obtained by connecting a node to all the remaining nodes. If a graph has n nodes, there are n-1 edges as shown in example star graph below. You are not allowed to use depth first search or breadth first search.

25. Write the method in linked list implementation

`Graph* inverseGraph()`

which constructs an inverse graph of a given graph in linked list implementation. In inverse graph, two distinct vertices are adjacent if and only if they are not adjacent in the original graph. You are not allowed to use extra data structures apart from the constructed graph.

9.9. PROBLEMS

10

Sorting Algorithms

In this chapter we will examine the problem of sorting a list of elements. In order to simplify the problem, we will assume that the array **A** contains only integers. If one wants to sort more complex objects, the thing to do is to write external functions to implement the comparison operators (`isSmaller (<)` or `isLarger (>)`) for those objects.

Throughout all the chapter we will assume that all elements to be sorted can be stored in memory (RAM). If the elements to be sorted can not be sorted in memory, we will require external sorting algorithms. External sorting algorithms are outside the scope of this book.

Sorting algorithm can be examined in three separate categories. In the first category, there are algorithms those based on comparison of adjacent elements, namely insertion, selection, and bubble sort. The main idea of these algorithms can be summarized as to bring unordered elements into their correct orders. Due to their adjacent elements comparison nature, the average case complexity of these algorithms can not be smaller than $\mathcal{O}(N^2)$.

In the second category, there are faster sorting algorithms, those are complex, depend on complex data structures and algorithm principles. From these algorithms, the heap sort algorithm uses the heap data structure shown in Chapter 7, merge sort and quick sort algorithm are based on the divide-and-conquer idea. Since all three sorting algorithms are based on the comparison of elements, due to their nature, their average case complexity can not be smaller than $\mathcal{O}(N \log N)$.

The third and last category contains the sorting algorithms which are not based on comparison principles, and therefore they can be applied on

discrete data only. These algorithms, not like the previous algorithms, can not be applied to any type of data, and therefore are not accepted as general sorting algorithms. These algorithms work only if the assumptions on the data hold. The average time complexity of the algorithms can not be smaller than the reachable minimum time complexity of $\mathcal{O}(N)$.

10.1 Insertion Sort

Insertion sort algorithm uses the same approach that we use in sorting the playing cards. Starting from the second card, each time we *insert* the next card into an appropriate position with respect to the previously sorted cards. After each insertion operation, the inserted card and the cards before this card will be sorted. When the insertion is finished for the last card, all cards will be sorted.

Table 10.1 shows the insertion sort algorithm that sorts an array of A $A[0 \dots n - 1]$ consisting n integers. Variable t represents the element that is currently inserted at an appropriate position.

Table 10.1: Insertion Sort Algorithm

<pre> 1 void insertionSort(int* A, int size){ 2 for (int j = 1; j < size; j++){ 3 int t = A[j]; 4 int i = j - 1; 5 while (i >= 0 && A[i] > t){ 6 A[i + 1] = A[i]; 7 i--; 8 } 9 A[i + 1] = t; 10 } 11 }</pre>	<pre> void insertionSort(int[] A){ for (int j = 1; j < A.length; j++){ int t = A[j]; int i = j - 1; while (i >= 0 && A[i] > t){ A[i + 1] = A[i]; i--; } A[i + 1] = t; } }</pre>
--	--

The external for loop scans all elements to be sorted starting from the second element (Line 3). The inner while loop, on the contrary, continues until we either encounter an element that is smaller than t or we reach at the beginning of the array (Line 6). At each step, the elements are shifted one position right (Line 7). The aim of this shifting is to open a place for t

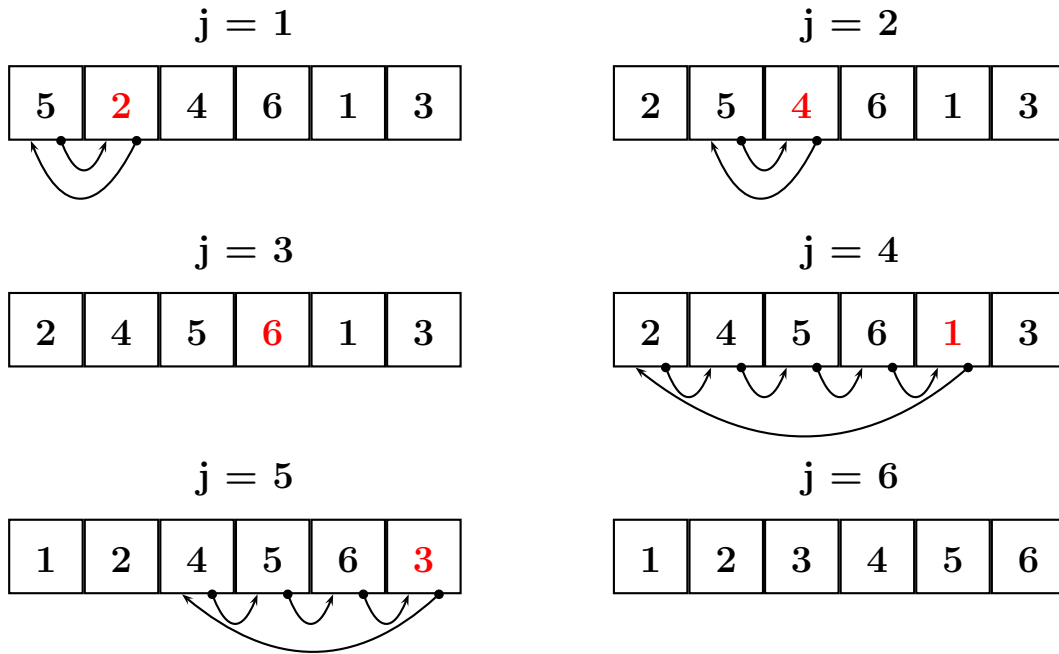


Figure 10.1: Insertion sort of an array of six elements

when we find the position to insert t . When we find the position to insert t , it is inserted into that position (Line 10).

Figure 10.1 shows the application of the insertion sort algorithm on an example array consisting of six elements. We first find the position of 2 and it is swapped with 5. At the second step, 4 is swapped with 5. At the third step, no swapping occurs, because 6 is in its place. At the fourth step, 1 is moved to the beginning of the array and all elements on the left hand side of it before, are shifted one position to the right. At the last step, in order to insert 3 between 2 and 4; numbers 4, 5, and 6 are shifted one right and 3 is inserted into its position.

10.2 Selection Sort

In selection sort, the elements are sorted by *selecting* them in increasing order one at a time. First we find the minimum element in the array and this element is swapped with the first element of the array. Then we find the second minimum element and this element is swapped with the second element of the array. When this operation is continued to the last element of the array, the array will be sorted.

Table 10.2: Selection Sort Algorithm

<pre> 1 void selectionSort(int* A, int size){ 2 for (int i = 0; i < size - 1; i++){ 3 int min = A[i]; 4 int pos = i; 5 for (int j = i + 1; j < size; j++){ 6 if (A[j] < min){ 7 min = A[j]; 8 pos = j; 9 } 10 } 11 if (pos != i){ 12 A[pos] = A[i]; 13 A[i] = min; 14 } 15 } 16 }</pre>	<pre> void selectionSort(int[] A){ for (int i = 0; i < A.length - 1; i++){ int min = A[i]; int pos = i; for (int j = i + 1; j < A.length; j++){ if (A[j] < min){ min = A[j]; pos = j; } } if (pos != i){ A[pos] = A[i]; A[i] = min; } } }</pre>
--	--

Table 10.2 shows the selection sort algorithm that sorts an array of A $A[0 \dots n - 1]$ consisting n integers. The external for loop finds the minimum, the second minimum, \dots , $n - 1$ 'th second minimum in order and swaps with the i 'th element of the array (Line 10-11). The internal for loop on the other hand, finds the minimum element between $A[i]$ and $A[n - 1]$. For this, we first set $A[i]$ as the minimum element (Line 4). Then we compare this minimum element with each element in this interval (Line 6). If the compared element is smaller than the minimum element, the minimum is updated (Line 7) and the position of the minimum element is saved (Line 8).

