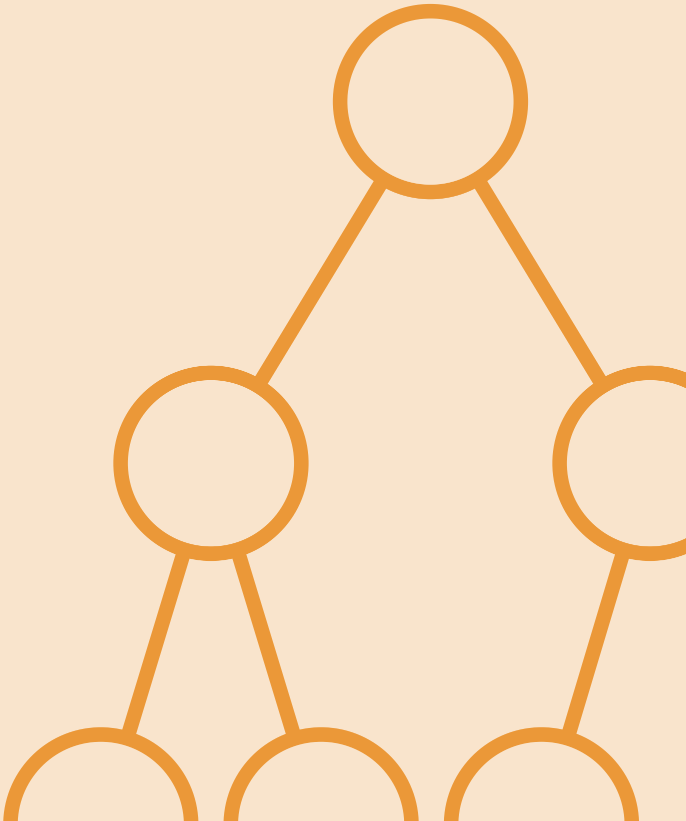
Data Structures and Analysis



Copyright © 2021 Kevin Gao

TERRA-INCOGNITA.DEV

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at `http://creativecommons.org/licenses/by-nc/3.0`. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Contents

I Data Structures

1 Abstract Data Types 13
1.1 Abstract Data Type 13
1.2 Data Structures 13
1.3 Algorithm Compleixty 13
1.4 Dictionary ADT and Implementations 14
1.4.1 Dictionary ADT 14
1.4.2 Data Structures for Dictionary 14

2 Binary Search Trees 19
2.1 Binary Search Tree 19
2.2 Searching in BST 19
2.3 Insertion 19
2.4 Deletion 20

3 Balanced Search Trees 21
3.1 Balanced Trees 21
3.2 Red-Black Tree 21
3.2.1 Definition and Properties 21

3.3 Insertion and Deletion in Red-Black Tree 23
3.3.1 Rotation Operations 23
3.3.2 Insertion 23
3.3.3 Deletion 25
3.4 AVL Tree 26
3.4.1 AVL Insertion 27
3.4.2 AVL Deletion 27
3.4.3 AVL Rotation Recipes 27
3.5 B-Tree 28
3.5.1 2-3 Tree 28

4 Augmenting Data Structures 29
4.1 Augmenting Data Structures 29
4.2 Order Statistics With Red-Black Trees 29
4.2.1 The RANK Operation 29
4.2.2 Maintaining the Size Property at Each Node 30
4.2.3 The SELECT Operation 30
4.3 Steps To Create Augmented Data Structures 31
4.4 Intervals ADT 31

5 Priority Queue and Heap 33
5.1 Priority Queue 33
5.1.1 Priority Queue ADT 33
5.1.2 Primitive Implementation Using Linked Lists 33
5.2 Heap 34
5.2.1 Types of Binary Trees 34
5.2.2 Heap Property 35
5.3 Maintaining the Heap Property 35
5.3.1 Correctness of MAX-HEAPIFY 36
5.3.2 Running Time of MAX-HEAPIFY 36
5.4 Inserting Into Max-Heap 36
5.5 Build Heap From Unsorted Array 36
5.5.1 Running Time of BUILD-MAX-HEAP 36
5.6 Heapsort 37

II Algorithm Analysis

6 Average Case Complexity and Randomized Algorithms 41
6.1 Basic Probability Theory 41
6.1.1 Sample Space and Events 41
6.1.2 Properties of Probability Functions 41

6.2 Conditional Probability and Independence 42
6.3 Average Case Analysis 42
6.4 Average Case Analysis of Linear Search 43
6.5 Average Case Analysis of Quick Sort 44
6.6 Randomized Quicksort 46
6.7 Randomized Selection 47

7 Randomness 49
7.1 Random Permutations of an Array 49
7.2 The Hiring Problem 50
7.3 Reservoir Sampling 51

8 Amortized Analysis 53
8.1 Amortized Cost 53
8.1.1 Amortized Analysis of Sorted Linked List 54
8.1.2 Increasing Binary Counter 54
8.2 Aggregate Method 56
8.3 Accounting Method 56
8.4 Potential Method 58
8.4.1 Analysis of Binary Counter Using Potential Method 58
8.5 Amortized Analysis of Stack with Multipop 59
8.5.1 Amortized Analysis of Multi-Pop Using Accounting Method 59

9 Dynamic Arrays 61
9.1 Dynamic Array 61
9.2 Amortized Analysis of Dynamic Array Using Accounting Method 61
9.3 Amortized Analysis of Dynamic Array Using Potential Method 62
9.4 Deletion From Dynamic Array 63
9.4.1 Amortized Analysis of Dynamic Array With Deletion 64
9.5 Amortized Analysis of Dynamic Array With Delete Using Potential Method 66

10 Scapegoat Tree and Splay Tree 67
10.1 Scapegoat Tree 67
10.2 Amortized Analysis of Scapegoat Tree 67

III Hashing

11 Hashing 71
11.1 Hashing and Hash Function 71

11.2 Resolving Collision 71
11.3 Universal Hashing 72
11.4 Analysis of Hashing with Chaining 73
11.5 Perfect Hashing 74
11.5.1 Constructing a Perfect Hash Functions 74
11.5.2 FKS Hashing 76
11.5.3 Representing FKS Hash Table 77
11.6 Application of Hashing 79

12 Open Addressing 81
12.1 Open Addressing 81
12.1.1 Linear Probing 81
12.1.2 Analysis of Linear Probing 82
12.2 Quadratic Probing 82
12.3 Double Hashing 82
12.4 Deletion in Open Addressing 82

13 Bloom Filter 85

IV Advanced Data Structures

14 Binomial Heaps 89
14.1 Mergeable Priority Queue ADT 89
14.2 Binomial Heap 89
14.3 Operations on Binomial Heap 91
14.3.1 Linking Two Binomial Trees 91
14.3.2 Union Two Binomial Heaps 91
14.3.3 Insertion 92
14.3.4 Getting the Minimum 93
14.3.5 Extract Min 93
14.3.6 Decreasing Priority 93

15 Fibonacci Heaps 95

16 Disjoint Sets 97
16.1 Disjoint Set ADT 97
16.2 Data Structures for Disjoint Sets 97
16.2.1 Circular Linked List 98
16.2.2 Linked List With Back Pointers 99
16.2.3 Linked List With Back Pointers Using Union By Weight 99
16.2.4 Trees 100
16.2.5 Trees With Union By Height 100

16.2.6 Trees With Path Compression 101
16.2.7 Additional Implementations 101
16.2.8 Trees With Union By Rank and Path Compression 101
16.3 Amortized Analysis of Disjoint Set 102
16.3.1 An Alternative Proof 105

V Lower Bounds

17 Decision Trees 109
17.1 Lower Bounds 109
17.2 Comparison Model 109
17.3 Comparison Tree 110
17.3.1 Intuition 110
17.3.2 Decision Tree for Searching 110

18 Information Theory 113
18.1 Height of Trees 113
18.2 Lower Bounds for Searching 113
18.2.1 Searching Sorted List 113
18.2.2 Searching Unsorted List 114
18.3 Sorting In Linear Time 114

19 Adversary Arguments 115
19.1 Guess a Number Game 115
19.1.1 Battleship 115
19.2 Proving Problem Lower Bound Using Adversary Argument 115

20 Reduction 117

VI Graphs

21 Graph Representation 121
21.1 Graph 121
21.1.1 Other Types of Graphs 121
21.2 Operations on Graphs 121
21.3 Data Structures for Representing Graph 122
21.3.1 List of Edges 122
21.3.2 Adjacency Matrix 122
21.3.3 Adjacency List 122

21.4 Incidence Matrix 123
21.5 Time Complexity 123

22 Breadth-First Search 125
22.1 Breadth-First Search (BFS) 125
22.1.1 BFS Algorithm 125
22.1.2 Edge Classification 126
22.1.3 Correctness of BFS 126
22.1.4 Running Time of BFS 127

23 Depth-First Search 129
23.1 Depth-First Search (DFS) 129
23.1.1 DFS Algorithms 129
23.1.2 Edge Classification 130
23.2 Applications of BFS/DFS 131
23.2.1 Cycle Detection 131
23.2.2 Determining If a Graph is Bipartite 131
23.2.3 Topological Sort 131

24 Minimum Spanning Trees 133
24.1 Spanning Forest and Spanning Tree 133
24.2 Computing the Minimum Spanning Tree 133
24.3 Kruskal's Algorithms 135
24.4 Prim's Algorithm 135

25 Single-Source Shortest Path Algorithms 139
25.1 Shortest Path 139
25.2 Dijkstra's Algorithm 139

Appendix

Axioms & Theorems 143

Basic Prerequisite Mathematics 147

Proof Templates 153

Index 163

Bibliography 167
Courses 167
Books 167

Data Structures

1 Abstract Data Types 13
1.1 Abstract Data Type
1.2 Data Structures
1.3 Algorithm Compleixty
1.4 Dictionary ADT and Implementations

2 Binary Search Trees 19
2.1 Binary Search Tree
2.2 Searching in BST
2.3 Insertion
2.4 Deletion

3 Balanced Search Trees 21
3.1 Balanced Trees
3.2 Red-Black Tree
3.3 Insertion and Deletion in Red-Black Tree
3.4 AVL Tree
3.5 B-Tree

4 Augmenting Data Structures 29
4.1 Augmenting Data Structures
4.2 Order Statistics With Red-Black Trees
4.3 Steps To Create Augmented Data Structures
4.4 Intervals ADT

5 Priority Queue and Heap 33
5.1 Priority Queue
5.2 Heap
5.3 Maintaining the Heap Property
5.4 Inserting Into Max-Heap
5.5 Build Heap From Unsorted Array
5.6 Heapsort

Chapter 1 Abstract Data Types

1.1 Abstract Data Type

Definition 1.1.1 — Abstract Data Type. An abstract data type (ADT) is a set of mathematical objects and a set of operations on those objects. An ADT describes how information can be used in a program, which is important for specification and provides modularity and reuseability.

■ **Example 1.1 — ADT for Integers.**

- Objects: $\mathbb{Z}$
- Operations: $\textsc{Add}(x,y)$ $\textsc{Substract}(x,y)$, $\textsc{Multiply}(x,y)$, $\textsc{Quotient}(x,y)$, and $\textsc{Remainder}(x,y)$.

■

■ **Example 1.2 — Stack ADT.**

- Objects: sequences
- Operations: $\textsc{Push}(S)$ $\textsc{Pop}(S)$, $\textsc{Empty}(S)$.

■

1.2 Data Structures

Definition 1.2.1 — Data Structure. A data structure is an implementation of an abstract data type.

■ **Example 1.3 — Data Structures for Stack.** A data structure for stacks is an array with a counter. Alternatively, a stack can be implemented as a singly linked list with the top of the stack at the beginning of the list. ■

An ADT specifies what kind of data you can have and what you can do with the data. A data structure specifies how the data is implemented; in other words, it specifies how the data is stored and how you actually do the operations on the data.

1.3 Algorithm Compleixty

The complexity of an algorithm tells us the amount of resources used by the algorithm, expressed by a function of the size of the input. Such resources can be time, space, number of messages, numbers of bits of communication, etc. We are interested in analyzing the complexity of algorithms because it allows us to:

- compare different algorithms
- give bound to resources needed for a given input
- determine the largest size of input for which the algorithm is still efficient

The definition of input size depends on the problem that we are interested in. Below are examples of some common definitions of input size for the type of data that we are dealing with.

- integer: number of bits
- list: number of elements
- array: dimension of the array, or number of bits
- graph: number of vertices, or number of edges, or both

1.4 Dictionary ADT and Implementations

1.4.1 Dictionary ADT

In this seciton, we will take a look at an example of ADT and some implementation of it. For the dictionary ADT, the objects are defined to be the set of elements each of which has a key drawn from a totally ordered set. And the dictionary ADT should support the following operations:

- SEARCH(S,k): search the set S for an element with the key k and return a pointer to one such element. If no such element exists, return NIL.
- INSERT(S,x): insert element pointed by the pointer x into the set S.
- DELTE(S,x): delete the element pointed to by the pointer x from the set S.

Each element x will contain a property k that stores the key.

R Some examples of totally ordered sets are: $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, colors, English words. The set of complex numbers $\mathbb{C}$ is not totally ordered. A more formal definition of a total order can be found in the notes on Theory of Computation (CSC 240/CSC 236).

1.4.2 Data Structures for Dictionary

There are many ways to implement a dictionary. The simplest and most common way is to use a hash table, but there are also equally valid implementations.

Hashing

SEARCH: average complexity $O(1)$, worst-case complexity $O(n)$
INSERT: average complexity $O(1)$, worst-case complexity $O(n)$

DELETE: average complexity $O(1)$, worst-case complexity $O(n)$

Array

We can use two arrays, one with keys, the other with values in the correesponding position of the keys. In the case of unsorted arrays, the time complexities of the operations are:

SEARCH: worst-case complexity $O(n)$; since unsorted, we need to perform linear search to find the element
INSERT: worst-case complexity $O(1)$
DELETE: worst-case complexity $O(1)$

Another assumption that we need to make is that the elements and keys are placed consecutively in the array. However, empty slots might be created upon deleting elements. To solve this, we simply replace the deleted element with the last element in the array. By doing so, we ensure that the number of elements in S is at most the size of the array.

Binary Search Tree

SEARCH: $\Theta(h)$ INSERT: $\Theta(h)$
DELETE: $\Theta(h)$
where h is the height of the tree.

Sorted Array with Counter

SEARCH: $O(\log n)$ using binary search
INSERT: $\Theta(n)$
DELETE: $\Theta(n)$

Unsorted Singly Linked List

SEARCH: $\Theta(n)$
INSERT: $\Theta(1)$
DELETE: $\Theta(n)$; this is because for a singly linked list, it takes $\Theta(n)$ time to find the pointer to the previous element in order to reconnect the links after deleting

Unsorted Doubly Linked List

SEARCH: $\Theta(n)$
INSERT: $\Theta(1)$
DELETE: $\Theta(1)$

Sorted Doubly Linked List

SEARCH: $\Theta(n)$; we cannot perform binary search because we don't know the length of the array
INSERT: $\Theta(n)$ to insert into the correct position to keep the list sorted
DELETE: $\Theta(1)$

Direct Access Table

If our set of keys S is a subset of some finite universe $U = \{0, 1, \ldots, m-1\}$ where $|U| = m$, then we can use a direct access table to implement a dictionary ADT. To represent the dictionary, we use an array A with m slots indexed from 0 to m, each of which corresponds to a key in S. The value of the slot $A[i]$ is the pointer to the element in the set S with the key i. The limitations of such implementation include:

- keys have to be unique
- size of the list can get arbitrarily large
- size of the list is limited

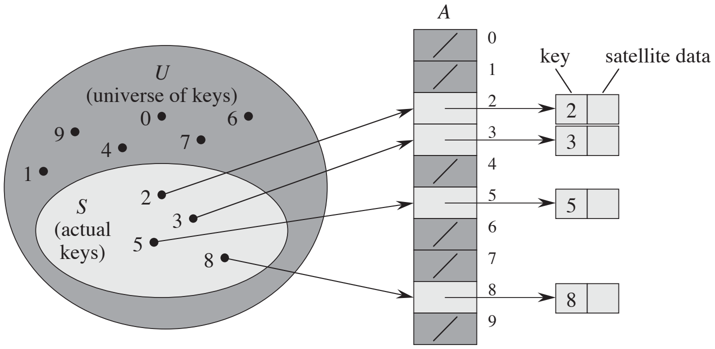

A
U
(universe of keys)
S
(actual
keys)
0
1
2
3
4
5
6
7
8
9
key
satellite data
2
3
5
8

Figure 1.1: Direct access table

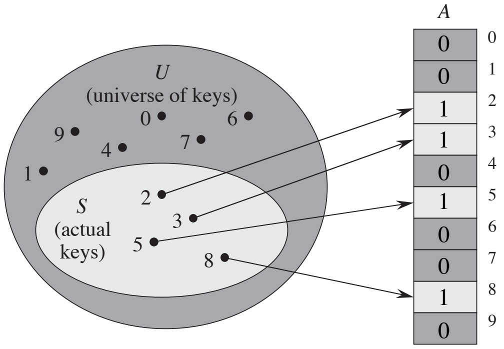

A
U
(universe of keys)
S
(actual
keys)
0
1
2
3
4
5
6
7
8
9
0
0
1
1
0
1
0
0
1
0

Figure 1.2: Bit vector direct access table

```
SEARCH($A, k$)
return $A[k]$
```

```
INSERT($A, x$)
    $A[x.key] = x$
```

```
DELETE($A, x$)
    $A[x.key] = $ NIL
```

Each of the three operations above takes constant time $O(1)$.

Alternatively, we can use value 1 to indicate the presence of element at a given slot, and value 0 to denote the absence of element at a given slot. The resulting data structure called a bit vector direct access table.

Chapter 2 Binary Search Trees

2.1 Binary Search Tree

Definition 2.1.1 — Binary Search Property. If y is in the left subtree of x, then $y.key \leq x.key$. If y is in the right subtree of x, hen $y.key \geq x.key$.

Definition 2.1.2 — Binary Search Tree. A binary tree is a binary search tree if all nodes x, y of the tree satisfies the binary search property.

If we perform an in-order traversal of a binary search tree, we gets the nodes in increasing order by key.

2.2 Searching in BST

```
SEARCH$(T, k)$
if $T =$ NIL
    return NIL
if $T.key == k$
    return $T$
if $T.key > k$
    return SEARCH$(T.left, k)$
if $T.key < k$
    return SEARCH$(T.right, k)$
```

2.3 Insertion

```
INSERT$(T, x)$
$X =$ SEARCH$(T, x)$
if $X ==$ NIL
    $X = x$
else
    // there is already an element with key $x.k$ stored in BST, we can do the following
    // 1. return without inserting $x$
    // 2. continues to search until NIL is found
    // 3. store all elements with the same key in an auxillary data structure (e.g. linked list)
    // 4. replace old element with $x$
```

2.4 Deletion

Deletion is the trickiest operation for BST.

```
DELETE$(T, x)$
$y = x.left$
while $y.right \neq$ NIL
    $y = y.right$
$x = y$
$y =$ NIL
```

Chapter 3 Balanced Search Trees

3.1 Balanced Trees

There are two different ways of defining balancedness of a binary tree: weight balance and height balance. In this chapter, we will mainly focus on height balanced trees. As it turns out, weight balance is a more strict requirement than height balance, and weight balance implies height balance. Since the runtime complexity of binaryy tree operations are height-dependent, both definitions should give us $O(\lg n)$ time on most operations.

Definition 3.1.1 — Height Balancedness. A binary tree is height balanced if for every node in the tree, the height of its left and right subtrees differ by at most one.

Definition 3.1.2 — Weight Balancedness. A binary tree is weight balanced if for every node in the tree, the number of nodes of its left and right subtrees differ by at most one.

Corollary 3.1.1 Weight-balanced binary trees are height-balanced.

In this section we will look at a few height balanced search tree including red-black trees, AVL (Adelson-Velskii and Landis) trees, 2-3 trees, and B-trees which is a more general form of 2-3 trees. The first two are binary trees while 2-3 tree and B-tree are not necessarily binary.

3.2 Red-Black Tree

3.2.1 Definition and Properties

Definition 3.2.1 — Red-Black Tree. A red-black tree is a binary search tree in which every node is either red or black and satisfies the following properties:

1. The root is black
2. Every leaf node (NIL node) is black
3. If a node is red, then both its children are black
4. For each node, all paths from the node to descendant leaves (NIL nodes) contain the same number of black nodes

Alternatively, the properties can be stated without using the NIL node.

1. The root is black
2. A red node has no red children
3. Every path from the root to a node with at most one child contains the same number of black nodes

Figure 3.1 illustrates the properties of red-black trees.

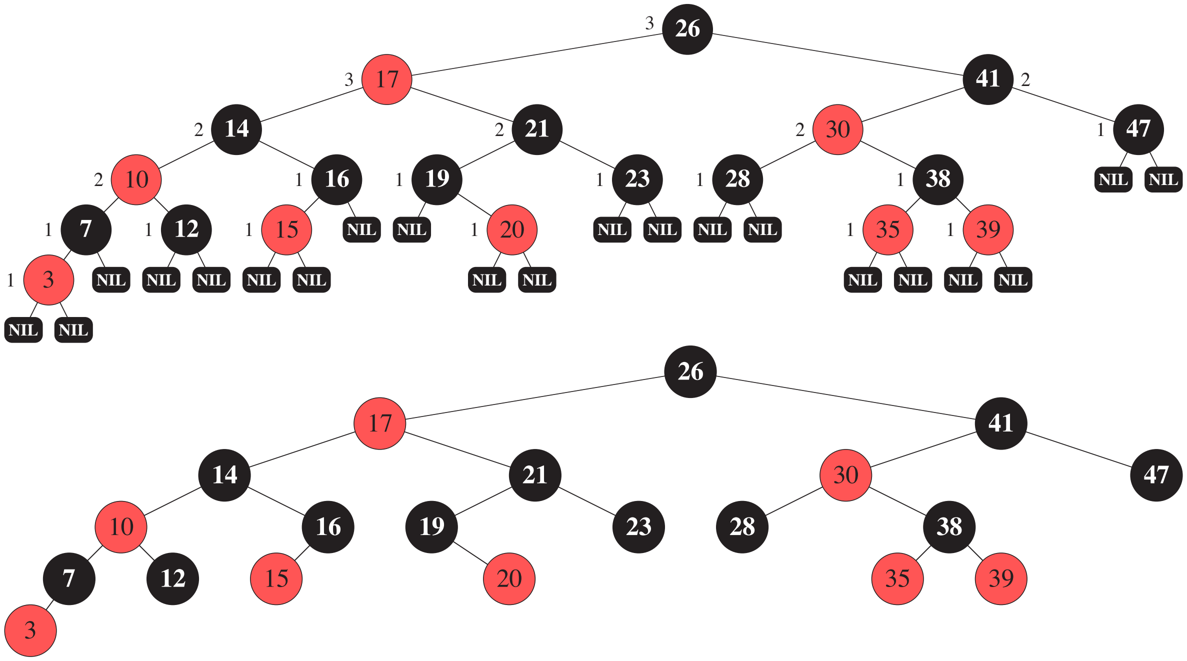

3 26
3 17
41 2
2 14
2 21
2 30
1 47
2 10
1 16
1 19
1 23
1 28
1 38
NIL NIL
1 7
1 12
1 15
NIL
NIL
1 20
NIL NIL NIL NIL
1 35
1 39
1 3
NIL NIL NIL NIL NIL
NIL NIL
NIL NIL NIL NIL
NIL NIL
26
17
41
14
21
30
47
10
16
19
23
28
38
7
12
15
20
35
39
3

Figure 3.1: Red-black trees. The first tree is represented with the NIL sentinel node. The second tree is without the NIL node. The two trees are equivalent and both satisfies the red-black tree properties.

Lemma 3.2.1 The number of nodes in a red-black tree of height h is at least $2^{\lceil (h+1)/2 \rceil} - 1$.

Proof. A red-black tree of height h has a path of length h from the root to a leaf node. This path contains $h+1$ nodes, the first of which is black. Since it does not contain two consecutive red nodes, the path contains at least $b = \lceil (h+1)/2 \rceil$ black nodes (i.e. at least half of the nodes on that path are black). Hence, every path from the root to a node with at most one child contains at least b black nodes. It suffices to prove that there are at least $2^b - 1$ black nodes in a red-black tree of height h such that the number of black nodes in the path from the root to a leaf node is b.

BASE CASE: If the height of the tree is 0, then the number of nodes is 1.

INDUCTIVE STEP: Let $h \in \mathbb{N}$ be arbitrary. Assume that for all tree with height $h' < h$, there are $2^{b'} - 1$ black nodes where b' is the number of black nodes in the path from the root of that tree to a leaf node.

Consider an arbitrary tree T with height h with two subtrees. It follows that there are $b = \lceil (h+1)/2 \rceil$ black nodes from the root of the tree to a leaf node. If the subtree has a black root, then the path going from the root of the subtree to a leaf contains $b-1$ black nodes. If the root is red, then the path contains

b black nodes. Therefore, the number of black nodes in the path from the root of the tree to a leaf node is $b-1$ or b. Then, by induction hypothesis, the number black nodes in each subtree is at least $2^{b-1}-1$.

Since the root of a red-black tree is black, the number of black nodes in T is the number of nodes in the left subtree plus the number of nodes in the right subtree plus the root node.

$$\left(2^{b-1}-1\right)+\left(2^{b-1}-1\right)+1=2^{b}-1$$

By induction, the number of black nodes in a red-black tree of height h is at least $2^{b}-1=2^{\lceil (h+1)/2\rceil}-1$.

■

Corollary 3.2.2 A red-black tree with n nodes has height $h \leq 2\log_2(n+1)-1$.

It follows immediately from this corollary that the SEARCH operation will run in $O(\lg n)$ time on a red-black tree.

3.3 Insertion and Deletion in Red-Black Tree

3.3.1 Rotation Operations

It is obvious that INSERT and DELETE will also run in $O(\lg n)$ time, but the resulting tree may not satisfy the red-black tree properties, meaning that the tree after insertion and deletion of nodes may not be a red-black tree. We can fix this using a technique known as rotation.

3.3.2 Insertion

INSERT(T,z)

We want to first deal with the cases where simple recoloring can fix the problem. We need to determine what color should the newly inserted node be.

If the tree is empty, z should be black.

If the tree is not empty and the newly inserted node has a black parent, we can make the new node red. This will fix the violation.

If the tree is not empty and the newly inserted node has a red parent, we cannot easily fix the violation by recoloring. If we let that newly inserted node be black, it may lead to a violation of property 3, and if we let the new node be red, it will violate property 2 because it has a red parent.

Suppose that the parent p of the new node is red. Then, p will satisfy the following properties.

- p has a black parent g because of property 2;
- the parent has no other child other than the newly inserted node. This is because because of property 2, p's children must be black, but then by property 3, the black child will violate property 3.
- if g has another child, it would be red

z Is a Leaf

Case 0: g has only 1 child. In this case, we perform a right rotation around p and recolor.

Case 1: g has 2 children, recolor according to Figure 3.2. But this might create a new violation at g, so we need to fix that. If g is the root, we can simply change it to black. Otherwise, we need to continue the fix-up procedure at g. By doing so, we move the violation up the tree.

z Is an Internal Node

Now suppose that z is an internal node.

Case 1: z's two children are black, and P has another black child. G is red. In this case, recolor according to Figure 3.2.

Case 2: G is black. Recoloring won't work. We need to do rotation.

z is a left child of a right child or right child of a left child (i.e. a zigzag path from $G \to P \to z$). In this case, do a left rotation at P according to Figure 3.3. After this operation, the tree falls into the next case.

z is a left child of a left child or right child of a right child (i.e. a straight path from $G \to P \to z$). In this case, do a right rotation at G according to Figure 3.4.

Case 1

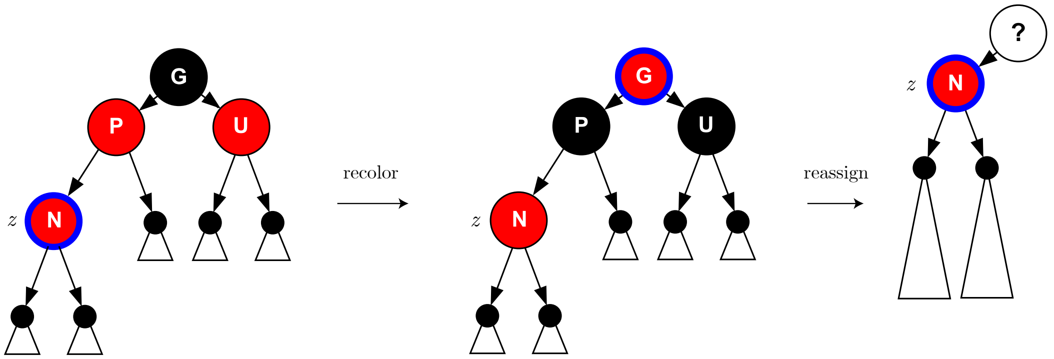


Figure 3.2: <caption>

Case 2

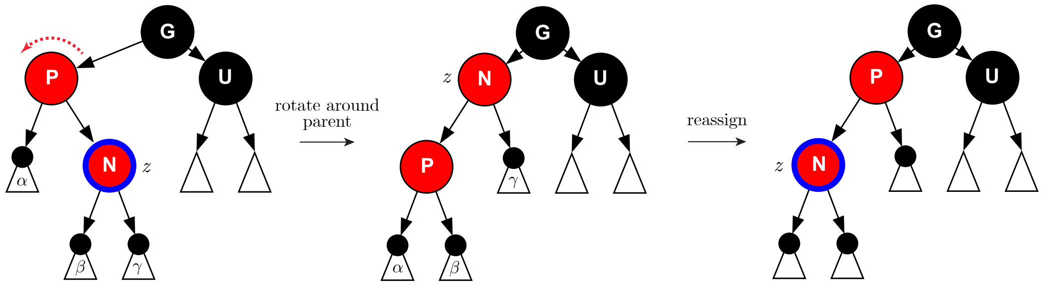

G
P
U
N
z
α
β
γ
rotate around parent
G
N
U
z
P
γ
α
β
reassign
G
P
U
z
N

Figure 3.3: <caption>

Case 3

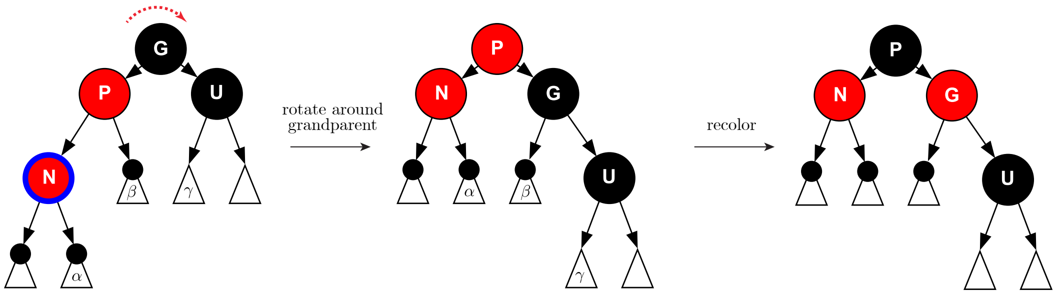

G
P
U
N
β
γ
α
rotate around grandparent
P
N
G
U
α
β
γ
recolor
P
N
G
U

Figure 3.4: <caption>

3.3.3 Deletion

Use the BST implementation of DELETE(T,x).