Figure 10.2 shows the application of selection sort algorithm to an array of six elements. First we find the position of the minimum element and

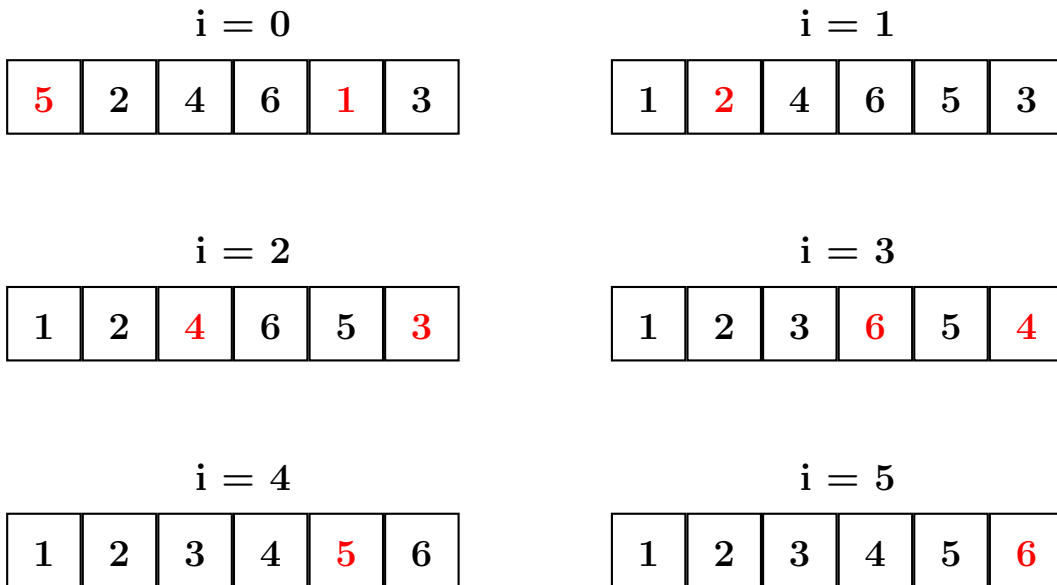


Figure 10.2: Selection sort of an array of six elements

is swapped with 5. Since second minimum element is already in its place, there is no swapping at the second step. At the third step, the third minimum element 3 is swapped with 4. At the fourth step, the fourth minimum element 4 is swapped with 6. At the fifth and last step, since the minimum element is already in its place, there is no swapping.

10.3 Bubble Sort

Bubble sort is a swap sort algorithm. Each number is compared with its neighbor and if the order of these two numbers is wrong, for example if the larger one is on the left side of the smaller one, these two numbers will be swapped. At each step, the first number is compared with the second number and if necessary they are swapped, the second number is compared with the

10.3. BUBBLE SORT

third number and if necessary they are swapped, etc. If at one step there is no swapping, the algorithm stops. Since the smaller numbers of moving up like bubbles (to the beginning of the array), this algorithm is named as bubble sort.

Table 10.3: Bubble sort algorithm

1	void bubbleSort(int * A, int size){	void bubbleSort(int [] A){
2	boolean exchanged = true ;	boolean exchanged = true ;
3	while (exchanged){	while (exchanged){
4	exchanged = false ;	exchanged = false ;
5	for (int i = 0; i < size - 1; i++){	for (int i = 0; i < A.length - 1; i++){
6	if (A[i] > A[i + 1]){	if (A[i] > A[i + 1]){
7	exchanged = true ;	exchanged = true ;
8	int tmp = A[i];	int tmp = A[i];
9	A[i] = A[i + 1];	A[i] = A[i + 1];
10	A[i + 1] = tmp;	A[i + 1] = tmp;
11	}	}
12	}	}
13	}	}
14	}	}

Table 10.3 shows the bubble sort algorithm that sorts an array of A $A[0 \dots n - 1]$ consisting n integers. Since the algorithm will stop when there is no swapping, we use a control variable **exchanged** to check if there is a swapping place at each step (Line 3). This variable is initialized to 0 at the beginning of each step (Line 5), if there is a swapping at that step will be set to 1 (Line 8). The process at each step of the bubble sort is the same. We compare 0^{th} element with 1^{th} element, 1^{th} element with 2^{nd} , ..., in short i^{th} element with the $(i + 1)^{th}$ element (Line 7), if the i^{th} element is larger than the $(i + 1)^{th}$ element, these two elements are swapped (Lines 9-11).

Figure 10.3 shows the application of bubble sort on an array of six elements.

- At the first step, 5 and 2, 5 and 4, 6 and 1, 6 and 3 are swapped.
- At the second step, 5 and 1, 5 and 3 are swapped.
- At the third step, 4 and 1, 4 and 3 are swapped.
- At the fourth step, 2 and 1 are swapped.

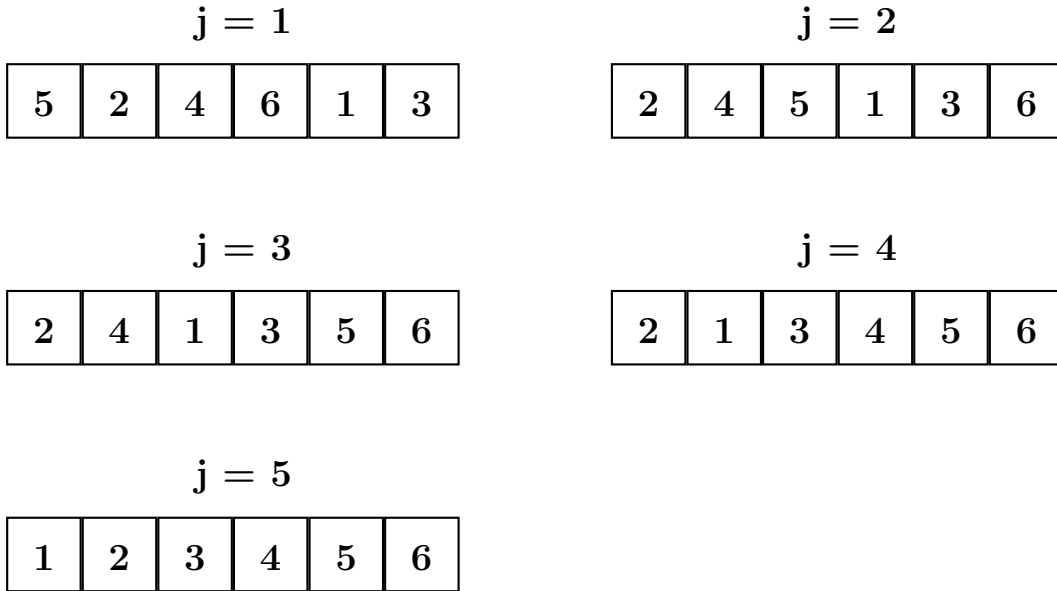


Figure 10.3: Bubble sort of an array of six elements

- At the fifth step, there is no swapping, the array is sorted.

10.4 Shell Sort

Shell sort algorithm is invented by Donald Shell, and its time complexity is smaller than $\mathcal{O}(N^2)$. The sorting algorithms, insertion sort, selection sort, and bubble sort, we have covered in the previous sections, are comparing the adjacent elements of the array and therefore can not be faster than $\mathcal{O}(N^2)$ on the average. Contrary, since Shell sort algorithm compares and swaps distant elements (not adjacent ones), it can be faster than $\mathcal{O}(N^2)$.

Shell sort algorithm works with increment sequence h_1, h_2, \dots, h_k . As long as $h_1 = 1$, all increment sequences are able to work with Shell sort algorithm, but some increment sequences are more successful than the others.

10.4. SHELL SORT

For example, the first increment sequence that was proposed by Shell, namely the sequence $1, 2, 2^2, \dots, 2^k$ has the worst case complexity with Shell sort algorithm as $\Theta(N^2)$, the increment sequence that was proposed by Hibbard, namely the sequence $1, 3, 7, \dots, 2^k - 1$ has the worst case complexity with Shell sort algorithm as $\Theta(N^{\frac{3}{2}})$. The best increment sequences known until today are $9 \times 4^i - 9 \times 2^i + 1$ or $4^i - 3 \times 2^i + 1$, and their worst case complexity is $\mathcal{O}(N^{\frac{4}{3}})$.

At the k^{th} step of the Shell sort algorithm we use the increment h_k , and after step k , for each i , $A[i] \leq A[i + h_k]$. In other words, the elements that have a distance of multiples of h_k to each other are sorted in themselves. The most important property of the Shell sort algorithm is that, the sorted property of the elements after the k^{th} step is not corrupted after the steps $k - 1, k - 2, \dots$

Table 10.4: Shell sorting algorithm

1	void shell(int* A, int size, int* H, int size2)	void shell(int [] A, int [] H){
2	for (int k = 0; k < size2; k++){	for (int k = 0; k < H.length; k++){
3	int increment = H[k];	int increment = H[k];
4	for (int j = increment; j < size; j++)	for (int j = increment; j < A.length; j++){
5	int t = A[j];	int t = A[j];
6	int i = j - increment;	int i = j - increment;
7	while (i >= 0 && A[i] > t){	while (i >= 0 && A[i] > t){
8	A[i + increment] = A[i];	A[i + increment] = A[i];
9	i -= increment;	i -= increment;
10	}	}
11	A[i + increment] = t;	A[i + increment] = t;
12	}	}
13	}	}
14	}	}

Table 10.4 shows the Shell sort algorithm that sorts an array of $A[0 \dots n-1]$ consisting n integers. H represents the increment sequence, $\text{increment} = H[k]$ on the other hand represents the increment used at step k (Line 4). For each increment $H[k]$, the elements $A[H[k]], A[2H[k]], \dots$ are sorted in themselves using the insertion sort algorithm (Line 5-13).