When x has two children, we replace x by its successor, which gets the same color as x. But then we also need to delete the successor first.

When x has only one child, delete x, promote w and make it black.

When x is a red leaf, we can just delete it without recoloring.

When x is a black leaf, deleting it will cause a violation of property 3, so we need to fix the violation. Let P be the parent of x, and let W be the sibling of x.

Case 1: p is red.

Case 2: p is black

Case 2(a): w has exactly one child Case 2(b): w is black and has two children Case 2(c): w is red and has two children Case 2(d): w has no children

In Case 2(d), w has to be black. Once x is deleted, we have only P and W, which will gives us a black-height of 1 on both sides. This is not fixable, so we make p double-black. And then our goal is to remove the double-black.

3.4 AVL Tree

AVL tree is a height balanced binary search tree. Every node stores a balance factor

$$BF(x) = height(x.left) - height(x.right)$$

To keep an AVL tree balanced, we need to maintain the invariant that every node in an AVL tree has balance factor of -1, 0, or 1.

AVL tree was invented by Adelson-Velski and Landis, and hence the name AVL tree.

Theorem 3.4.1 The height of an AVL tree with n nodes is $O(\log n)$.

Proof. Let $M(h)$ be the minimum number of nodes in any AVL tree with height h. M could be recursively defined such that $M(0) = 1$, $M(1) = 2$, and for all $h \geq 2$, $M(h) = 1 + M(h-1) + M(h-2)$.

We can obtain a closed form of this recurrence, $M(h) = F_{h+3} - 1$ where F_k represents the kth Fibonacci number. Because $F_h > \frac{\phi^h}{\sqrt{5}} - 1$ where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio,

$$n \geq M(h) > \left(\frac{\phi^{h+3}}{\sqrt{5}} - 2 \right)$$

and

$$h < 1.44 \log_2(n+2)$$

■

An alternative proof:

Proof. $M(h) > M(h-1)$, so $M(h-1) > M(h-2)$ for all $h \geq 2$. It follows that

$$M(h) = 1 + M(h-1) + M(h-2) > 2M(h-2)$$

Substitute $h-2$ for h, and we get $M(h-2) > 2M(h-4)$. It can be shown using induction that $M(h) > 2^k M(h-2k)$ for $k > 0$.

Take $k = h/2$. Then, $M(h) > 2^{h/2} M(0) = 2^{h/2}$. Taking logarithm of both sides, we get $h < 2\log_2 M(h)$, and thus $h \in O(n \log n)$. ■

3.4.1 AVL Insertion

At a high level, insertion to an AVL tree works as follows:

1. Perform standard BST insertion.
2. From the inserted node up to the root, update the balance factor, and perform rotations as necessary.

For AVL insertion, we can prove that 1 rotation is sufficient to restore height balance of the entire tree. The idea of the proof is to look at the height of subtrees before the insertion, and compare with the height of the subtrees after the insertion.

3.4.2 AVL Deletion

At a high level, to delete a node from an AVL tree, we perform the following operations:

1. Perform standard BST deletion
    - If x has no children, x is a leaf and can be safely removed;
    - If x has one child y, then y is a leaf. Replace x with y and remove y;
    - x has two children. Replace x with its successor y. Remove y by following one of the previous two cases
2. From deleted leaf up to the root, update the balance factor and perform rotations as necessary.

3.4.3 AVL Rotation Recipes

If we detect a imbalance at a node x where $|BF(x)| > 1$, perform the following checks and rotate accordingly

- LL: $height(x.left.left) \geq height(x.left.right)$
- LR: $height(x.left.left) < height(x.left.right)$
- RR: $height(x.right.right) \geq height(x.right.right)$
- RL: $height(x.right.right) < height(x.right.left)$

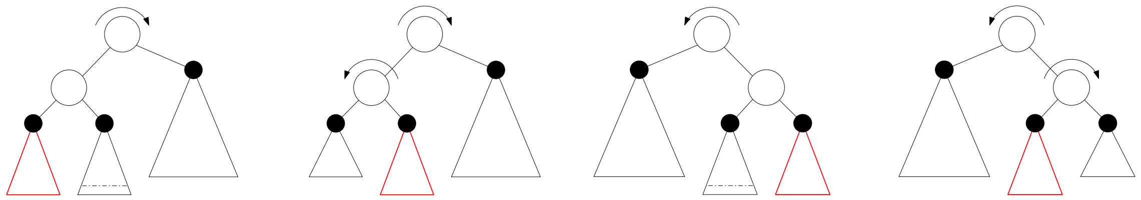


Figure 3.5: AVL rotation recipes, from left to right: LL, LR, RR, RL cases

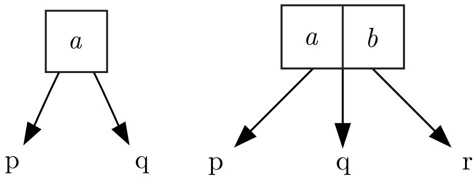
In particular, after insertion, we only need one rotation to fix the imbalance. After deletion, as we retrace back to the root, we might need to fix multiple nodes along the path to root.

3.5 B-Tree

3.5.1 2-3 Tree

Definition 3.5.1 A search tree is a **2-3 tree** if every internal node has either two children with one element, or three children with two elements. Leaf nodes of a 2-3 tree have no children and one or two elements. More formally, a tree T is a 2-3 tree if and only if one of the following is true:

- T is empty
- T has one element a and two children p, q (p being the left child and q being the right child). p and q are 2-3 trees of the same height, and a is greater than every element in p, and a is less than every element in q
- T has two elements $a < b$ and three children p, q, r (left, middle, right children, respectively). p and q are 2-3 trees of the same height; and a is greater than every element in p and less than every element in q; and b is greater than every element in q and less than every element in r.



Chapter 4 Augmenting Data Structures

4.1 Augmenting Data Structures

For many problems, it is not enough to use only the elementary data structures such as linked list, hash table, or binary tree. But for most of those problems, we don't need to reinvent the wheel. Instead, we can augment the data structures we already have along with some additional information.

An example of augmenting data structure is a dictionary ADT with a size variable. This has the benefit of allowing for computing the size of the dictionary in $O(1)$ time.

4.2 Order Statistics With Red-Black Trees

Dictionary implemented by a red-black tree with a MINIMUM(S) operation that returns the pointer to the element with smallest key in S.

Normally, this MINIMUM operation would take $O(\lg n)$ by following the left-most path. But we can make it better by augmenting the red-black tree:

- Maintain a pointer MINPTR to the minimum element.
  When performing INSERT, compare key of newly inserted element and if it is less than the key of the element pointed to by MINPTR, update MINPTR.
  For DELETE, if the element to be deleted is not the minimum element, then do nothing. If the current minimum is deleted, we need to recompute MINPTR using $O(\lg n)$ time.
- Add a variable MINVAL storing the node with minimum value.
- Maintain a doubly linked list of the elements in the list ordered by their key.
  If you insert a node v as a left child of a node p, then v is the predecessor of p. If v is inserted as the right child, then v is the successor of p.
- At each node, store the minimum element in the subtree rooted at that node.
  After insertion and deletion, update the minimum field of the ancestor of the inserted or deleted node. Also updaate the minimum field during red-black tree rotation.

4.2.1 The RANK Operation

The function RANK(S,k) returns the number of elements in S with key $\leq k$. Let's take a look at some elementary data structures that enables the RANK operation.

- Unordered array: linear search, $\Theta(n)$
- Ordered array: binary search, $\Theta(\lg n)$
- Red-black tree: compare with every element in T, or perform an in-order traversal until reached a key $> k$; count the number of nodes visited while doing the traversal. In both cases, $\Theta(n)$.

Without augmentation, the best we can achieve is doing binary search on a sorted array, with $\Theta(n)$ time for the RANK operation.

We can augment a red-black tree in the following way.

With each node, store the number of nodes in its subtree. By doing so, $v.size$ is equal to the number of nodes in the subtree rooted at v.

For $\textsc{Rank}(S,k)$, search for k. Whenever you go right, add the one plus the number of nodes in the left subtree.

If multiple elements in S has the same key, find the last node (in an inorder traversal) with value $\leq k$.

4.2.2 Maintaining the Size Property at Each Node

The problem then arises with how to maintain the size field under insertion and deletion.

When inserting, add 1 to the size field of proper ancestors of newly inserted node. Similarly, when deleting, subtract 1 from the size field of all proper ancestors of the node being deleted v (v is the physically deleted node with ≤ 1 children).

We also need to maintain the size field when doing rotations. Size of a node can be recomputed from the sizes of its children.

$$v.size = v.left.size + v.right.size + 1$$

2. As a small improvement from the previous implementation, we can store $v.left.size$ instead of $v.size$.

3. Store the rank of each node within the node. But then, INSERT and DELETE are now $\Theta(n)$ in the worst case because we need to go back to recalculate the rank.

4.2.3 The SELECT Operation

$\textsc{Select}(S,i)$ returns the element of rank i in the set S.

Suppose S is represented by a RB-tree T. If T is not augmented, we need to do an in-order traversal until i nodes have been visited, which costs $\Theta(n)$ time.

If $T.rootsizeleft \geq i$, then search in the left subtree. If $T.rootsizeleft = i-1$, then return the root. If $T.rootsizeleft < i$, then search in the right subtree for element of rank $i - T.rootsizeleft - 1$.

```
SELECT($v, i$)
r = $v.sizeleft + 1$
if $r > i$
    return SELECT($v.left, i$)
elseif $r == i$
    return $r$
return SELECT($v.right, i - r$)
```

4.3 Steps To Create Augmented Data Structures

To create augmented data strucutres, we typically follow these four steps:

1. choose an underlying data structure
2. determine additional information to be maintained
3. verify that the additional information can be maintained by update operations (or basic steps of update operations, e.g. rotations)
4. develop new operations

Theorem 4.3.1 — Augmenting Red-Black Tree. Let f be a field augmenting each node of a red-black tree and suppose that $x.f$ can be computed using information in node x, $x.left$, and $x.right$, possibly including $x.left.f$ and $x.right.f$. Then, the f field can be maintained in all nodes during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance of these operations.

Proof Idea. A change to $x.f$ only propagates to $y.f$ for the ancestors y of x. Since the height of a red-black tree is $O(\lg n)$, at most $O(\lg n)$ nodes have their f fields changed an each change takes $O(1)$ time.

4.4 Intervals ADT

- Objects: a set of closed intervals $[t, t']$ where $t \leq t'$, or equivalently, the set $\{x \in \mathbb{R} \mid t \leq x \leq t'\}$.
- Operations: $\textsc{Interval-Insert}(S, [t,t'])$, $\textsc{Interval-Delete}(S, [t,t'])$, $\textsc{Interval-Search}(S, [t,t'])$ that returns a pointer to the interval in S that overlaps with $[t,t']$ (non-empty intersection).

Naive implementations:

- Unsorted linked list: $\Theta(n)$
- Sorted linked list: $\Theta(n)$
- Sorted array: $\Theta(n)$
- Red-black tree: store $i.low$ as the key, $O(\lg n)$ for insertion and deletion. For search, skip intervals $i \leq T$ with $t' \leq i.low$.

Augment each node x with

$$\textsc{Max-High}(x) = \max\{y.high \mid \ y \text{ is an interval stored in the subtree rooted at } x\ \}$$

This field can be calculated using

$$x.\mathit{max\text{-}high} = \max\{x.\mathit{high}, x.\mathit{right}.\mathit{max\text{-}high}, x.\mathit{right}.\mathit{max\text{-}high}\}$$

```
INTERVAL-SEARCH($T$, $[t,t']$)
$x = T.root$
while $x \neq$ NIL and $[t,t']$ does not intersect $[x.low, x.high]$
    if $x.left \neq$ NIL and $x.left.max\text{-}high \geq t$
        $x = x.left$
    else
        $x = x.right$
return $x$
```

The time complexity of INTERVAL-SEARCH is $O(\lg n)$.

Theorem 4.4.1 Any execution of INTERVAL-SEARCH$(T, [t,t'])$ either returns a pointer to a node whose interval intersects $[t,t']$, or returns NIL if no such interval exists.

Lemma 4.4.2 Loop invariant: if T contains a node whose interval intersects $[t,t']$, then there is such an interval in the subtree rooted at x.

Chapter 5 Priority Queue and Heap

5.1 Priority Queue

5.1.1 Priority Queue ADT

The priority queue ADT is a data type that stores a collection of items with priorities (keys) that supports the following opeations:

- INSERT(Q,x) inserts the element x into the priority queue Q.
- MAXIMUM(Q) returns the element of Q with the largest key.
- EXTRACT-MAX(Q) removes and returns the element of q with the largest key.
- INCREASE-KEY(S,x,k) increases the value of element x's key into the new value k, assuming that $k \geq x$.

Priority queue allows us to access the element with largest (if max-priority queue) or smallest (if min-priority queue) more efficiently. It has many applications in computer science, such as: job scheduling in operation systems, bandwith management, or finding minimum spanning tree of a graph, etc.

5.1.2 Primitive Implementation Using Linked Lists

We can have a naive implementation of a priority queue simply using a sorted linked list, which has the following time complexity:

- INSERT(Q,x): $\Theta(n)$ in the worst case. We have to linearly search the correct location of insertion.
- MAXIMUM(Q): $\Theta(1)$ by returning the head of the list.
- EXTRACT-MAX(Q): $\Theta(1)$ by removing and returning the head of the list.
- INCREASE-KEY(S,x,k): $\Theta(n)$ in the worst case. Need to move element to new location after increase.

However, we want to have something that is more efficient than $\Theta(n)$. As it turns out, by putting the elements in a specific way, we can achieve worst-case time complexity of $\Theta(\log n)$ for INSERT, EXTRACT-MAX, and INCREASE-KEY. Even better, we can show that the amortized complexity of INSERT is $\Theta(1)$ and EXTRACT-MAX is $\Theta(\log n)$.

5.2 Heap

5.2.1 Types of Binary Trees

Before starting to formally define heaps, let's review some definitions about binary trees.

Definition 5.2.1 — Full Binary Tree. A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.

Definition 5.2.2 — Heap-Shape. A binary tree is in heap-shape if every level of the binary tree, except possibly the last, is completely filled, and all nodes are as far left as possible.

Importantly, a binary tree in heap-shape with n nodes has $\lfloor n/2 \rfloor$ internal nodes (nodes that are not leaves).

Definition 5.2.3 — Complete Binary Tree. A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.

A note on the terminologies: some instructors and textbooks refer to a binary tree in heap-shape as "complete binary tree", and call a complete binary tree "perfect binary tree". In these notes, we will use "heap shape" and "complete binary tree". Unfortunately, both sets of terminologies are accepted and widely used.

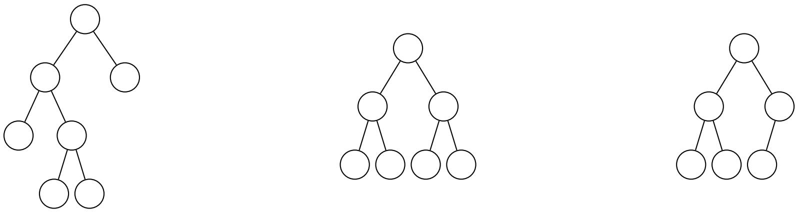


Figure 5.1: From left to right: full binary tree, binary tree in heap shape, complete binary tree.

Conveniently, a binary tree in heap-shape can be represented as an array.

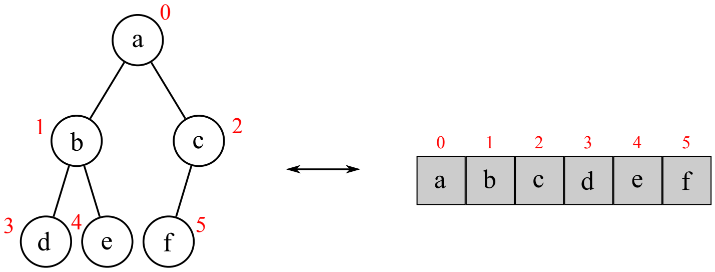


Figure 5.2: A heap-shaped binary tree and its corresponding array representation

Assuming that we index from 0, we can compute the indices of each node's parent, left, and right child.

PARENT(i) = $\lfloor (i-1)/2 \rfloor$

$\textsc{Left}(i) = (2i) + 1$
$\textsc{Right}(i) = (2i) + 2$

If we index from 1, the indices are calculated as follows:

$\textsc{Parent}(i) = \lfloor i/2 \rfloor$
$\textsc{Left}(i) = 2i$
$\textsc{Right}(i) = 2i + 1$

5.2.2 Heap Property

Then, we can define a max-heap as a binary tree in heap-shape with the max-heap property.

> **Definition 5.2.4 — Max-heap Property.** In a max-heap represented by the array A , the max-heap property is that for every node i other than the root,
>
> $$A[\textsc{Parent}(i)] \geq A[i].$$
>
> that is, the value of a node is at most the value of its parent.

The max-heap property guarantees that the largest element in a heap is always stored at its root.

For our heap implementation, we will include the following operations: INSERT, MAXIMUM, EXTRACT-MAX, INCREASE-KEY, MAX-HEAPIFY, BUILD-MAX-HEAP. The first few operations allow us to use heap to implement the priority queue ADT, and in addition to those, BUILD-MAX-HEAP allows us to produce a max-heap from an unordered array.

5.3 Maintaining the Heap Property

Given an array A and index i, MAX-HEAPIFY will correct a single violation of the max-heap property in the subtree with i as its root. To implement MAX-HEAPIFY, we use a technique called “trickle down”.

First, assume that the trees rooted at $\textsc{Left}(i)$ and $\textsc{Right}(i)$ are max-heaps. If element $A[i]$ violates the max-heap property, we correct thie violation by “trickling” element $A[i]$ down the tree until it reaches the correct position. By doing so, we can make the subtree rooted at index i a max-heap. In every trickle-down step, swap $A[i]$ with its largest child.

```
MAX-HEAPIFY$(A, i)$
$l = $ LEFT$(i)$
$r = $ RIGHT$(i)$
if $l \leq A.heapsize$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \leq A.heapsize$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY$(A, largest)$
```

5.3.1 Correctness of MAX-HEAPIFY

5.3.2 Running Time of MAX-HEAPIFY

5.4 Inserting Into Max-Heap

To insert into a max-heap while maintaining the heap property, we use a similar technique. We first append the new element to the end of the heap. The new element will become the right-most leaf at the last level. Then, we check if the element is already at the right position. If not, we "bubble" the element up the tree, until it reaches the correct position.

5.5 Build Heap From Unsorted Array

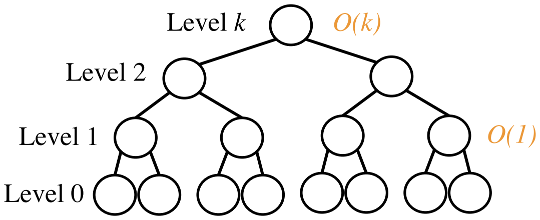
```
BUILD-MAX-HEAP$(A)$
$A.heapsize = A.length$
for $i = \lfloor A.length/2 \rfloor$ downto 1
    MAX-HEAPIFY$(A, i)$
```

The reason we start calling MAX-HEAPIFY at $\lfloor A.length/2 \rfloor$ is because elements beyond that $A[n/2+1, \cdots, n]$ are all leaves of the tree. Recall that for a heap-shaped binary tree with n nodes, there are only $\lfloor n/2 \rfloor$ internal nodes.

5.5.1 Running Time of BUILD-MAX-HEAP

Each call of MAX-HEAPIFY takes $O(\log n)$ time, and BUILD-MAX-HEAP calls MAX-HEAPIFY $O(n)$ times. Thus, the running time of BUILD-MAX-HEAP is $O(n \log n)$. However, this upper bound is not tight. We will prove a tighter upper bound of $O(n)$.

Note that MAX-HEAPIFY takes $O(1)$ time for nodes tthat are one level above the leaves, and in general, it takes $O(k)$ for nodes that are k levels above the leaves. We have $n/4$ nodes at level 1, $n/8$ at level 2, etc. At the root level, which is $\log_2 n$ levels above the leaves, we have only 1 node.



More generally, a heap with n nodes has a height of $\lfloor \log_2 n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes at any height h. Hence, the total cost of BUILD-MAX-HEAP can be written as

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\log_2 n} \frac{h}{2^h} \right).$$

The last summation is bounded by a constant, namely

$$\sum_{h=0}^{\log_2 n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = 2 = O(1).$$

Thus,

$$O\left(n \sum_{h=0}^{\log_2 n} \frac{h}{2^h} \right) = O(n)O(1) = O(n).$$

5.6 Heapsort

```
HEAPSORT($A$)
for $i = A.length$ downto 2
    exchange $A[1]$ with $A[i]$
    $A.heapsize = A.heapsize - 1$
    MAX-HEAPIFY($A, 1$)
```

II

Algorithm Analysis

6 Average Case Complexity and Randomized Algorithms 41
6.1 Basic Probability Theory
6.2 Conditional Probability and Independence
6.3 Average Case Analysis
6.4 Average Case Analysis of Linear Search
6.5 Average Case Analysis of Quick Sort
6.6 Randomized Quicksort
6.7 Randomized Selection

7 Randomness 49
7.1 Random Permutations of an Array
7.2 The Hiring Problem
7.3 Reservoir Sampling

8 Amortized Analysis 53
8.1 Amortized Cost
8.2 Aggregate Method
8.3 Accounting Method
8.4 Potential Method
8.5 Amortized Analysis of Stack with Multipop

9 Dynamic Arrays 61
9.1 Dynamic Array
9.2 Amortized Analysis of Dynamic Array Using Accounting Method
9.3 Amortized Analysis of Dynamic Array Using Potential Method
9.4 Deletion From Dynamic Array
9.5 Amortized Analysis of Dynamic Array With Delete Using Potential Method

10 Scapegoat Tree and Splay Tree 67
10.1 Scapegoat Tree
10.2 Amortized Analysis of Scapegoat Tree

Chapter 6 Average Case Complexity and Randomized Algorithms

6.1 Basic Probability Theory

6.1.1 Sample Space and Events

Definition 6.1.1 — Probability Space and Sample Space. A ***probability space*** $(\Omega, \Pr)$ consists of a finite or countable set Ω called the sample space, and the probability function $\Pr: \ \Omega \to \mathbb{R}$ such that for all $\omega \in \Omega$, $\Pr(\omega) \geq 0$ and $\sum_{\omega \in \Omega} \Pr(\omega) = 1$. We cann an element $\omega \in \Omega$ a sample point, or ***outcome***, or ***simple event***.

A sample space models some random experiment, where Ω contains all possible outcomes, and $\Pr(\omega)$ is the probability of the outcome ω. We always talk about probabilities in relation to a sample space.

Definition 6.1.2 — Event. An event A is a set of outcomes, $A \subseteq \Omega$. We define the probability of an event A to be the sum of the probabilities of its elements

$$\Pr(A) = \sum_{\omega \in A} \Pr(\omega)$$

If we have $\Pr(\omega) = \Pr(\omega')$ for all distinct $\omega, \omega' \in \Omega$, we say that the probability is uniform over Ω.

6.1.2 Properties of Probability Functions

Definition 6.1.3 — Complement. The complement of an event A is

$$\overline{A} = \Omega \setminus A$$

The complement of A is also denoted by not A.

Theorem 6.1.1

$$\begin{aligned} \Pr(\overline{A}) &= 1 - \Pr(A) \\ \Pr(A \cup B) &= \Pr(A) + \Pr(B) - \Pr(A \cap B) \\ \Pr(A \cup B) &\leq \Pr(A) + \Pr(B) \end{aligned}$$

It can then be proved by induction that

$$\Pr(A_1 \cup A_2 \cup \cdots \cup A_k) \leq \Pr(A_1) + \Pr(A_2) + \cdots + \Pr(A_k)$$

for any events $A_1, \cdots A_k$.

We say that two events A and B are disjoint or mutually exclusive if $A \cap B = \emptyset$. If A and B are disjoint, $\Pr(A \cup B) = \Pr(A) + \Pr(B)$.

6.2 Conditional Probability and Independence

Conditional probability allows us to compute the probability of some event given that we already know that some other events have occurred.

Definition 6.2.1 — Conditional Probability. The probability of an event A conditional on an event B is

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)}$$

given that $\Pr(B) > 0$.

6.3 Average Case Analysis

Let A be an algorithm, and let

$t(x) =$ number of steps taken by A on input x.

We know that the worst case time complexity of A is

$$T(n) = \max\{t(x) \mid x \text{ has size } n\}$$

We can define the average case time complexity of A as

Definition 6.3.1 The average case time complexity of A is

$$T'(n) = \mathbb{E}[t(x) \mid x \text{ has size } n]$$

If all inputs of size n are equally likely, then

$$T'(n) = \frac{\sum\{t(x) \mid size(x) = n\}}{|x \mid size(x) = n|}$$

In general, the average case time complexity is less than or equal to the worst case time complexity, that is $T'(n) \leq T(n)$ for all n.

The average case time complexity is dependent on the probability distribution of the inputs, which in turn can depends on the application where the algorithm is to be used. However, this is usually unknown, in which case, assuming uniform distribution usually makes the analysis easier. For some applications, an algorithm with good average case behavior but bad worst case behavior is sufficient.

We follow the following steps before we can analyze the average case time complexity of an algorithm:

- define the sample space of the inputs
- define the probability distribution function
- define any necessary random variables

6.4 Average Case Analysis of Linear Search

Consider the following algorithm for linear search in an unsorted array.

```
LINEAR-SEARCH$(L,k)$
$j = 1$
while $j \leq n$
    if $L[k] == k$
        return $j$
    $j = j+1$
return 0
```

It is obvious that the worst case complexity is $O(n)$ because there will be at most n comparisons with the searching key k. Following the procedure introduced above, we can perform an average case complexity analysis.

1. **Define sample space**: This will be the set of possible inputs that we are considering in our analysis. We define our sample space to be all pairs (L,k) where L is an array of size n and k is a key. Some observations on our choice of sample space:
    - If we fix $L = [l_1, \cdots, l_n]$, then the sample space for k is $[\text{NIL}, l_1, \cdots, l_n]$ where NIL is used to indicate a value that is not in L. Hence, for a fixed L of size n, the sample space for k contains $n+1$ sample points.
    - The number of comparisons performed by LINEAR-SEARCH is the same for (L,k) and (L',k') if k occurs in L at the same position as k' in L'. Duplicates don't matter because the algorithm returns as soon as it finds the first occurrence.
    - If k and k' are not in L, then LINEAR-SEARCH(L,k) will also perform the same number of comparisons.
    - The relative order between elements of L does not matter since LINEAR-SEARCH only performs equality tests, so choosing $L = [1, \cdots, n]$ is just as good as $L = [n, \cdots, 1]$.

    Hence, we can formally define our sample space as

    $$S_n = \{(L,i) \mid i \in \{\text{NIL}, 1, \cdots, n\}\}$$

    where $L = [1, \cdots, n]$.

2. **Probability distribution**: We choose uniform distribution, which means that every point in the sample space has the same probability $\frac{1}{n+1}$.

$$\Pr(k=j)=\frac{1}{n}$$

Alternatively, we can say that NIL has probability of $1/2$ and everything else has probability $1/2n$.

$$\Pr(k=j)=\begin{cases}\frac{1}{2} & \text{if } j=\text{NIL}\\ \frac{1}{2n} & \text{otherwise}\end{cases}$$

3. **Define random variable**: Let $t_n : S_n \to \mathbb{N}$ be such that $t_n(i)$ is the number of comparisons on elements L performed when the input is (L,i) where

$$t_n(i)=\begin{cases} i & \text{for } i=1,\cdots,n\\ n & \text{for } i=\text{NIL}\end{cases}$$

4. **Analysis**:
For the uniform distribution,

$$\begin{aligned}T'(n)=\mathbb{E}[t_n]&=\frac{\sum\{t_n(L,i)\mid i=\text{NIL},\cdots,n\}}{n+1}\\ &=\frac{n+\sum_{i=1}^{n} i}{n+1}\\ &=\frac{n+\frac{n(n+1)}{2}}{n+1}\\ &=\frac{n}{2}+\frac{n}{n+1}\\ &<\frac{n}{2}+1\end{aligned}$$

Hence, $T'(n)\in O(\frac{n}{2})$. This says the average case is twice better than the worst case.

6.5 Average Case Analysis of Quick Sort

Quicksort is ued to sort a multi-set of elements S from a totally ordered domain. The pseudocode for quicksort is shown below.

```
QUICKSORT(S)
if |S| ≤ 1
    return S
pivot = select a pivot
// partition S into L, E, and G such that L contains all elements less than pivot
// E contains all elements equal to pivot, and G contains all elements greater than pivot
L, E, G = PARTITION(S, pivot)
return QUICKSORT(L) + E + QUICKSORT(G)
```

In practice, we often choose the first element of S as the pivot. The worst case time complexity of quicksort is $O(n^2)$ because it has to partition the input S into three parts, L, E, and G, which gives us this recurrence.

$$T(n) = T(n-1) + T(0) + \Omega(n) \in O(n^2)$$

Following the procedure introduced above, we can perform an average case complexity analysis. In QUICKSORT, or most comparison-based sorting algorithms, only the relative order of the elements matter, not their actual values. So, it's reasonable to assume that S_n is the set of all permutations of S_n.