The inner loop scans the elements to be sorted beginning from the $H[k]^{th}$ element (Line 5). The inner while loop on the other hand, continues until either we encounter an element that is smaller than t or we reach at the beginning of the array (Line 8). At each step, the elements are shifted $H[k]$

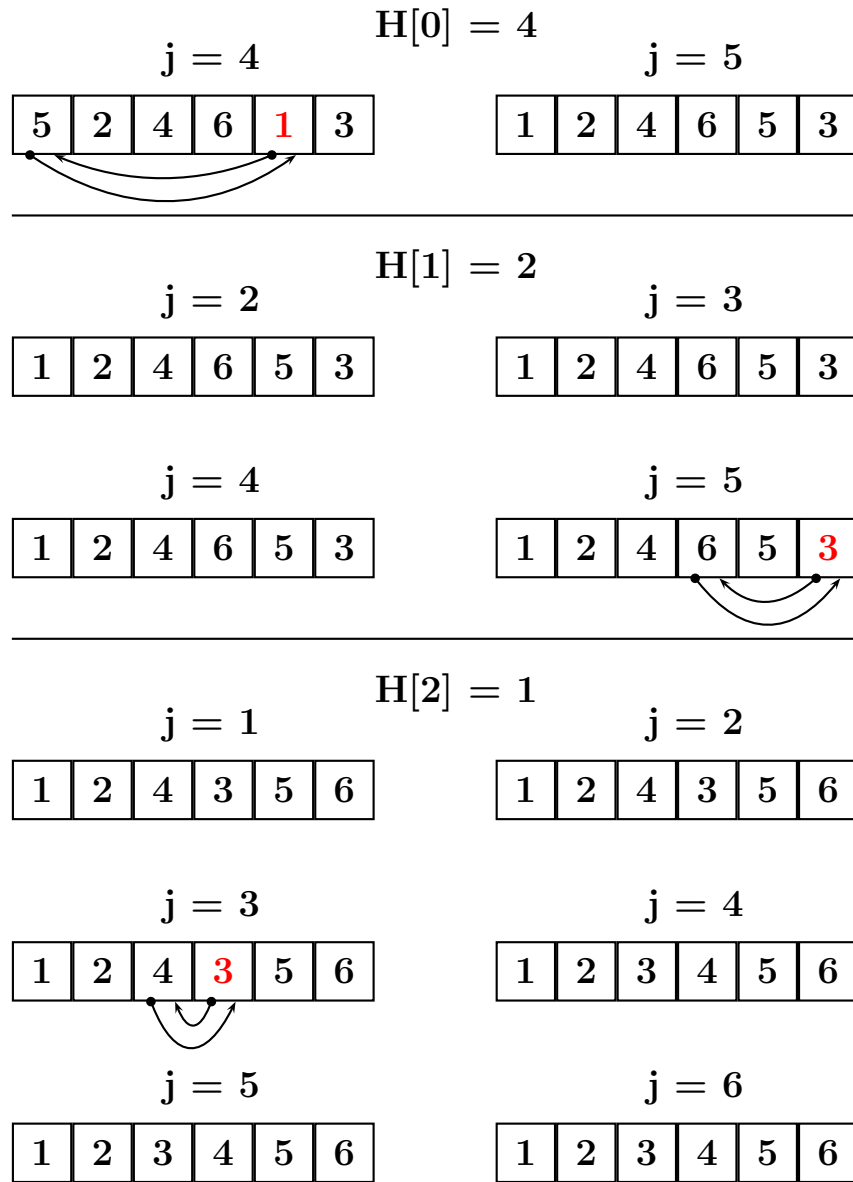


Figure 10.4: Shell sort of an array of six elements

positions right (Line 9). The aim of this shifting is to open a place for t when we find the position to insert t . When we find the position to insert t , it is inserted into that position (Line 12).

Figure 10.4 shows the application of Shell sort to an array of six elements.

- At the first step, the increment is $H[0] = 4$. At this step, the groups with indexes 0-4 and 1-5 are sorted in themselves. There is no need to swap any elements in the group 1-5, whereas in the group 0-4, the elements with indexes 0 and 4 are swapped.
- At the second step, the increment is $H[1] = 2$. At this step, the groups with indexes 0-2-4 and 1-3-5 are sorted in themselves. There is no need to swap any elements in the group 0-2-4, whereas in the group 1-3-5, the elements with indexes 3 and 5 are swapped.
- At the third step, the increment is $H[2] = 1$. At this step, all elements of the array are sorted with insertion sort. But since the array is nearly sorted, when only the elements with indexes 3 and 4 are swapped, the array is sorted.

10.5 Heap Sort

Heap sort approach is a sorting algorithm that uses the heap data structure. Let say, our array **A** is transferred to an heap. If we delete the maximum elements of the heap one by one, we will have the elements of the heap in decreasing order. Table 10.5 shows the heap sort algorithm.

Table 10.5: Heap sort algorithm

<pre> 1 void heapSort(int* A, int size){ 2 HeapNode heapNode; 3 MinHeap heap = MinHeap(size); 4 for (int i = 0; i < size; i++){ 5 heapNode = HeapNode(A[i], i); 6 heap.insert(heapNode); 7 } 8 for (int i = 0; i < size; i++){ 9 heapNode = heap.deleteTop(); 10 A[i] = heapNode.getData(); 11 } 12 }</pre>	<pre> void heapSort(int[] A){ HeapNode heapNode; MinHeap heap = new MinHeap(A.length); for (int i = 0; i < A.length; i++){ heapNode = new HeapNode(A[i], i); heap.insert(heapNode); } for (int i = 0; i < A.length; i++){ heapNode = heap.delete(); A[i] = heapNode.getData(); } }</pre>
---	--

CHAPTER 10. SORTING ALGORITHMS

The algorithm assumes that the elements are already stored in an array before. In the first loop of the algorithm, these elements are taken from the array one by one (Line 6) and added to the heap (Line 8). When an element is inserted into the heap, its structure is rearranged. In the second for loop, by calling function `deleteMax` (Line 11) we place the largest element to the first position, second largest element to the second position, etc. (Line 12). In the function `deleteMax`, we select the maximum element, delete this element and the structure of the heap is rearranged. Thus the next largest element is placed to the root node.

Although the heap sort algorithm works fast, it is not preferred against the quick sort algorithm. The reason for this is, it requires an external structure such as heap. Figures 10.5 and 10.6 show the heap structure corresponding to an array of 10 elements, the formed heap after deleting the maximum element at each step and the last state of the sorted array.

- (a) The first state of the heap is given. The maximum element 16 is selected and placed to the 0^{th} position of the resulting array.
- (b) 16 is removed from the heap, 1 is placed into its position, while 1 is percolating down, 14, 8 and 4 are percolating up. The maximum element 14 is selected and placed to the 1^{th} position of the resulting array.
- (c) 14 is removed from the heap, 1 is placed into its position, while 1 is percolating down, 10 and 9 are percolating up. The maximum element 10 is selected and placed to the 2^{th} position of the resulting array.
- (d) 10 is removed from the heap, 2 is placed into its position, while 2 is percolating down, 9 and 3 are percolating up. The maximum element 9 is selected and placed to the 3^{th} position of the resulting array.
- (e) 9 is removed from the heap, 2 is placed into its position, while 2 is percolating down, 8 and 7 are percolating up. The maximum element 8 is selected and placed to the 4^{th} position of the resulting array.
- (f) 8 is removed from the heap, 1 is placed into its position, while 1 is percolating down, 7 and 4 are percolating up. The maximum element 7 is selected and placed to the 5^{th} position of the resulting array.

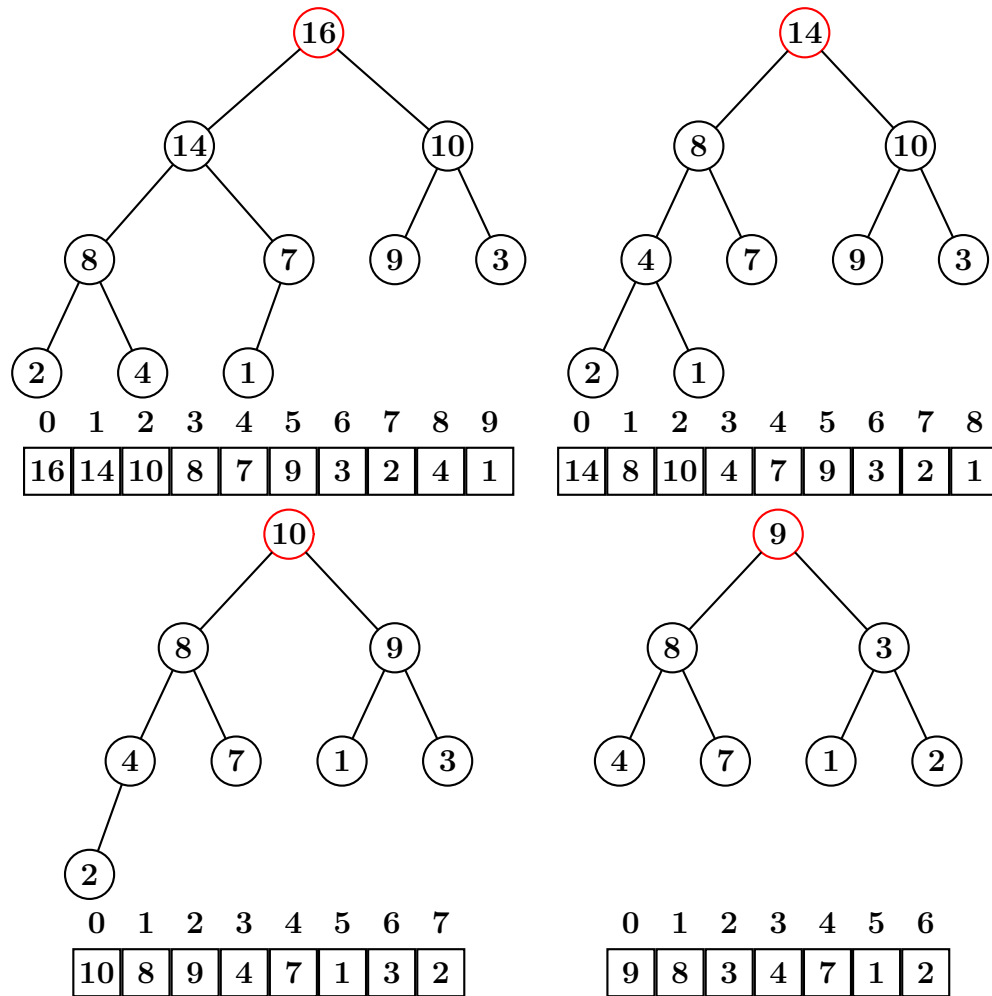


Figure 10.5: Sorting an example array using heap sort (1)