1. Define sample space: S_n = all permutations of $\{1, \cdots, n\}$
2. Probability distribution: We know that $|S_n| = n!$. Assuming uniform distribution, we have

$$\Pr[\pi] = \frac{1}{n!} \qquad \text{for all } \pi \in S_n$$

3. Random variables: Let $t_n : S_n \to \mathbb{N}$ be the random variable such that

$$t_n(i) = \text{number of element comparison performed by } \textsc{Quicksort}(S)$$

4. Analysis: We first consider the complexity measure and the worst-case analysis. We choose the number of comparisons as the complexity measure. The comparisons occur when PARTITION is called. In the worst case, the number of comparisons performed by QUICKSORT is $\leq \binom{n}{2} \in O(n^2)$. The matching lower bound on the worst-case compexity is achieved when QUICKSORT is performed on a sorted array.
Let $T'(n) = \mathbb{E}[t_n]$. Let $X_{ij}\ S_n \to \{0, 1\}$ be an indicator random variable such that $X_{ij}(\pi) = 1$ if and only if elements i and j are compared during $\textsc{Quicksort}(\pi)$.
No pair of elements is compared more than once, so

$$t_n(\pi) = \sum_{1 \leq i < j \leq n} X_{ij}(\pi),$$

and

$$\begin{aligned} T'(n) = \mathbb{E}[t_n] &= \sum_{1 \leq i < j \leq n} \mathbb{E}[X_{ij}] && \text{by linearity of expectation} \\ &= \sum_{1 \leq i < j \leq n} \Pr[X_{ij} = 1] && \text{since } X_{ij} \text{ is an indicator variable} \end{aligned}$$

In $\textsc{Quicksort}(\pi)$, as long as no element in $\{i, \cdots, j\}$ is chosen as a pivot, these elements will stay together, either all going to L, or all going to G in recursive calls.
Eventually, one of the elements in $\{i, \cdots, j\}$ is chosen as pivot. Suppose that p is the first of the elements in $\{i, \cdots, j\}$ to be chosen as pivot. If $i < p < j$, then i and j are not compared during QUICKSORT. If $p = i$ or $p = j$, then i and j are compared, making $X_{ij} = 1$.
There are $j - i + 1$ possibilities for p to be chosen as pivot. Among those possible choices, $X_{ij} = 1$ for two possibilities, and $X_{ij} = 0$ for the rest. Since all permutations of the inputs are equally likely, each of these possibilities for p is equally likely and has probability of $\frac{1}{j-i+1}$.
Hence,

$$\Pr[X_{ij} = 1] = \frac{2}{j - i + 1}$$

Using this, we can rewrite $T'(n)$ as with $k = j - i + 1$

$$\begin{aligned}
T'(n) &= \mathbb{E}[t_n] \\
&= \sum_{1 \le i < j \le n} \Pr[X_{ij} = 1] \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n} \sum_{k=1}^{n-1} \frac{2}{k+1} && \text{substituting } k \\
&< \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} \\
&= n \sum_{k=1}^{n} \frac{2}{k} \\
&\in O(n \lg n) && \text{harmonic series}
\end{aligned}$$

Therefore, the average case time complexity of quicksort is $O(n \lg n)$.

From this analysis, we notice that the average case time complexity for quicksort is much smaller than the worst case time complexity. If the actual distribution of inputs is close to uniform distribution, then the average case time complexity will serve as a more realistic estimate of running time. But problem arises when the actual distribution is not uniform. To solve this issue, we can modify the quicksort algorithm to use a randomized pivot selection.

6.6 Randomized Quicksort

In randomized quicksort, we randomly pick the pivot instead of using the first element of S. Pick each element of S to be the pivot with probability of $\frac{1}{|S|}$ using a random number generator.

```
RANDOMIZED-QUICKSORT($S$)
if $|S| \le 1$
    return $S$
$r = \text{RANDOM}(1, |S|)$
$pivot = S[r]$
// partition $S$ into $L$, $E$, and $G$ such that $L$ contains all elements less than $pivot$
// $E$ contains all elements equal to $pivot$, and $G$ contains all elements greater than $pivot$
$L, E, G = \text{PARTITION}(S, pivot)$
return $\text{RANDOMIZED-QUICKSORT}(L) + E + \text{RANDOMIZED-QUICKSORT}(G)$
```

RANDOM(i, j) will return a number in $\{i, i+1, \cdots, j-1, j\}$ each equally likely, assuming $i \le j$.

The behavior of a randomized algorithm A may depend on its input I and the sequence of choices made by the algorithm.

Let $t(I,p)$ denote the running time of algorithm A on input I with the sequence of random choices p. Then, the expected running time of algorithm A on input I is

$$\mathop{\mathbb{E}}_{p}[t(T,p)] = \sum_{e} \Pr[p]t(I,p)$$

The worst case expected time complexity of algorithm A is

$$T''(n) = \max_{I \in S_n} \mathop{\mathbb{E}}_{p}[t(I,p)]$$

where S_n is the set of all inputs of size n.

Note that the worst case complexity does not depend on any assumption about the input distribution.

R Be careful that the worst-case expected time complexity is the max of expectation (rather than the other way around).

Since all elements in S are equally likely to be chosen as the pivot in each call to Randomized-Quicksort, each of the $j-i+1$ possibilities for p is equally likely. The same average-case analysis for the non-randomized quicksort applies to the worst-case expected analysis of Randomized-Quicksort.

Therefore, the worst-case expected time complexity of Randomized-Quicksort is $O(n \log n)$.

6.7 Randomized Selection

This section is covered in CLRS 9.2. Given a multi-set S and an integer k such that $1 \leq k \leq |S|$, we want to find the kth smallest element in S.

If $S = \{a_1 \leq a_2 \leq \cdots \leq a_n\}$, then return a_k. A naive implementation involves sorting S using Randomized-Quicksort and returning the kth element of the sorted list.

```
Randomized-Select$(S,k)$
if $|S| == 1$
    return $S$
$r = $ Random$(1,|S|)$
$pivot = S[r]$
// partition $S$ into $L$, $E$, and $G$ such that $L$ contains all elements less than $pivot$
// $E$ contains all elements equal to $pivot$, and $G$ contains all elements greater than $pivot$
$L,E,G = $ Partition$(S, pivot)$
if $|L| \geq k$
    return Randomized-Select$(L,k)$
if $|L|+|E| \geq k$
    return $pivot$
return Randomized-Select$(G, k-|L|-|E|)$
```

In Randomized-Select we perform one fewer recursive call than Randomized-Quicksort, so we know that the worst-case time complexity is less than or equal to the compexity of Randomized-Quicksort.

Let S_n be the set of all permutations of $\{1,\cdots,n\}$, which is our sample space. There is no probability distribution because we are already considering a randomized algorithm. Similar to randomized quicksort, we consider the worst-case expected time complexity $T''(n)$.

$$T''(n) \leq \frac{1}{n}\sum_{i=1}^{n}\max\{T''(i-1),\, T''(n-i)\} + n - 1$$

Chapter 7 Randomness

7.1 Random Permutations of an Array

We aim to devise and analyze an algorithm for permuting an array A of n elements. The input is an array A of n elements. The output is the array A with the elements permuted in place.

```
RANDOMIZE-IN-PLACE($A$)
$n = A.length$
for $i$ from 1 to $n-1$
    SWAP($A[i]$, RANDOM($A, i, n$))
```

Claim: at the beginning of the ith iteration, $A[1,\cdots,i-1]$ contains a uniformly random array of length $i-1$. That is, $x_1,\cdots,x_{i-1}$ appears with the probability $\frac{1}{(i-1)!\binom{n}{i-1}} = \frac{(n-i+1)!}{n!}$.

Proof. We will prove the claim using induction.

Base case: for $i = 2$, the first element was chosen randomly from all of $A[1\cdots n]$.

Inductive step: Suppose that as our inductive hypothesis, any sequence of length $k-1$ is likely $k-1$ is likely to occur in $A[1\cdots k-1]$ after $k-1$ iterations. Consider iteration k.

$$\begin{aligned}
\Pr(x_1\cdots x_k \text{ at } A[1\cdots k]) &= \Pr(x_k \text{ at } A[k] \mid x_1\cdots x_{k-1} \text{ at } A[1\cdots k-1]) \cdot \Pr(x_1\cdots x_{k-1} \text{ at } A[1\cdots k-1]) \\
&= \frac{1}{n-k-1} \cdot \frac{(n-k+1)!}{n!} \qquad \text{by inductive hypothesis} \\
&= \frac{(n-k)!}{n!}
\end{aligned}$$

This result implies that $x_1\cdots x_k$ occurs randomly uniformly among all length k subsequences of A. ■

```
PERMUTE-WITH-ALL($A$)
$n = A.length$
for $i$ from 1 to $n-1$
    SWAP($A[i]$, RANDOM($A, \mathbf{1}, n$))
```

This algorithm, unlike the first one, does not produce uniform permutation. As an example, consider the array $A = [1,2,3]$ and the probability that the algorithm produces the output $[1,2,3]$.

Cases:

- $A[1] \leftrightarrow A[1]$ and $A[2] \leftrightarrow A[2]$, with a probability of $1/9$.
- $A[1] \leftrightarrow A[2]$ and $A[2] \leftrightarrow A[1]$, with a probability of $1/9$.
- $A[1] \leftrightarrow A[3]$, not possible to get $[1,2,3]$ by swapping 2.

7.2 The Hiring Problem

The hiring problem is a classic example of average case analysis. Consider the scenario where you want to hire someone from a pool of candidates to replace your current assistant. You want to fire your current assistant only if the new candidate is better than the current one. There is a cost every time you interview an applicant, and another cost if you fire your current assistant. We want to know what is the average cost of hiring the best applicant through this process.

This problem can be alternatively phrased as finding the maximum in an array.

```
ARRAY-MAX(A, n)
curr-max = 0
for i in 1...n
    if A[i] > curr-max
        curr-max = A[i]
return curr-max
```

Let X_i be the indicator random variable such that X_i is 1 if and only if $A[i] = \max\{A[1\dots i]\}$. Observe that the incurred cost is equal to $c \sum X_i$ where c is the cost associated with firing your current assistant (or updating the current max variable). Then, by linearity of expectation, we have the following:

$$\begin{aligned}
\mathbb{E}[c\sum_i X_i] &= c\sum_i \mathbb{E}[X_i] \\
&= c\sum_i \Pr(X_i = 1) \\
&= c\sum_{i=1}^{n} \frac{1}{i} \\
&\leq c\sum_{i=1}^{\infty} \frac{1}{n} = c\log n
\end{aligned}$$

7.3 Reservoir Sampling

Reservoir sampling is an algorithm that allows us to uniformly choose sample from a stream of input of unknown size n. Because the size is unknown, we cannot just pick sample with a fixed probability each time.

```
RESERVOIR-SAMPLING($X, k$)
$reservoir = []$
$i = 1$
for $x$ in $X$
    if $i < k$
        $reservoir$.APPEND($x$)
    $idx =$ RANDOM($1, i$)
    if $idx < k$
        $reservoir[idx] = x$
    $i = i + 1$
return $reservoir$
```

It can be proved by induction that for all $1 \leq j \leq i \leq n$, $\Pr[X_i \in reservoir] = \Pr[X_j \in reservoir]$ after the ith step.

Chapter 8 Amortized Analysis

8.1 Amortized Cost

For many data structures, they need to process a long sequence of operations that access and/or modify the data structures. Knowing the time to process each individual operation is important, which we can measure using the worst-case complexity. However, the time for processing the entire sequence is often more important. In this case, we will use amortized analysis to find the amortized complexity. For example, symbol table in a compiler is analyzed using amortized analysis.

> **Definition 8.1.1 — Worst-Case Sequence Complexity.** The worst-case sequence complexity $C(m)$ is defined as
>
> $$C(m) = \max\{T(\sigma) \mid \sigma \text{ is a sequence of } m \text{ operations}\}$$
>
> C is a function of the length of the sequence.

If there are many possible initial states, we use the notation $T(\sigma, S_0)$ and $C(m, S_0)$ where S_0 is an initial state.

To compute $C(m)$:

- To get an upper bound on $C(m)$, prove that for every sequence σ of m operations, the time to process σ from the initial initial state is $\leq f(m)$. Thus, $C(m) \leq f(m)$.
- To get a lower bound on $C(m)$, prove that for some sequence of m operations from the initial state, the time to process σ is $\geq g(m)$. Thus, $C(m) \geq g(m)$.

> **Definition 8.1.2 — Amortized Cost.** The amortized cost per operation for a sequence of n operations is the total cost of the operations divided by n. Formally,
>
> $$A(m) = \frac{C(m)}{m}$$
>
> for a sequence of m operations.

As a motivating example, consider if you borrowed \$100k mortgage to buy a house. You need to pay it back over 10 years. Assume interest rate is 10% per year.

There are two ways to pay the mortgage:

1. Pay \$10k of the principal each year plus interest that has accumulated over the year. For example, if the interest rate is 10%. Then in the first year, we pay \$10k of the principal and 10% × \$100k which is \$20k in total. In the second year, we pay another \$10k of the principal and 10% × \$90k with a total of \$19k, so on and so forth.
2. Pay the same amount \$16,274 each year. In the first year, most of the payment is the interest. In the tenth year, most of the payment is sthe principal. We say that the mortgage is **amortized** over 10 years. Amortization refers to the notion of spreading payments over multiple periods. Instead of paying a variable amount every year, we split the total payment into equal amounts each year.

In general, the worst-case sequence complexity is less than or equal to m times the worst-case complexity. This happens if every operation in that sequence is the worst case. It follows that the amortized complexity is always less than or equal to the worst-case complexity.

8.1.1 Amortized Analysis of Sorted Linked List

Consider a sorted linked list. There are sequence of m INSERT and SEARCH operations starting from an empty list. For complexity measure, we will count element comparisons.

ith operation takes less than or equal to $i-1$ steps since before the ith operation, the list has length $\leq i-1$. So worst-case time for one operation in a sequence of m operations is $m-1$.

For the amortized complexity, consider the sequence $\text{INSERT}(1), \text{INSERT}(2), \cdots \text{INSERT}(m)$. In this sequence, the ith operation performs exactly $i-1$ comparisons and

$$T(\sigma) = \sum_{i=1}^{m}(i-1) = \frac{m(m-1)}{2}$$

Thus the amortized complexity of insertions and searchings in a sorted linked list is at least $T(\sigma)/m = (m-1)/2$.

In this case, the worst-case complexity is a good upper bound for amortized complexity.

8.1.2 Increasing Binary Counter

Consider an array $X\ [0, \cdots, k-1]$ of bits representing an integer such that

$$x = \sum_{i=0}^{k-1} X[i] \cdot 2^i.$$

Initially, $x = 0$.

We have an operation $\text{INCREMENT}(x)$ which adds 1 to x modulo 2^k.