- (g) 7 is removed from the heap, 2 is placed into its position, while 2 is percolating down, 4 is percolating up. The maximum element 4 is selected and placed to the 6th position of the resulting array.
- (h) 4 is removed from the heap, 1 is placed into its position, while 1 is percolating down, 3 is percolating up. The maximum element 3 is selected and placed to the 7th position of the resulting array.

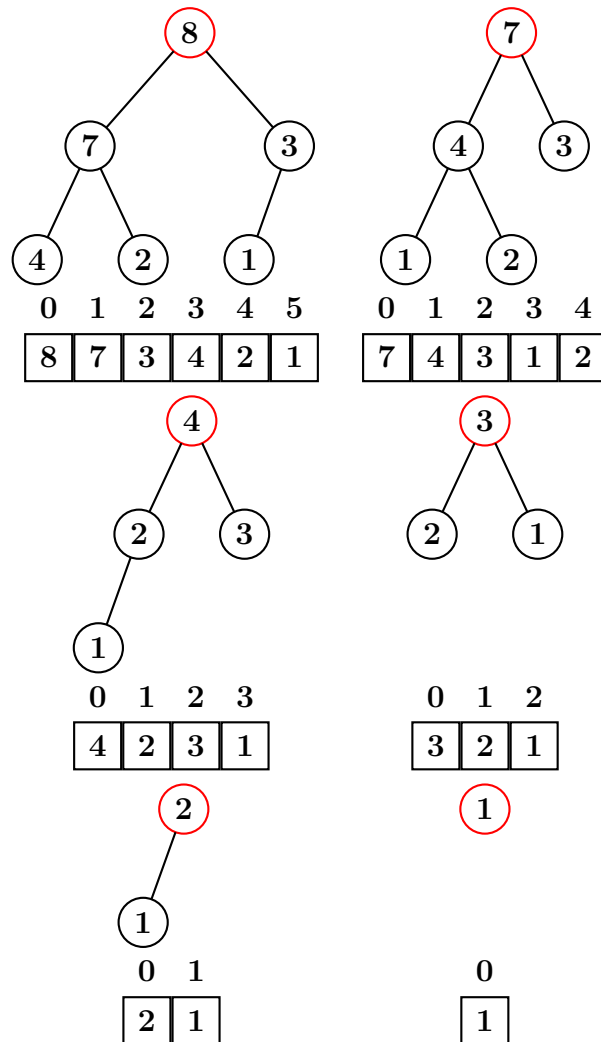


Figure 10.6: Sorting an example array using heap sort (2)

- (i) 3 is removed from the heap, 1 is placed into its position, while 1 is percolating down, 2 is percolating up. The maximum element 2 is selected and placed to the 8th position of the resulting array.
- (j) 2 is removed from the heap, 1 is placed into its position. The maximum element 1 is selected and placed to the 9th position of the resulting array.

10.6 Merge Sort

Merge sort uses divide-and-conquer approach to sort an array. The steps of the algorithm are:

- **Divide** Array consisting of n elements is divided into two subarrays consisting of $\frac{n}{2}$ elements.
- **Conquer** Both subarrays are sorted with merge sort using a recursive function.
- **Merge** Sorted subarrays are merged to produce the main sorted array.

Table 10.7 shows both the function `merge`, which merges two sorted arrays, and the recursive function `mergesort` which calls this `merge` function. `merge` function merges the sorted subarrays $A[p \dots q]$ and $A[q + 1 \dots r]$ and write the result into $A[p \dots r]$. In order not to lose the elements that were previously there, first these elements are copied (Lines 9-12) to two temporary arrays `L` and `R` (Lines 7-8). In order to make the function work easy, we place the largest possible integer to the end of these two temporary arrays (Lines 13-14). Afterwards, the elements of arrays `L` and `R` are compared one by one (Line 17). The variables `i` and `j` are the indexes of the compared elements in their corresponding arrays. If the element in the array `L` is smaller than the element in the array `R` (Line 18), the element in the array `L` is placed to a proper position in the array `A` (Line 19) and the index `i` is incremented by one (Line 20), if the element in the array `R` is smaller than the element in the array `L` (Line 22), the element in the array `R` is placed to a proper position in the array `A` (Line 23) and the index `j` is incremented by one (Line 24).

Figure 10.7 shows merging of two sorted arrays of 4 elements. The elements in red color are the elements which are compared at that step of the algorithm. For example, the first element 2 of the first array is compared with the first element 1 of the second array, the smallest of these two numbers, namely 1, is placed to the first position of array `A`. In the second step, the first element 2 of the first array is compared with the second element of 2 of the second array, since they are the same, 2 is placed as the second element of the array `A`.

Table 10.6: Merge sort algorithm (C++)

```

1 void merge(int* A, int start, int middle, int end){
2     int leftCount = middle - start + 1;
3     int rightCount = end - middle;
4     int* leftPart = new int[leftCount + 1];
5     int* rightPart = new int[rightCount + 1];
6     for (int i = 0; i < leftCount; i++){
7         leftPart[i] = A[start + i];
8     }
9     for (int i = 0; i < rightCount; i++){
10        rightPart[i] = A[middle + i + 1];
11    }
12    leftPart[leftCount] = INT_MAX;
13    rightPart[rightCount] = INT_MAX;
14    int i = 0, j = 0;
15    for (int k = start; k <= end; k++){
16        if (leftPart[i] <= rightPart[j]){
17            A[k] = leftPart[i];
18            i++;
19        } else {
20            A[k] = rightPart[j];
21            j++;
22        }
23    }
24    delete [] leftPart;
25    delete [] rightPart;
26 }
27 void mergeSort(int* A, int first, int last){
28     if (first < last){
29         int pivot = (first + last) / 2;
30         mergeSort(A, first, pivot);
31         mergeSort(A, pivot + 1, last);
32         merge(A, first, pivot, last);
33     }
34 }

```

The more simpler main function `A` sorts the elements of the subarray indexed between `first` and `last`. First, we find the index of the middle element of this subarray `pivot` (Line 31). Then the same function is called by the left (Line 32) and right (Line 33) subarrays of this subarray. If both left or right subarray consist of only one element, the function `mergesort` will not

Table 10.7: Merge sort algorithm (Java)

```

1 void merge(int[] A, int start, int middle, int end){
2     int leftCount = middle - start + 1;
3     int rightCount = end - middle;
4     int[] leftPart = new int[leftCount + 1];
5     int[] rightPart = new int[rightCount + 1];
6     for (int i = 0; i < leftCount; i++){
7         leftPart[i] = A[start + i];
8     }
9     for (int i = 0; i < rightCount; i++){
10        rightPart[i] = A[middle + i + 1];
11    }
12    leftPart[leftCount] = Integer.MAX_VALUE;
13    rightPart[rightCount] = Integer.MAX_VALUE;
14    int i = 0, j = 0;
15    for (int k = start; k <= end; k++){
16        if (leftPart[i] <= rightPart[j]){
17            A[k] = leftPart[i];
18            i++;
19        } else {
20            A[k] = rightPart[j];
21            j++;
22        }
23    }
24 }
25 void mergeSort(int[] A, int first, int last){
26     if (first < last){
27         int pivot = (first + last) / 2;
28         mergeSort(A, first, pivot);
29         mergeSort(A, pivot + 1, last);
30         merge(A, first, pivot, last);
31     }
32 }

```

call itself for these two subarrays. In this case, these two subarrays will be merged by the function `merge` (Line 34). The algorithm progresses for other subarrays the same, by merging the sorted subarrays, the main array will be sorted.

Figure 10.8 shows the application of merge sort on an array of 8 elements. The arrows show which subarrays are merged with which subarrays and therefore produce which subarrays. As can be seen, first subarrays of one

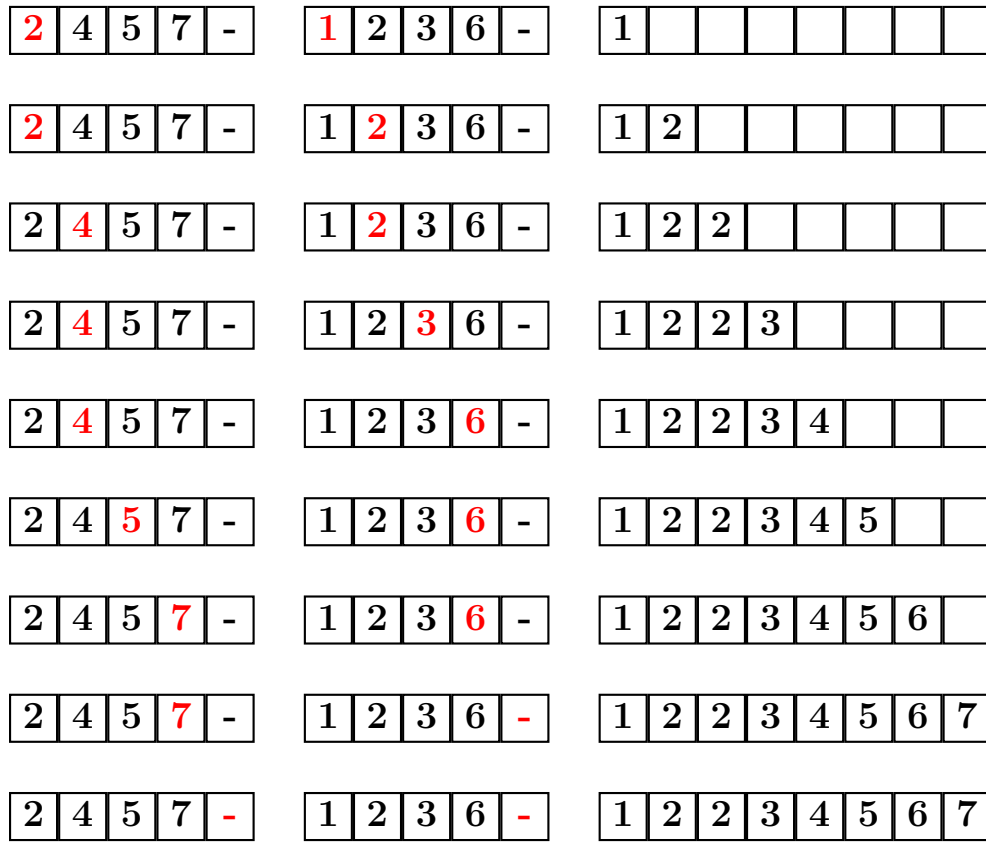


Figure 10.7: Merging two sorted arrays of 4 elements

element are merged to produce subarrays of two elements, subarrays of two elements are merged to produce subarrays of four elements, and last two subarrays of four elements are merged to produce the sorted array.

10.7 Quick Sort

Quick sort is the fastest sorting algorithm known. This sorting algorithm is one of the most beautiful application of divide-and-conquer approach. Let say the subarray of the elements with indexes between p and r of an array A be $A[p \dots r]$. Let's divide this array with respect to the q^{th} element ($p \leq$

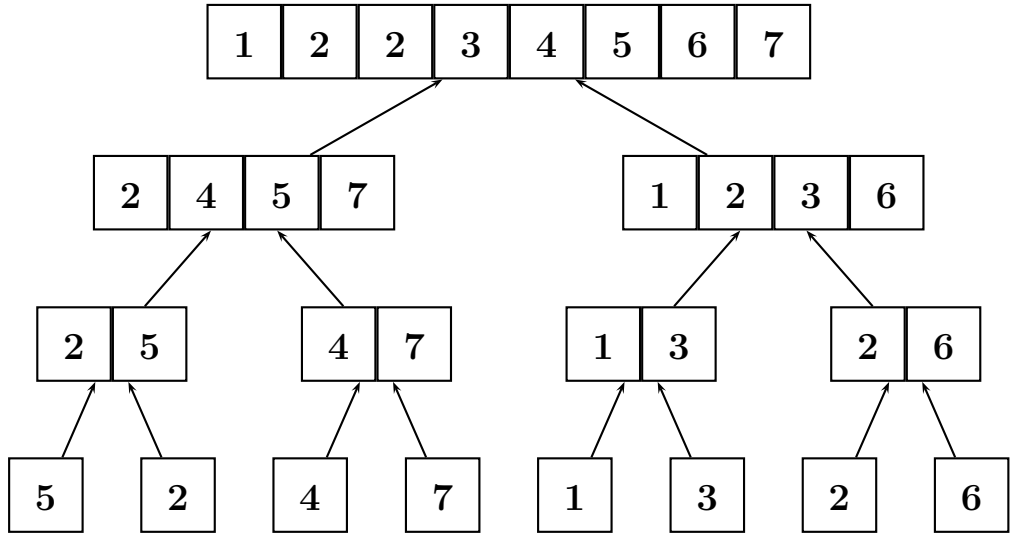


Figure 10.8: Mergesort of an example array of 8 elements

Table 10.8: Quicksort algorithm

<pre> 1 void quickSort(int* A, int first , int last){ 2 if (first < last){ 3 int pivot = partition(A, first , last) 4 quickSort(A, first , pivot - 1); 5 quickSort(A, pivot + 1, last); 6 } 7 }</pre>	<pre> void quickSort(int[] A, int first, int last){ if (first < last){ int pivot = partition(A, first , last); quickSort(A, first , pivot - 1); quickSort(A, pivot + 1, last); } }</pre>
---	--

$q \leq r$) such that each element of the subarray $A[p \dots q - 1]$ be smaller than $A[q]$, and each element of the subarray of $A[q + 1 \dots r]$ be larger than $A[q]$. Afterwards, we need to sort these two subarrays in themselves. With a similar approach, we will divide these two subarrays into two and continue like that until the whole array is sorted. The most important parts of the quick sort are, selecting the pivot element $A[q]$ and dividing the array such that each element on the left hand side will be smaller than $A[q]$ and each element on the right hand side will be larger than $A[q]$. Finding the pivot and dividing the array operations are done using the function **partition**, the remaining quick sort algorithm is given in Table 10.8.

Table 10.9 shows the algorithm to partition the array. Function **partition**

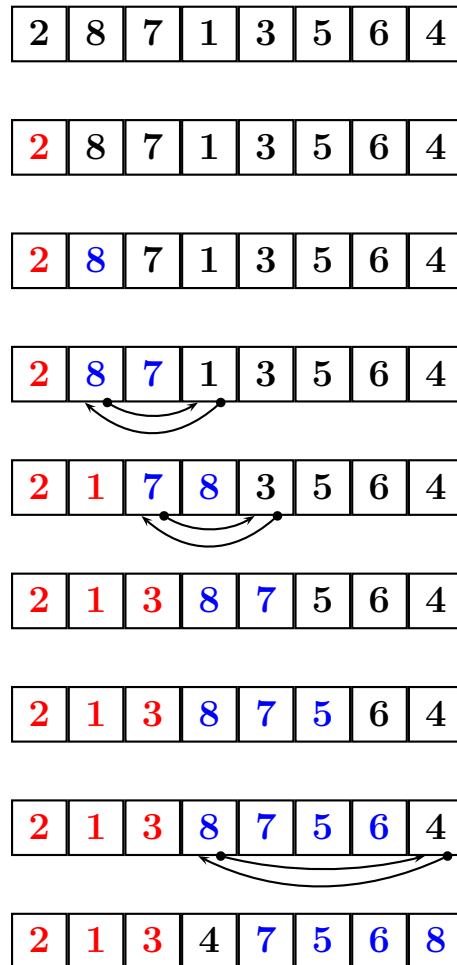
Table 10.9: Partitioning the array

1	int partition(int * A, int first , int last){	int partition(int [] A, int first, int last){
2	int x = A[last];	int x = A[last];
3	int i = first - 1;	int i = first - 1;
4	for (int j = first; j < last; j++){	for (int j = first; j < last; j++){
5	if (A[j] <= x){	if (A[j] <= x){
6	i++;	i++;
7	int tmp = A[i];	int tmp = A[i];
8	A[i] = A[j];	A[i] = A[j];
9	A[j] = tmp;	A[j] = tmp;
10	}	}
11	}	}
12	int tmp = A[i + 1];	int tmp = A[i + 1];
13	A[i + 1] = A[last];	A[i + 1] = A[last];
14	A[last] = tmp;	A[last] = tmp;
15	return i + 1;	return i + 1;
16	}	}

uses always the last element of the subarray as the pivot (Line 8). It also produces two subarrays inside this array. The first subarray starts from the first element and each element of this subarray is smaller than the pivot. The second subarray starts from the end of the first subarray and each element of this subarray is larger than the pivot. These two subarrays grow to the right. Each element is compared with the pivot (Line 11) and if it is smaller than the pivot, it is swapped with the first element of the second array (Line 13). Thus, it will be the last element of the first array. If the current element is larger than the pivot then only the second array is enlarged by one item, thus the current element is now the element of the second array. Note that, if the j^{th} element of the array A is smaller or equal to the pivot then it is swapped with the i^{th} element of the array. The i^{th} element of the array always shows the first element of the largest array. The last operation outside the for loop is to put the pivot between two subarrays. This can be done just by swapping the first element of the second array with the pivot (Line 15).

Figure 10.9 shows an example application of function **partition** on an array.

- The array A is given.
- Value 2 is swapped with itself and put into the subarray containing

Figure 10.9: An example application of the function `partition`

smaller numbers.