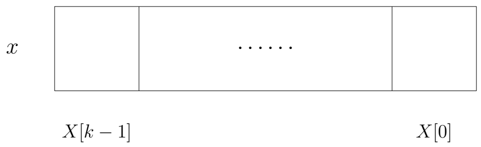


Figure 8.1: The example of a binary counter representing the value x using an array X of length k.

```
INCREMENT(X)
$i = 0$
while $i < k$ and $X[i] == 1$
    $X[i] = 0$
    $i = i + 1$
if $i < k$
    $X[i] = 1$
```

We will count the number of bits flipped as the measure of time complexity. The worst-case number of bit flips in an INCREMENT operation is k (e.g. occurs when all bits of X are 1). As an example, consider the sequence of operations to increment a binary counter of 3 bits from 000 to 111.

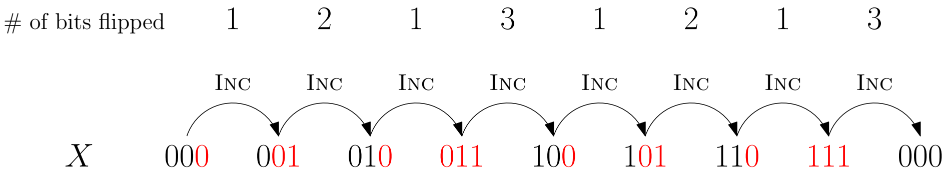


Figure 8.2: The sequence of operations to increment a binary counter of 3 bits from 000 to 111. The bits highlighted in red are the ones to be flipped in the next INCREMENT operation.

There is only one sequence of m INCREMENT operations. In this sequence, $X[0]$ is flipped m times; $X[1]$ is flipped $\lfloor m/2 \rfloor$; $X[2]$ is flipped $\lfloor m/4 \rfloor$; and $X[i]$ is flipped $\lfloor m/2^i \rfloor$ times.

Hence, the worst-case sequence complexity is

$$C(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{m}{2^i} \right\rfloor < m \sum_{i=0}^{\infty} 2^{-i} = 2m$$

Thus,

$$A(m) = \frac{C(m)}{m} \leq 2$$

while the worst-case complexity of an INCREMENT is k.

Generally, there are three methods for performing amortized analysis. Although all methods should give us the same answer, depending on the circumstances, some methods will be easier than others.

- Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the amortized cost to be $T(n)/n$.

- The accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. We use the analogy of a bank account. Prior operations that may impact future operations can store some credits in the bank for uses by future operations. Usually, many short-running operations accumulate credits in small increments, while rare long-running operations use the credits.
- The potential method is like the accounting method, but the balance of the imaginary bank account at each state is given by the potential function.

8.2 Aggregate Method

Get an upper and lower bound on $C(m)$ worst-case sequence complexity and then divide it by m. The previous two analyses on the examples (sorted linked list and binary counter) are all done using the aggregate method.

8.3 Accounting Method

In the accounting method, each kind of operation is assigned a fixed charge $a(op)$ called the **allocated charge** or **amortized cost** of that operation.

When the allocated cost $a(op)$ is greater than the actual cost of op, the excess is associated with specific objects in the data structure as credit. Credits can be used later to help pay for operations where allocated costs are less than their actual costs. The can be viewed as storing money in a bank account for later uses.

If the total credit in the data structure is always non-negative (i.e. the bank account never runs out of money), then the total actual cost of the sequence of operations is less than or equal to the sum of allocated charges of the operations in the sequence.

To prove that the total credit in the data structure is always non-negative, we will use a **credit invariant** (similar to loop invariant used to prove correctness). The credit invariant is a rule that says that parts of the data structure with certain properties will have a certain number of credits associated with them. Credit invariants are proved by induction.

Sometimes, credits are needed to establish the credit invariant for the initial state. Then, the total actual cost of the sequence $\leq$ the cost to establish the credit invariant plus the sum of allocated charge to the operations in the sequence.

$$\begin{aligned}\text{total cost of the sequence} \leq\ & \text{cost to initially establish the credit invariant} + \\ & \text{sum of allocated charges to } m \text{ operations in the sequence}\end{aligned}$$

R It is important NOT to explicitly implement the credit system in the code. It is for analysis purpose only.

We will use the binary counter as a running example when discussing different analysis methods. It is important to define what "a dollar" means and what are the allocated charges for each type of operation.

- We define that \$1 = cost of one bit flip.
- The initial value of the account is 0.
- The allocated charge to INCREMENT is \$2.

Credit Invariant: Each bit with value value 1 has \$1 credit.

When we flip a bit from 0 to 1, we use \$1 to actually flip that bit, and store the additional \$1 as the credit on that bit (in order to maintain the credit invariant).

When we flip a bit from 1 to 0, the associated \$1 credit at that bit can be used to pay for performing the bit fip.

Proof. Initially, all bits are 0, so the credit invariant is vacuously true.

Assume that the credit invariant is true before an INCREMENT operation. Then, all bits of 1 in X contain a \$1 credit.

Case 1: The i least significant bits $X[0], \cdots, X[i-1]$ are 1, and $X[i] = 0$ for some i such that $0 \leq i \leq k$. The actual number of bit flips in this case is $i+1$. Use the i credits associated with the i least significant bits to pay for flipping these bits from 1 to 0. Use \$1 of allocated charge to pay for flipping $X[i]$ from 0 to 1. Use the remaining \$1 of allocated charge to associate as credit with $X[i]$. All other 1 bits still have \$1 credit. Thus the credit invariant is true after this INCREMENT operation.

Case 2: All of the bits are 1, so all k bits are flippted. Use the k credits on these bits to pay for the flips. This maintains the credit invariant. The \$2 allocated to the current INCREMENT are not needed (so they can be donated to charity).

Then by induction, the credit invariant is true after every INCREMENT operation. The cost to establish the credit invariant is 0.

The total actual cost of m operations $C(m) \leq 0 + 2m$. Hence, the amortized cost $A(m)$ of INCREMENTE is at most 2 bit flips. ■

Template 8.1 — Accounting Method.

1. State complexity measure: what \$1 credit can pay for.
2. Assign credits to operations: how many credits are given to what operations; where are credits stored.
3. Explicitly state the credit invariant and prove it.
4. Conclude the amortized cost of each operation.

8.4 Potential Method

Prepaid credits is represented as potential energy associated with the whole data structure. Potential energy in a data structure can be released to pay for later operations.

Definition 8.4.1 — Potential Function. Let D be some data structure with initial state D_0. For each $i = 1, 2, \cdots, n$, let c_i be the actual cost of the i-th operation and D_i be the resulting data structure after applying the i-th operation to data structure D_{i-1}.

The potential function Φ maps each data structure at state i, denoted D_i, to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i.

The amortized cost a of an operation with respect to the potential function Φ is

$$a = \underbrace{c_i}_{\substack{\text{actual}\\ \text{cost}}} + \underbrace{\Phi(D')}_{\substack{\text{potential of configuration}\\ \text{after operation}}} - \underbrace{\Phi(D)}_{\substack{\text{potential of configuration}\\ \text{before operation}}}$$

Consider a sequence of m operations that result in the sequences of data structure configurations $D_0 \to D_1 \to D_2 \to \cdots \to D_m$. Then, the total amortized cost of the m operations is

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} (t_i + \Phi(D_i) - \Phi(D_{i-1})) = \Phi(D_m) - \Phi(D_0) + \sum_{i=1}^{m} t_i$$

Intuitively, $\Phi(D') - \Phi(D)$ is the increase in potential, or the prepaid work (credit). If $\Phi(D_m) \geq \Phi(D_0)$, then $T(\sigma) = \sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i$.

Coming up with a good potential function is the hard and creative part of the analysis. Usually, we choose the potential of the initial configuration to be 0, and the potential to be non-negative after every operation. Similar to a credit invariant, this is typially proved using induction.

8.4.1 Analysis of Binary Counter Using Potential Method

Let C_i be the configuration after ith INCREMENT. Let $\Phi(C)$ be the number of 1 bits in the stored representation of the counter in counfiguration C.

$\Phi(C) \geq 0$ for all configurations C. For the initial configuration C_0, $\Phi(C_0) = 0$ since all bits are 0 initially.

Let a_i be the cost of the ith INCREMENT operation with respect to Φ and let t_i be the actual cost of the ith INCREMENT operation.

If all bits are 1 in configuration C_{i-1}, then there are $t_i = k$ bit flips and $\Phi(C_{i-1}) = k$ and $\Phi(C_i) = 0$. In this case $a_i = k + 0 - k = 0$.

Otherwise, the ith INCREMENT flips 1 bit from 0 to 1 and flips $t_i - 1$ bits from 1 to 0. In this case,

$\Phi(C_i) - \Phi(C_{i-1}) = 1 - (t_i - 1)$, so

$$\begin{aligned} a_i &= t_i + \Phi(C_i) - \Phi(C_{i-1}) \\ &= t_i + (1 - t_i + 1) \\ &= 2 \end{aligned}$$

Thus the amortized cost of each INCREMENT operation with respect Φ is at most 2.

8.5 Amortized Analysis of Stack with Multipop

In this section, we consider stack with an additional MULTI-POP operation, defined as follows:

MULTI-POP(S, k): k is a positive integer. If $|S| > k$, pop top k elements from S; if $|S| \leq k$, pop the entire stack.

Assume that pop and push operations each costs 1, then the actual cost of MULTI-POP(S, k) is $\min(|S|, k)$.

In a sequence of n PUSH and MULTI-POP operations, the worst-case cost of MULTI-POP is $O(n)$, so the worst-case sequence complexity is $O(n^2)$. However, this is not a tight upper bound since we cannot find a matching lower bound where a sequence of operations takes $\Omega(n^2)$ steps. In this case, it is a good idea to consider amortized analysis.

8.5.1 Amortized Analysis of Multi-Pop Using Accounting Method

For the purpose of analysis, assume 1 credit can pay for a PUSH or POP operation.

Each PUSH operation will be assigned a \$2 credit. One credit is used to pay for the PUSH operation, and another credit is stored in the newly pushed element in case it was later popped.

Each MULTI-POP operation is assigned \$0 credit since we can use the credits stored in each element to pay for the POP operation of that element.

We maintain the credit invariant that each element in the stack stores 1 credit.

Proof.

Base case: When the stack was empty, there is no element. The credit invariant is vacuously true.

Inductive step: Assume credit invariant holds before the ith operation. Show it is still true after the ith operation. ■

Chapter 9 Dynamic Arrays

9.1 Dynamic Array

In this section, we will consider the data structure known as dynamic array. It is similar to a regular array, but its length changes according to the "fullness" of the array.

The array will be initialized with a fixed length. Upon calling INSERT, it will insert an element into the array, and whenver the current array becomes full, we create a longer array and copy everything into the new array. Similarly, after calling DELETE, we will shrink the array whenever the array becomes too empty. If we only consider the worst case, the runtime complexities are bad: $O(n)$ for both operations. However, if we look at the amortized cost, the runtime is actually smaller.

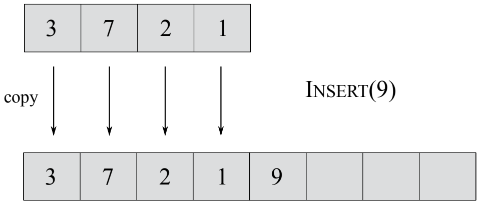


Figure 9.1 The dynamic array after the operation INSERT(9). The length of the new array is doubled, and old elements are copied to the new array.

Idea: When we want to append but the array is full, then we copy the list to an array that is twice as large.

Let $n = \#$ of elements in the list before an APPEND operation. The worst-case cost of APPEND is $n+1$.

9.2 Amortized Analysis of Dynamic Array Using Accounting Method

Let the allocated charge of APPEND be \$3:

- \$1 to pay for the write into current array
- \$1 to save as credit to pay for itself being copied into a new array
- \$1 tax (to pay for a senior citizen to be copied to a new array)

Credit invariant: Each element in the last half of the array has \$2 credits.

Proof. Base case: No elements in the list. The array size can be 0, 1, or 2. For the first append, the credit invariant is true regardless of the initial size of the array.

Inductive step: For any future APPEND, assume that the credit invariant is true before it is performed.

Case 1: If the array is not full, write the element in the first available slot. Use \$1 to pay for the write, and store the remaining \$1. In this case, the credit invariant is clearly maintained after the APPEND operation.

Case 2: If the array is full, all elements in the last half of the array have \$2 credits (if array size is 1, all elements have \$2). The number of elements in the last half of the array is equal to the number of elements in the first half (or no element in the first half if the array size if 0). Copy the array elements from old array to new array of twice the size. Pay for this using stored \$1 credit. Finally, write the new element to the first available slot. It is the only element in the last half of the new array. Hence, the credit invariant is true.

By induction, the credit invariant holds.

Hence, the allocated charge (amortized cost) of APPEND is \$3 $\in O(1)$. ■

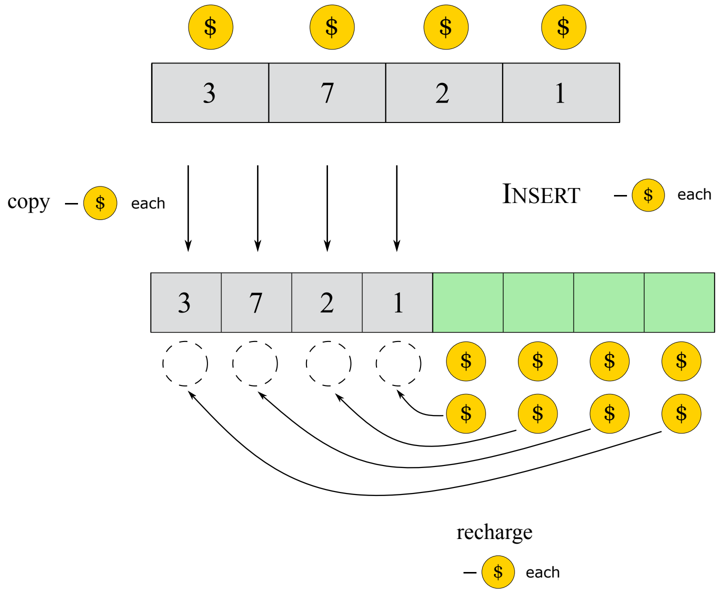


Figure 9.2: The credit scheme for amortized analysis of dynamic array.

9.3 Amortized Analysis of Dynamic Array Using Potential Method

The potential function in any configuration C is defined as

$$\Phi(C) = (2 \times \# \text{ of element in array}) - \text{size of array}$$

If C_0 is the initial configuration and $a_0 \in \{0,1,2\}$ is the initial size of the array, then

$$\Phi(C_0) = 0 - a_0 = -a_0$$

Immediately after the first APPEND, $\Phi(C_1) = 2 - \begin{cases} 1 & \text{if } a_0 = 0,1 \\ 2 & \text{if } a_0 = 2 \end{cases}$. So $0 \le \Phi(C_1) \le 1$.

The amortized cost of the first append with respect to Φ is

$$\underbrace{1}_{\substack{\text{actual cost of} \\ \text{writing new element}}} + \begin{cases} 1 \\ 2 \end{cases} \le 3$$

Immediately before any subsequent expansion, the number of elements in the array is equal to the size of the array so $\Phi(D)$ is equal to the number of elements in the array.