- c. Value 8 is put into the subarray containing larger numbers.
- d. Value 7 is put into the subarray containing larger numbers.
- e. Value 1 and value 8 are swapped. The subarray containing smaller numbers enlarges.
- f. Value 3 and value 7 are swapped. The subarray containing smaller

numbers enlarges.

- g. Value 5 is put into the subarray containing larger numbers.
- h. Value 6 is put into the subarray containing larger numbers.
- i. The pivot element is swapped with the first element of the subarray containing larger numbers.

10.8 Bucket Sort

The bucket sort algorithm is different than the previous sorting algorithms we have discussed. Since the previous sorting algorithms depend on the comparison of elements, their best case complexities are $\mathcal{O}(n \log n)$. On the other hand, the time complexity of bucket sort is $\mathcal{O}(n)$. Why not use this algorithm always? Because bucket sort algorithm catches this performance only if the elements to be sorted satisfy some specific property. If this property is not satisfied, bucket sort algorithm can not be applied. Thus, previous sorting algorithms can be categorized in general sorting algorithms, whereas bucket sort algorithm can be categorized in special sorting algorithms. In bucket sort, each data to be sorted is assumed to be an integer smaller than k . If this assumption is correct, we can sort n items in $\mathcal{O}(n)$ time using bucket sort.

The main idea in bucket sort is to find the number of items in the array smaller than a specific number t . If we can find the number of elements that are smaller than t , we will also determine the position of the number t in the sorted array. For example, if there are 8 elements that are smaller than t , t must be on the ninth position in the sorted array. Table 10.10 shows bucket sort algorithm. The algorithm takes array **A** as a parameter and writes the sorted array into the temporary array **B**. While doing sorting, we also make use of another temporary array **C**. The length of this array is k . Here k represents the maximum value that an element can take.

There are four for loops in the bucket sort algorithm.

- With the first for loop (Line 5), if an element of the array **A** takes the value t , then the t^{th} value of the array **C** is incremented by one (Line 6). After this loop, the i^{th} element of the array **C** will show the number of elements in array **A** taking the value i .

Table 10.10: Bucket sort algorithm

<pre> 1 void bucketSort(int* A, int size, int k){ 2 int* C = new int[k]; 3 int* B = new int[size]; 4 for (int i = 0; i < size; i++){ 5 C[A[i]]++; 6 } 7 for (int i = 1; i < k; i++){ 8 C[i] += C[i - 1]; 9 } 10 for (int i = size - 1; i >= 0; i--){ 11 B[C[A[i]] - 1] = A[i]; 12 C[A[i]]--; 13 } 14 for (int i = 0; i < size; i++){ 15 A[i] = B[i]; 16 } 17 delete [] C; 18 delete [] B; 19 }</pre>	<pre> void bucketSort(int[] A, int k){ int[] C = new int[k]; int[] B = new int[A.length]; for (int i = 0; i < A.length; i++){ C[A[i]]++; } for (int i = 1; i < k; i++){ C[i] += C[i - 1]; } for (int i = A.length - 1; i >= 0; i--){ B[C[A[i]] - 1] = A[i]; C[A[i]]--; } for (int i = 0; i < A.length; i++){ A[i] = B[i]; } }</pre>
---	---

- In the second for loop (Line 7), the adjacent elements of the array **C** are summed together (Line 8). After this loop, the i^{th} element of the array **C** will be the sum of all elements before itself and itself. This number will then show how many elements in the array **A** take the value smaller than or equal to i .
- With the third for loop (Line 9), the sorted element are placed on the array **B**. This operation is done with the first line of the loop (Line 10). As we have mentioned before, if there are t numbers that are smaller than i , then the number i must be placed in the position $t+1$. Second line of the for loop is placed for the numbers that take same value. For example, let i be 9 and t be 5. If there is only one number that takes value 9, then that number must be placed in the position $t + 1 = 6$. But if there are more than one number that take value 9, then first 9 will be placed in the 6^{th} position, second 9 will be placed in the 7^{th} position etc. Since $C[i]$ stores the number of integers that take value smaller than or equal to i , if there is only one number taking that value, its position must be $C[i]$. But if there are more than one numbers which

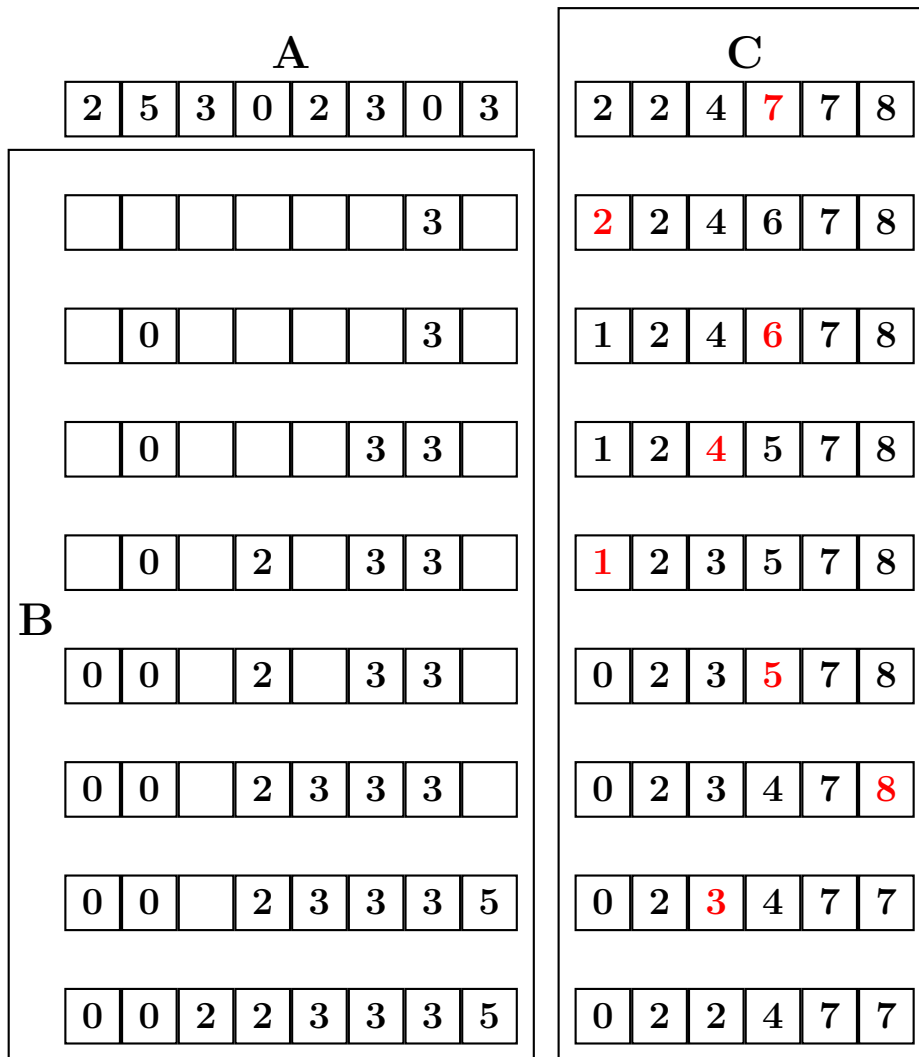


Figure 10.10: Bucket sort of an eight element array A

has a value of i , then the first one must be placed in $C[i]$, second in $C[i] - 1$, etc.

- In the fourth for loop (Line 13), the elements that are temporarily sorted in array B are transferred back to the array A (Line 14).

Figure 10.10 shows the steps applied by bucket sort on an eight element

3	2	9	7	2	0	7	2	0	3	2	9
4	5	7	3	5	5	3	2	9	3	5	5
6	5	7	4	3	6	4	3	6	4	3	6
8	3	9	4	5	7	8	3	9	4	5	7
4	3	6	6	5	7	3	5	5	6	5	7
7	2	0	3	2	9	4	5	7	7	2	9
3	5	5	8	3	9	6	5	7	8	3	9

Figure 10.11: Radix sort on an array of 7 elements

array A. With the first and second for loops, we get first array C which shows how many times a number occur in array A, then the cumulative array C. This array is shown in the beginning of the second column. After that, at each step, starting from the end of the array A, the elements are placed on the appropriate positions of array B. If the value of the element $A[j]$ is i then that element is placed in the position $C[i]$ of array B. After positioning, the value of $C[i]$ is decremented by one.

Since in the bucket sort algorithm each loop is iterated at most n times, the time complexity of this algorithm is $\mathcal{O}(n)$.

10.9 Radix Sort

Radix sort is an algorithm that works on integers and has a worst case complexity of $\mathcal{O}(n)$. As we have explained in Section 10.8, the average case complexity of algorithms those based on comparison of elements can not decrease below $\mathcal{O}(n \log n)$. But the worst case complexity of bucket sort, that was explained in the same section, was $\mathcal{O}(n)$. Similar to the bucket sort, Radix sort algorithm has this best time complexity by making assumptions about the data to be sorted. For example, bucket sort assumes that the data are numbers which are between 1 and k . Similarly, Radix sort assumes that the data consists of only integers. The difference from bucket sort is that the numbers must not be smaller than a certain threshold.

CHAPTER 10. SORTING ALGORITHMS

The radix sort algorithm is very simple. First the numbers are sorted with respect to their least significant digit. Then the numbers are sorted with respect to their second least significant digit. The sorting process continues until all digits of the numbers are consumed. We can use bucket sort in sorting the numbers with respect to their i^{th} significant digit. The digits appearing in any number must be in the interval 0-9.

Figure 10.11 shows the application of the Radix sort algorithm on an array of 7 elements. As can be seen, we first sort the numbers with respect to their ones digit, then to their tenths digit, then to their hundredths digit. The important point in Radix sort is not to change the order of numbers whose sorting digits are the same. For example, in Figure 10.11, the ones digit of 457 and 657 are the same. When we are sorting with respect to their ones digit 457 comes before, therefore at the end of sorting with respect to ones digit 457 must come before 657.

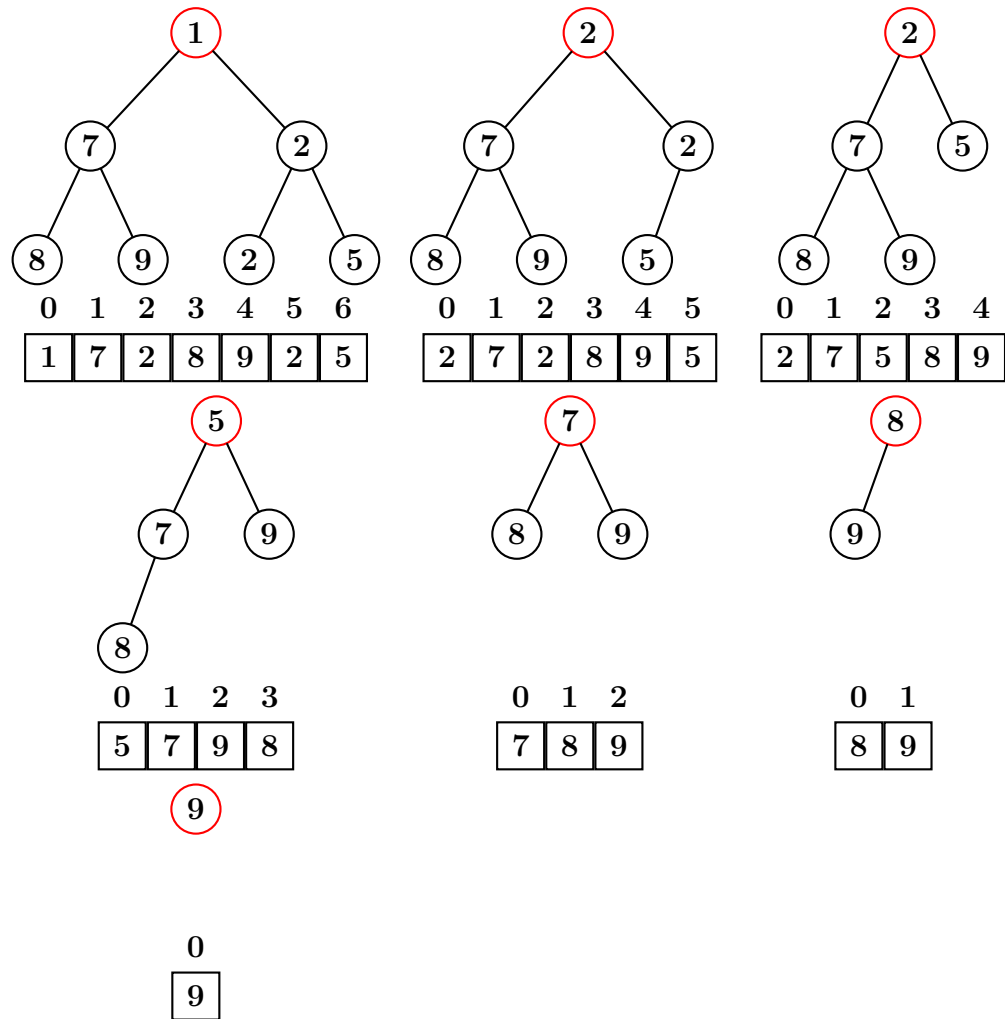
10.10 Notes

The Knuth's book on sorting algorithms is the largest resource on sorting [12]. Heap sort algorithm was proposed by Williams [16]. The inventor of Quick sort is Hoare [7].

10.11 Solved Exercises

1. Sort the sequence 2, 7, 1, 8, 9, 2, 5 using heap sort.

Sorting numbers 2, 7, 1, 8, 9, 2, 5 using heap sort in increasing order is shown below.



2. Write a function that sorts a doubly linked list using selection sort.

void selectionSort (DoublyLinkedList A)

The function that sorts a doubly linked list using selection sort is given below. The external while loop finds the minimum, the second minimum, ..., $n - 1^{th}$ minimum in order and swaps with i^{th} element at each time (Lines 17-19). The internal for loop on the other hand, determines the minimum elements between the elements $A[i]$ and $A[n - 1]$. To do this, first the current minimum is set as $A[i]$ (Line 7). Then each element in this interval is compared with the current minimum (Line

CHAPTER 10. SORTING ALGORITHMS

11). If the compared element is less than the current minimum, current minimum is updated (Line 12) and the position of the minimum element is stored (Line 13). If the i^{th} element is the minimum element (Lines 17), there is no need to swap.

```
1 void selectionSort(DoublyLinkedList A){
2     DoubleNode i, j, k, pos;
3     T min;
4     i = A.first;
5     while (i != null){
6         k = i.next;
7         min = i.data;
8         pos = i;
9         j = i.next;
10        while (j != null){
11            if (j.data < min){
12                min = j.data;
13                pos = j;
14            }
15            j = j.next;
16        }
17        if (pos != i){
18            A.swap(i, pos);
19        }
20        if (pos.next != i)
21            i = k;
22    }
23 }
```

3. What will be the time complexity of insertion sort if the input is (i) sorted (ii) reverse-sorted?

If the input is already sorted in the insertion sort, then for each element $A[j]$, the $A[i] > t$ check of the internal while loop (Line 6) returns false at the first time and the function will not enter into the while loop. In this case, the insertion sort algorithm does 4 operations for each element $A[j]$, and the number of operations will be $4n$ and the time complexity will be $4n \in \mathcal{O}(N)$.

If the input is reverse-sorted in the insertion sort, then for each element $A[j]$, the inner while loop is processed (Line 6) until the check $i \geq 0$ is false and the function iterates in the while loop j times. In this case, the insertion sort algorithm does $2j + 3$ operations for each element

$A[j]$, the the number of operations will be

$$\begin{aligned}
 T(N) &= \sum_{j=1}^N 2j + 3 \\
 &= 2 \sum_{j=1}^N j + 3 \sum_{j=1}^N 1 \\
 &= 2 \frac{N(N+1)}{2} + 3N \\
 &= N^2 + 4N \in \mathcal{O}(N^2)
 \end{aligned}$$

olur.

- Suppose you are given a sorted list of N elements followed by 1 random element. Write a function to sort the entire array.

void addOne(int[] A, int N)

The function that sorts a given sorted list of N elements followed by 1 random element is given below. The position of the n^{th} element that is not sorted is determined like in the insertion sort (Lines 5-8) and the elements right to this position are shifted one position (Line 6). At the end, the unsorted element is inserted into the determined position (Line 9). Since the while loop iterates at most N times, the time complexity of the function will be $\mathcal{O}(N)$.

```

1 void addOne(T[] A, int N){
2     int i;
3     T t = A[N];
4     i = N - 1;
5     while (i >= 0 && A[i] > t){
6         A[i+1] = A[i];
7         i = i - 1;
8     }
9     A[i + 1] = t;
10 }
```

- Given an array of N numbers write a function that determines if there are two numbers whose sum equals a given number K . Your function should run in $\mathcal{O}(N \log N)$ time.

CHAPTER 10. SORTING ALGORITHMS

boolean pairwiseSumK(**int**[] A, **int** N, **int** K)

Given an array of N numbers, the function that determines if there are two numbers whose sum is equal to a given number K in $\mathcal{O}(N \log N)$ time is given below. First the array **A** is sorted using quick sort in $\mathcal{O}(N \log N)$ times (Line 3), then for each element $A[i]$, the number $K - A[i]$ is searched in the remaining part of the array (the elements indexed between $i + 1$ and $N - 1$) using binary search (Line 5). If such an element exists, the sum of $A[i]$ with this element will be K and we will have found two numbers whose sum is K (Lines 6-7). If there are no such two numbers, the function will return 0 (Line 9).

Since the binary search searches in N elements in $\mathcal{O}(\log N)$ time and since we search N elements like this way, the total time complexity will be $\mathcal{O}(N \log N)$.

```
1 boolean pairwiseSumK(int[] A, int N, int K){
2     int i, p;
3     quickSort(A, 0, N - 1);
4     for (i = 0; i < N - 1; i++){
5         p = binarySearch(A, i + 1, N - 1, K - A[i]);
6         if (p != -1)
7             return true;
8     }
9     return false;
10 }
```

10.12 Exercises

1. Sort the sequence 2, 7, 1, 8, 9, 2, 5 using merge sort.
2. Sort the sequence 2, 7, 1, 8, 9, 2, 5 using insertion sort.
3. Write a function that sorts a doubly linked list using insertion sort.

void insertionSort ()

4. Write a function that sorts a doubly linked list using bubble sort.

void bubbleSort()

5. Write a function that sorts a doubly linked list using heap sort.

`void heapSort()`

6. What will be the time complexity of merge sort if the input is (i) sorted (ii) reverse-sorted?

7. A sorting algorithm is stable if equal elements are left in the same order as they occur in the input. Which of the sorting algorithms in this chapter are stable and which are not?