Immediately after an expansion, $2 \times$ elements in array $=$ size of array, so $\Phi(D') = 0$. So the change in potential can be used to pay for copying elements from old array to new array.

Excluding expansion, an APPEND costs \$1 to write plus \$2 for increase in potential since the number of elements in the array increases one and the size of array does not change).

9.4 Deletion From Dynamic Array

Suppose we also want to allow deletion from the end of the array. A simple algorithm would be: when an array overflows, double it; when an array is less than half full, halve it.

The problem with this algorithm is that the amortized complexity is very big! Consider the following sequence of $n = 2^k + 1$ operations.

```
APPEND $2^{k-1}+1$ times
repeat
    DELETE 2 items
    APPEND 2 items
$2^{k-3}$ times
```

The time complexity can be summarized as this table

Operations	Number of resulting elements	Size of resulting array	Cost of operations
$2^{k-1}-1$ APPENDS	$2^{k-1}+1$	2^k	$3(2^{k-1}+1)$
2 DELETES	$2^{k-1}-1$	2^{k-1}	2^{k-1}
2 APPENDS	$2^{k-1}+1$	2^k	$2^{k-1}+2$

An improved algorithm is: when an array overflows, double it; when an array is less than **1/4 full**, then halve it.

9.4.1 Amortized Analysis of Dynamic Array With Deletion

Allocated charge for APPEND is \$3 and for DELETE is \$2. We maintain the following credit invariant:

In a nonempty array of size 2^k where $k \geq 0$, there are at least $\lfloor 2^k/4 \rfloor$ elements. Elements in the last half (i.e. last $\lfloor 2^k/2 \rfloor$ slots) have \$2 stored with them. Empty slots in the first half (i.e. first $\lfloor 2^k/4 \rfloor$ slots) have \$1 stored with them.

Prove the credit invariant by induction.

Proof.

Base case: $k = 0$, array is empty.

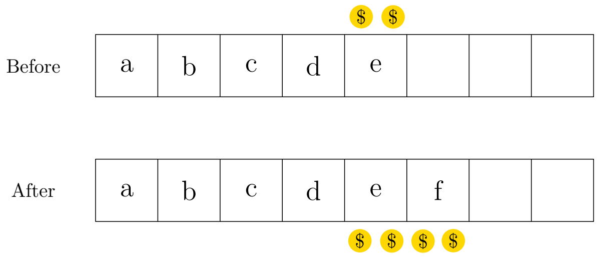
Inductive step: Assume the credit invariant is true before some operation. If the array is empty, then the operation must be APPEND.

If the array is empty, the operation must be APPEND. The element is put into the first half of an array of size 2. There are no elements in the second half and there is no empty slots in the first half. The actual cost of the operation uses \$1. The allocated cost is \$3, so the remaining \$2 can be donated to charity.

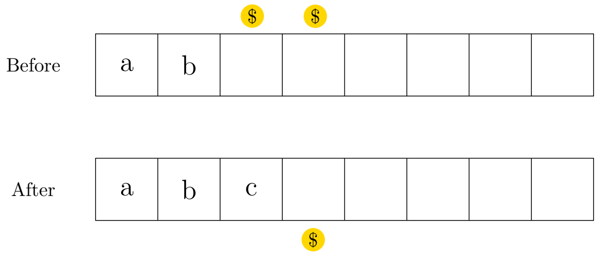
Otherwise, there are four major cases:

Case 1: Append without overflow:

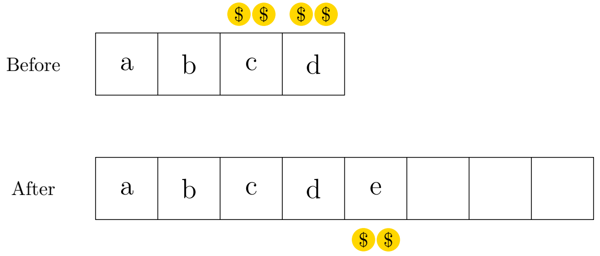
Case 1a (append without overflow, with previous expansion): \$1 actual cost, \$2 to store in the data structure.



Case 1b (append without overflow, without prior expansion): \$1 actual cost, \$1 from data structure, leftover \$3 can be donated to charity.

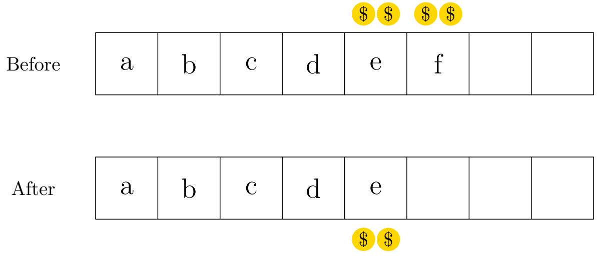


Case 2: Append with overflow: 2^k in data structure. Actual cost $2^k + 1$. Put \$2 into new array with new element. Allocated charge if \$3 is sufficient.

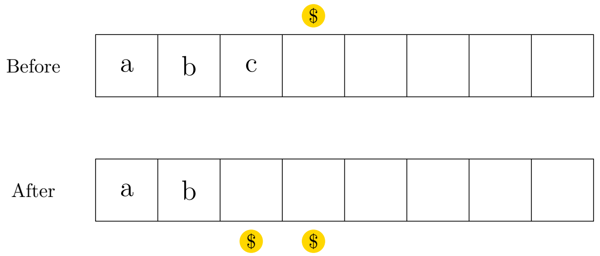

$ $ $ $
Before
a b c d
After
a b c d e
$ $

Case 3: Delete without contraction

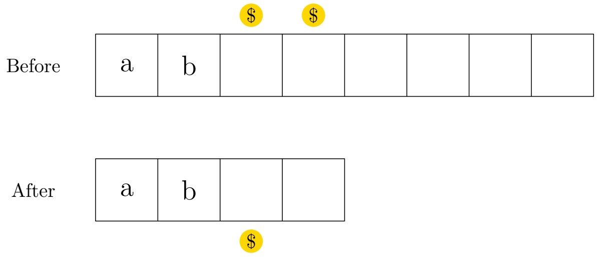
Case 3a: \$1 actual cost, \$2 in data structure. Leftover \$3 donated to charity.


$ $ $ $
Before
a b c d e f
After
a b c d e
$ $

Case 3b: \$1 actual cost, store \$1 in data structure. Allocated charge \$2 is sufficient.


$
Before
a b c
After
a b
$ $

Case 4: Delete with contraction: there is $\$2^{k-2}$ in the data structure, which is used to copy elements from old array. \$2 allocated charge donated to charity if there is leftover.


$ $
Before
a b
After
a b
$

■

Since credit invariant is true and the credit in the data structure is at least 0. The amortized cost of APPEND is at most \$3, and the amortized cost for DELETE is at most \$2.

9.5 Amortized Analysis of Dynamic Array With Delete Using Potential Method

Let Φ be the potential function such that

$$\Phi(D) = |2 \times \text{\# of elements in the list} - \text{size of array}|$$

Let D_i be the state of the data structure after ith operation, and let c_i be the actual cost of the ith operation. Let $a_i = c_i + \Phi(D_i) - Phi(D_{i-1})$. If $\Phi(D_n) > \Phi(D_0)$, then $\sum_{i=1}^{n} a_i = \Phi(D_n) - \Phi(D_0) + \sum_{i=1}^{n} c_i \geq \sum_{i=1}^{n} c_i$.

The absolute value in the definition of the potential function is necessary because otherwise, the potential in the case of deletion with contraction would be negative.

Let $\Phi(D_0) = 3$.

Consider the following cases:

1. Append without overflow: If elements in the list increases by 1, size of the array does not change, so $\Delta\Phi = \Phi(D_i) - \Phi(Di-1) = 2$. Actual cost is 1, so $a_i = 1 + \Delta\Phi = 3$.
2. Append with overflow: $a_i = 2^k + 1 + (2 - 2^k) = 3$:

	Before	After
number of elements	2^k	$2^k + 1$
array size	2^k	2^{k+1}
Φ	2^k	2

3. Delete without contraction: The number of elements in the list decreases by 1, and array size is unchanged. So, $\Delta\Phi = \Phi(D_i) - \Phi(D_{i-1}) \leq 2$. The actual cost is 1, so $a_i = 1 + \Delta\Phi \leq 3$.
4. Delete with contraction: The actual cost is $2^k - 1$ so $a_i = 2^k - 1 + 2 - 2^{k+1} \leq 3$:

	Before	After
number of elements	2^k	$2^k - 1$
array size	2^{k+1}	2^{k+1}
Φ	2^{k+1}	2

In all cases, $a_i \geq 3$, so the amortized cost of APPEND and DELETE is at most 3.

Chapter 10 Scapegoat Tree and Splay Tree

10.1 Scapegoat Tree

Theorem 10.1.1 We say a binary tree is unbalanced at x if either x's left or right subtree contains more than $2/3 size(x)$ nodes. If x is a node at depth greater than $\lfloor \log_{3/2} n \rfloor$ in a binary tree with n nodes, then the binary tree is unbalanced at some proper ancestor of x.

This theorem can be generalized so that $2/3$ is replaced with a constant factor $\frac{1}{2} \leq \alpha < 1$.

10.2 Amortized Analysis of Scapegoat Tree

III

Hashing

11 Hashing 71
11.1 Hashing and Hash Function
11.2 Resolving Collision
11.3 Universal Hashing
11.4 Analysis of Hashing with Chaining
11.5 Perfect Hashing
11.6 Application of Hashing

12 Open Addressing 81
12.1 Open Addressing
12.2 Quadratic Probing
12.3 Double Hashing
12.4 Deletion in Open Addressing

13 Bloom Filter 85

Chapter 11 Hashing

11.1 Hashing and Hash Function

Let U be the universe of possible keys, and let m be the size of the hash table. Then, we say that

$$h:\ U \to \{0, \cdots, m-1\}$$

is a hash function.

If two keys are mapped to the same location/bucket/slot by the hash function, we say that they collide.

Furthermore, if $|U| > m$, then by the pigeonhole principle, there are at least two keys that collide. And because in virtually all cases, $|U| > m$, collision is unavoidable. A well-chosen hash function will minimize the number of collisions, but we still need some means to resolve collisions.

R A fun fact about the etymology of the word "hash": it is said that the word "hash" originated from the french word "hache", which refers to the action of chopping something into pieces. Hashing, as we will see, involves the same notion of randomly chopping and mixing.

11.2 Resolving Collision

Chaining put all elements in $S \subseteq U$ that hash to the same slot in a linked list.

We define the load factor α to be $\alpha = n/m$ where $n = |S|$ and m is the number of buckets (slots) in the hash table. α is the average number of elements of S stored in a slot of the hash table.

Let n_i be the number of elements of S in slot i. Then,

$$\sum_{i=0}^{m-1} n_i = n$$

In the worst case, hashing with chaining takes $\Theta(\max_{0 \leq i \leq m-1} n_i)$ in addition to the time to compute $h(x)$ (the hashing step).

From the analysis above, we know that $\max\{n_i\} = n$ if all elements of S map to the same bucket.

If $|U| > m(n-1)$, then by the pigeonhole principle, there exists $S \subseteq U$ with $|S| = n$ such that all element of S hash to the same bucket.

Overall, hashing with chaining without randomization has the following time complexity:

- $\text{INSERT}(x)$: $O(1)$ assuming all linked lists are unsorted and $x \notin S$
- $\text{DELETE}(p)$: $O(1)$ if list is doubly linked; if singly linked, then $O(n_i)$ where n_i is the size of the bucket where p is in.

11.3 Universal Hashing

We can achieve better time complexity by randomly choosing the hash function. There is a family of hash functions that we call the universal hash functions, which we will define as follows.

> **Definition 11.3.1 — Universal Hashing.** Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U into the range $\{0, \cdots, m-1\}$. Such a collection is said to be universal if for each pair of distinct keys, $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between distinct keys is not more than the chance $1/m$ of a collision if $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, \cdots, m-1\}$.
>
> That is, a finite set $\mathcal{H}$ of hash functions from $U \to [0, \cdots, m-1]$ is universal if for all $x \neq y \in U$
>
> $$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$$
>
> or equivalently,
>
> $$|\{h \in \mathcal{H} \mid h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{m}$$

Here are some examples of universal hashing families.

1. $\mathcal{H}$ is the set of all functions from $U \to [0, \cdots, m-1]$ assuming U is finite. Let $u = |U|$. Then, $|\mathcal{H}| = m^u$. For any distinct $x, y \in U$

   $$|\{h \in \mathcal{H} \mid h(x) = h(y)\}| = m^{u-1} = \frac{|\mathcal{H}|}{m}$$

   since there are m choices for which bucket each element of $U - \{y\}$ gets mapped to. So,

   $$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{m}$$

   Therefore, $\mathcal{H}$ is universal.
2. Let p be prime and $U = \{0, \cdots, p-1\} = \mathbb{Z}_p$. And let

   $$h_{a,b}(x) = [(ax+b) \bmod p] \bmod m$$

and

$$\mathcal{H}_{p,m} = \{h_{a,b} : U \to [0,\cdots,m-1] \mid a,b \in U, a \neq 0\}$$

So $|\mathcal{H}_{p,m}| = p(p-1)$. We can prove that $\mathcal{H}_{p,m}$ is universal.

3. Let $U = \{0,\cdots,2^k-1\}$ and let

$$h_a(x) = \left\lfloor \frac{ax \bmod 2^k}{2^{k-m'}} \right\rfloor$$

This selects the $(k-m'+1)$th through kth least significant bits. In other words, we throw away everything in ax except the k least significant bits, and then remove $k-m$ least significant bits. The size of the hash table is $m = 2^{m'}$.

Let

$$\mathcal{H}' = \{h_a \mid 0 < a < 2^k, a \text{ is odd}\}$$

We have $|\mathcal{H}'| = u/2$. $\mathcal{H}'$ is universal. The proof is more complicated.

For each of the universal hash families, let's also take a look at their sizes

1. $|\mathcal{H}| = m^u$. To specify a hash function in $\mathcal{H}$, we need $u\log_2 m$ bits. This is too big.
2. $|\mathcal{H}_{p,m}| < u^2$. To specify a hash function $\mathcal{H}_{p,m}$, we need less than $2\log_2 u$ bits.
3. $|\mathcal{H}'| = u/2$. To specify a hash function in $\mathcal{H}'$, wee need $(\log_2 u) - 1$ bits.

11.4 Analysis of Hashing with Chaining

Fix $S \subseteq U$ where $|S| = n$. Pick $h \in \mathcal{H}$ randomly where $\mathcal{H}$ is a universal family of hash functions from $U \to \{0,\cdots,m-1\}$.

Let $x \in U$. Let $C_x : \mathcal{H} \to \mathbb{N}$ be such that

$$C_x(h) = \text{the number of keys in } S \text{ that hash to } h(x)$$

C_x is a random variable that depends on the choice of h.

For each $y \in S$, let $C_{x,y} : \mathcal{H} \to \{0,1\}$ be the indicator random variable that is 1 if and only if $h(x) = h(y)$.

$$C_x(h) = \sum_{y \in S} C_{x,y}(h)$$

Theorem 11.4.1 — Expected Number of Collisions For Each Key in Universal Hashing.

$$\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C_x] \leq 1 + \frac{n}{m} = 1 + \alpha$$

Proof. If $y = x$, then $C_{x,y}(h) = 1$ for all $h \in \mathcal{H}$ so $\mathbb{E}_{h\in\mathcal{H}}[C_x] = 1$.