8. Write a function that determines the number of misplaced pairs. Pair (i, j) is misplaced, if i comes before j when $i > j$.

`int misplaced_Pairs(int[] A)`

9. Suppose you are given a sorted list of N elements followed by \sqrt{N} randomly ordered elements. Write a function to sort the entire array.

`void addMore(int[] A)`

10.13 Problems

1. Suppose you have an array of N elements containing only two number 0 and 1. Write a function to rearrange the list so that all 0's precede all 1's. The time complexity of your function should be $\mathcal{O}(N)$.

`void sort01(int[] A)`

2. Write a function that sorts an array of N elements with respect to their last digits. The time complexity of your function should be $\mathcal{O}(N)$.

`void sortLastDigit(int[] A)`

3. Suppose arrays A and B are both sorted. Write a function that works in linear time and finds $A \cup B$.

`int[] union(int[] A, int[] B)`

4. Suppose arrays A and B are both sorted. Write a function that works in linear time and finds $A \cap B$.

`int[] intersection(int[] A, int[] B)`

CHAPTER 10. SORTING ALGORITHMS

5. Given an array of N numbers write a function that determines the nearest two numbers. Your function should run in $\mathcal{O}(N \log N)$ time. (Hint: Sort the numbers first)
6. Given an array of N numbers write a function that determines if there are four numbers whose sum equals a given number K . Your function should run in $\mathcal{O}(N^2 \log N)$ time.

boolean existsFour(**int** [] A, **int** K)

7. Suppose you are given two linked lists of sorted integers. Write an $\mathcal{O}(N)$ algorithm that merges these two linked lists such that the resulting linked list is also sorted.

LinkedList merge(**LinkedList** A, **LinkedList** B)

8. Suppose you are given two sorted arrays A and B. Write a function that finds elements in A / B (the elements that are in A but not in B) in $\mathcal{O}(N)$ time.

int [] difference (**int** [] A, **int** [] B)

9. Suppose you are given an array of N integers containing the birth years of students. Write an $\mathcal{O}(N)$ algorithm to sort these birth years.

void sortBirthYears (**int** [] A)

10. Suppose you are given a linked list of N integers to be sorted. Write an $\mathcal{O}(N)$ algorithm that checks if the linked list is already sorted.

boolean isSorted(**LinkedList** A)

11. Suppose you are given two sorted arrays A and B. Write a function that finds elements in $A \triangle B = (A - B) \cup (B - A)$ (the elements that are in A but not in B and the elements that are in B but not in A) in $\mathcal{O}(N)$ time.

int [] symmetric(**int** [] A, **int** [] B)

12. Suppose you are given an array of N integers. Write an $\mathcal{O}(N \log N)$ algorithm that find the minimum difference between any two elements in this array.

int minDifference(**int** [] A)

10.13. PROBLEMS

13. Suppose you are given an unsorted array of N integers and two numbers X and Y (Assume $X < Y$). Write an $\mathcal{O}(N)$ algorithm to partition the numbers in the array such that, the numbers that are smaller than X will be in the first part, the numbers that are larger than X but smaller than Y will be in the second part, and the numbers that are larger than Y will be in the third part.

```
void partition (int [] A, int X, int Y)
```

14. Suppose you are given two sorted arrays A and B . Write a function that returns the number of elements which are in A but not in B .

```
int inANotB(int[] A, int [] B)
```

15. Suppose you are given three sorted arrays A , B , and C . Write a function that returns the number of elements which are in A , B , and C .

```
int inABC(int[] A, int [] B, int [] C)
```

16. Write a function that sorts an array of N elements with respect to their first digits. The time complexity of your function should be $\mathcal{O}(N)$.

```
void sortFirstDigit (int* A, int N)
```

17. Suppose arrays representing sets A and B are both sorted. Write a linear time method that finds if A is a subset of B .

```
boolean isSubset(int [] A, int [] B)
```

18. Suppose arrays representing sets A and B are both sorted. Write a linear time method that finds the minimum difference between any element from A and any element from B .

```
int minDifference(int [] A, int [] B)
```

19. Write a method which returns the sorted form of the linked list (as a new linked list), which contains only numbers 1, 2, and 3. Your algorithm should run in linear time $\mathcal{O}(N)$.

```
LinkedList sortLinkedList (LinkedList list )
```

20. Suppose arrays representing sets A and B are both sorted. Write a method with **linear complexity** in **MergeSort** class that finds if C is the intersection of sets A and B . You are only allowed to use 1 loop.

```
bool isIntersection (int* A, int* B, int* C, int sizeA, int sizeB, int sizeC)
```

CHAPTER 10. SORTING ALGORITHMS

21. Modify the original insertion sort so that

```
void insertionSort (LinkedList l)
```

it will use the same algorithm but sorts the elements in the linked list *l*. You can only use `getPrevious` method as an external method except getters and setters.

22. Suppose arrays representing sets A and B are both sorted. Write a method with **linear complexity** in **MergeSort** class that finds if A is a superset of B.

```
bool isSuperSet(int* A, int* B, int sizeA, int sizeB)
```

23. Modify the original selection sort so that

```
int* sortNew(int* A, int* B, int size)
```

(i) it will return the original indexes of the elements as an array and
(ii) uses B as a secondary key when keys in A are equal. If original list is 20, 10, 40, 30; after sorting A will be 10, 20, 30, 40, and it will return index array as 1, 0, 3, 2 (10 was at 1., 20 was at 0., 30 was at 3., 40 was at 2. position in the beginning).

24. Modify the original bubble sort such that

```
void sort2(int* A, int* B, int* C, int N)
```

uses B as a secondary and C as a ternary key. If two elements in A are equal, algorithm uses B as a secondary key. If also their keys in B are equal, the algorithm will then resort to C array. Do not modify arrays B and C. N is the number of elements in the array. Your modified method should run in $\mathcal{O}(N^2)$ time.

25. Implement a modified version of partition algorithm to write a method in **QuickSort** class

```
void oddsBeforeEvens(int* A, int N)
```

which moves the odd numbers before even ones in A in one pass. The algorithm is as follows:

1. At the beginning of the algorithm, let say we have two indexes *i* and *j*, showing the first and last elements respectively.
2. While *i* is less than *j*,

- Increment i until you find an odd number.
 - Decrement j until you find an even number.
 - If $i < j$, swap the contents of i and j .
3. Continue with step 2.
26. Given an array of N numbers write a function that determines the maximum length of the sorted sublist in the array. A sublist start from index i and continues with indexes $i+1, i+2, \dots, i+k$. Your function should run in $\mathcal{O}(N)$ time.

```
int maxSortLength(int* A, int N)
```

27. Suppose you are given three sorted arrays A, B, and C. Write a function in MergeSort class that returns the number of elements which are in A or B or C. Assume that all arrays have the same size and the last elements of all three arrays are the same. Your algorithm should run in $\mathcal{O}(N)$ time.

```
int inAorBorC(const int* A, const int* B, const int* C, int N)
```

28. Suppose you are given an array of N integers. Write a single pass algorithm in QuickSort class that puts the odd integers before the even integers similar to the partition algorithm in QuickSort.

```
void oddsBeforeEvens(int* A, int N)
```

29. Suppose you are given A, an unsorted array of N integers, and a number X . Write $\mathcal{O}(N)$ algorithm

```
void threePartitionArray (int* A, int N, int X)
```

to partition the numbers in the array such that, the numbers that are smaller than X will be in the first part, the numbers that are equal to X will be in the second part, and the numbers that are larger than X will be in the third part. You may not use any additional data structure or array.

Bibliography

- [1] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet. Mat. Doklady*, 3:1259–1263, 1962.
- [2] R. E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.
- [3] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [4] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345, 1962.
- [5] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7:701, 1964.
- [6] L. J. Guibas and E. Szemerédi. The analysis of double hashing. *Journal of Computer and System Sciences*, 16:226–274, 1978.
- [7] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.
- [8] J. E. Hopcroft and R. E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16:372–378, 1973.
- [9] D. E. Knuth. Big omicron and big omega and big theta. *ACM SIGACT News*, 8:18–23, 1976.

BIBLIOGRAPHY

- [10] D. E. Knuth. *The Art of Computer Programming, Vol 1: Fundamental Algorithms*. Addison-Wesley, 1997.
- [11] D. E. Knuth. *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*. Addison-Wesley, 1998.
- [12] D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching*. Addison-Wesley, 1998.
- [13] W. W. Peterson. Addressing for random access storage. *IBM Journal of Research and Development*, 1:130–146, 1957.
- [14] R. E. Tarjan. Depth first search and linear graph algorithms. *Siam Journal on Computing*, 1:146–170, 1972.
- [15] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9:11–12, 1962.
- [16] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.