If $y \neq x$, then

$$\begin{aligned}\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C_{x,y}] &= \Pr_{h\in\mathcal{H}}[C_{x,y}(h)=1] && \text{since } C_{x,y} \text{ is indicator variable}\\ &= \Pr_{h\in\mathcal{H}}[h(x)=h(y)] \leq 1/m && \text{since } \mathcal{H} \text{ is universal}\end{aligned}$$

Hence,

$$\begin{aligned}\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C_x] &= \sum_{y\in S}\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C_{x,y}] && \text{by linearity of expectation}\\ &\leq 1+\sum_{y\in S-\{x\}}\frac{1}{m} \leq 1+\frac{n}{m} = 1+\alpha\end{aligned}$$

■

Corollary 11.4.2 The worst-case expected search time for x is $O(1+\alpha)$.

Corollary 11.4.3 Starting with an initially empty hash table of size m, the worst-case expected time to handle any sequence of s INSERT, DELETE, SEARCH operations containing $n = O(m)$ INSERT operations is $O(s)$. Hence, we can perform each operation in $O(1)$ time.

11.5 Perfect Hashing

h is perfect for S if for each element of S hashes to a different slot (i.e. no collisions). For example,

$$h(x) = x \bmod 5$$

is perfect for $\{1,14,20\}$, but not for $\{1,9,14\}$.

For a static dictionary (where S does not change with no INSERT and DELETE),then it may be worthwhile to find a perfect hash function for S. For example, it is useful to construct a perfect hash function if we are designing a compiler for a programming language, and our S is the set of reserved keywords in the language.

11.5.1 Constructing a Perfect Hash Functions

Let S be a set of n keys. Let $\mathcal{H}$ be a universal family of hash functions. Let $C : \mathcal{H} \to \mathbb{N}$ be the random variable so that

$$C(h) = \text{the number of collisions when } h \text{ is used to hash } S$$

More formally,

$$C(h) = |C_{x,y} \in S \mid x \neq y, h(x) = h(y)|$$

where $C_{x,y}$ is defined similarly as the indicator variable that is 1 if and only if $h(x) = h(y)$.

Lemma 11.5.1 — Expected Number of Collisions in Universal Hashing.

$$\mathbb{E}[C] \le \frac{\binom{n}{2}}{m}$$

Proof.

$$C(h) = \sum \{C_{x,y}(h) \mid x < y, x, y \in S\}$$

By linearity of expectation,

$$\begin{aligned}
\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C] &= \sum \{ \mathop{\mathbb{E}}_{h\in\mathcal{H}}[C_{x,y}] \mid x, y \in S, x < y\} \\
&= \sum \{ \mathop{\Pr}_{h\in\mathcal{H}}[h(x) = h(y)] \mid x, y \in S, x < y\} \\
&\le \sum \{1/m \mid x, y \in S, x < y\} && \text{since } \mathcal{H} \text{ is universal} \\
&= \frac{\binom{n}{2}}{m} && \text{since there are } \binom{n}{2} \text{ pairs of keys that may collide}
\end{aligned}$$

■

If $m > \binom{n}{2}$, then $\mathbb{E}_{h\in\mathcal{H}}[C] < 1$. Since $C(h) \in \mathbb{N}$ for all $h \in \mathcal{H}$, this implies that there exists some $h \in \mathcal{H}$ such that $C(h) = 0$. This tells if we are willing to use $\Theta(n^2)$ space for the hash table, there exists a hash function $h \in \mathcal{H}$ that is perfect for S, so that SEARCH only takes only one step.

Theorem 11.5.2 If $m > 2\binom{n}{2} = n(n-1)$, then

$$\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C] < \frac{1}{2}$$

This means more than half of the functions in $\mathcal{H}$ is perfect for S.

The randomized algorithm for constructing a perfect hash function, we

```
CONSTRUCT-PERFECT-HASH($\mathcal{H}$, $S$)
Pick $h \in \mathcal{H}$ uniformly at random.
for $s$ in $S$
    if there is a collision
        start over and try again
if no collision
    return $h$
```

If $m > 2\binom{n}{2}$, the expected number of tries is less than 2, and the algorithm takes $O(2n)$ time to find a perfect hash function for S.

11.5.2 FKS Hashing

However, sometimes when the size of S is large, the $O(n^2)$ space that our current perfect hashing table is not longer ideal. In this case, we want to use $m \in O(n)$ space for the hash table. Then,

$$\mathbb{E}[C] = \frac{\binom{n}{2}}{m} \in O(n)$$

Some buckets will have size > 1, but all buckets are likely to have size $O(\sqrt{n})$. Essentially, if a bucket have size b, there are at least $\binom{b}{2}$ collisions.

The idea is to use a perfect hash function to represent each bucket. If the size of bucket i is b, we will use b^2 space for it. The resulting data structure is a 2-level hash. For $i = 0, \cdots, m-1$, let $n_i(h) = \{x \in S \mid h(x_i) = i\}$ be the number of keys in S that get mapped to bucket i by h. An example of the resulting hash table is shown in Figure 11.1.

$$\sum_{i=0}^{m-1} n_i(h) = n$$

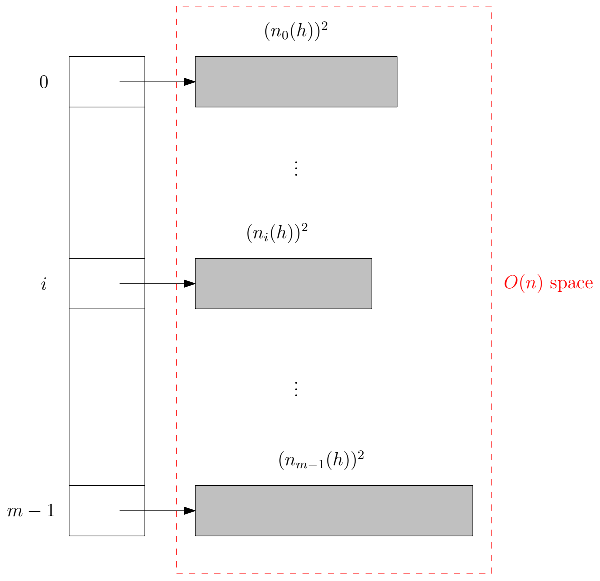


Figure 11.1: FKS 2-level hash table. As shown in the diagram and explained above, the length of each secondary hash table is $(n_i(h))^2$.

The expected amount of space used by all the secondary hash tables can be computed as follows:

$$\begin{aligned}
C(h) &= \sum_{i=0}^{m-1} \binom{n_i(h)}{2} \\
&= \sum_{i=0}^{m-1} \frac{n_i(h)^2}{2} - \sum_{i=0}^{m-1} \frac{n_i(h)}{2} \\
&= \frac{1}{2}\sum_{i=0}^{m-1} n_i(h)^2 - \frac{n}{2}
\end{aligned}$$

By linearity of expectation,

$$\begin{aligned}
\mathbb{E}\left[\sum_{i=0}^{m-1} n_i(h)^2\right] &= 2\,\mathbb{E}[C] + n \\
&= 2\frac{\binom{n}{2}}{m} + n \in \Theta(n) \quad \text{if } m \in \Theta(n)
\end{aligned}$$

h itself is not a perfect hash function. For each non-empty bucket, find a hash function $h_i : U \to \{0, \cdots, m_i - 1\}$, where $m_i = (n_i(h))^2$, that is perfect for the seconday hash table at bucket i (i.e. perfect for the set $\{x \in S \mid h(x) = i\}$).

11.5.3 Representing FKS Hash Table

In practice, we can represent the two-level FKS hash table for $S \subseteq U$ as a single array. Suppose that $U = \{0, \cdots, p-1\}$. We use $\mathcal{H}_{p,m}$ for different values of m.

The first 3 locations of the array contain m, a, and b used to specify the first-level hash function h. In particular, m is the size of the first-level hash table; a and b are the parameters for the first-level table.

The next m locations contain pointers to array corresponding to each second-level hash table.

Each second-level table is stored preceeded by the specifications of its hash function: $m_i = n_i(h_{a,b})$ and the parameter $a_i + b_i$ that specifies the perfect hash function h_{a_i,b_i} for the ith secondary hahs table.

■ **Example 11.1** Let $p = 31$. $U = \{0, \cdots, 30\}$, $n = 6$, and $S = \{2, 4, 5, 15, 18, 30\}$, so $n = 6$. Let the size of the first-level hash table be m.

We randomly choose a hash function from $\mathcal{H}_{31,6}$ for the first-level table. Suppose that we end up getting

$$h_{2,0}(x) = [(2x + 0) \bmod 31] \bmod 6$$

as the hash function.

In this case,

- Bucket 0 contains: 15; so $n_0(h) = 1$ and $n_0(h)^2 = 1$,
- Bucket 2 contains: 4; so $n_2(h) = 1$ and $n_2(h)^2 = 1$,

- Bucket 4 contains: 2, 5; so $n_4(h) = 2$ and $n_4(h)^2 = 4$,
- Bucket 5 contains: 18, 30; so $n_5(h) = 2$ and $n_5(h)^2 = 4$, and
- All other buckets are empty with $n(h) = 0$.

The two-level hash table would looks like Figure 11.2

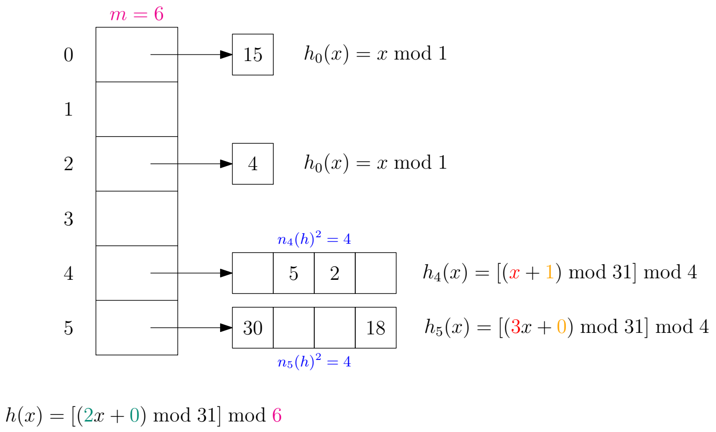

$m = 6$
0
1
2
3
4
5
15
$h_0(x) = x \bmod 1$
4
$h_0(x) = x \bmod 1$
$n_4(h)^2 = 4$
5
2
$h_4(x) = [(x + 1) \bmod 31] \bmod 4$
30
18
$h_5(x) = [(3x + 0) \bmod 31] \bmod 4$
$n_5(h)^2 = 4$
$h(x) = [(2x + 0) \bmod 31] \bmod 6$

Figure 11.2: Example of a two-level hash table

If we were to represent this two-level using a 1-D array, it should look like Figure 11.3 ■

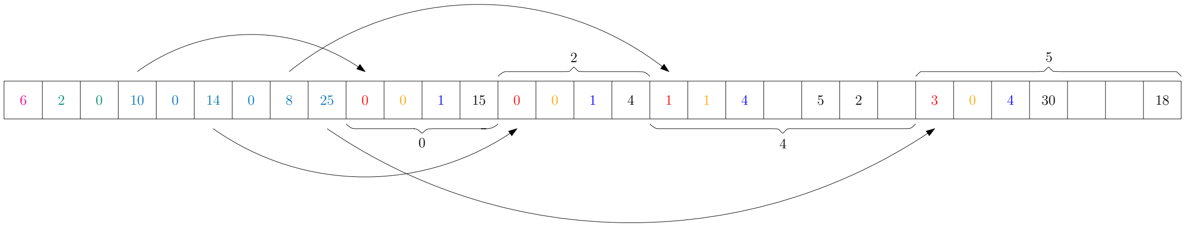

6
2
0
10
0
14
0
8
25
0
0
1
15
0
0
1
4
1
1
4
5
2
3
0
4
30
18
2
5
0
4

Figure 11.3: The previous two-level hash table represented by an array. The colored represents the corresponding parameters in the two-level hash table representation.

R This two-level hash table is often called the FKS hash table, which owes its name to its inventors M. Fredman, J. Komlós, and E. Szemerédi. They published the article titled *Storing a Sparse Table with O(1) Worst Case Access Time* in 1984, in which they first introduced this kind of hash table.

11.6 Application of Hashing

Given an array A of n numbers, determine if all of them are different.

- Sort the array and then check if adjacent numbers are duplicates. The time complexity would be $O(n\log n+n)$, where the $n\log n$ is from sorting the array and n is for iterating over the array.
- Insert every element into the hash table on at a time. Each time, check for duplicates with a search. This take $\Theta(n)$ worst-case expected time.

Given a list of n possible integers $x_1, x_2, \cdots, x_n$, determine in $O(n)$ worst-case expected time if there exists and i and j such that $x_i = x_j + 1$.

Chapter 12 Open Addressing

12.1 Open Addressing

The idea of open addressing is to store keys directly into the hash table. If the hash table is full, we cannot insert new elements, unless we create a new hash table and double the size. In open addressing, rather than mapping each key to a single slot and use chaning to resolve collision, we instead compute a sequence of slots. With such approach, the load factor α can never exceed 1.

Let $h: U \times \{0, \cdots, m-1\} \to \{0, \cdots, m-1\}$. For each key k_1, its probe sequence

$$(h(k,0), h(k,1), \cdots, h(k,m-1))$$

is a permutation of $(0, 1, \cdots, m-1)$

For $\textsc{Insert}(H,k)$, we put k into the first empty (N_i) slot in its probe sequence.

12.1.1 Linear Probing

Let $h': U \to \{0, \cdots, m-1\}$ be the auxillary hash function. Linear probing uses the hash function h, which is defined as follows.

$$h(x,i) = (h'(x) + i) \bmod m$$

```
Insert(H, x)
    Put x into the first empty slot in its probing sequence
```

```
Search(H, x)
    Examine each slot in the its probing sequence. Return if found.
    if empty slot or m elements have been searched
        return False
```

As an example, consider a hash table with $m = 8$ with the auxillary hash function $h'(x) = x \bmod m$ implemented using linear probing.

Linear probing is good at minimizing page fetches as it exploits locality of reference.

12.1.2 Analysis of Linear Probing

Theorem 12.1.1 Assume that for each key x_1 that its probe sequence is equally likely to be any permutation of $\{0 \cdots m-1\}$. For any $\alpha < 1$, the expected number of probes in

- a successful search $\leq \frac{1}{1-\alpha}$
- an unsuccessful search $\leq \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$

For linear probing, there are only m possible sequences, not $m!$. The problem with linear probing is that it suffers from primary clustering where clusters of keys occur and occupied slots gets longer, increasing the average search time.

12.2 Quadratic Probing

In quadratic probing, we use the auxillary hash function h' along with the hash function

$$h(x,i) = (h'(x) + c_1 i + c_2 i^2) \bmod m$$

for some constant c_1, c_2 and $c_2 \neq 0$.

Our goal is to generate permutations of $\{0 \cdots m-1\}$.

Similar to linear probing, we still only get m probing sequences, and it also has its own clustering problem known as secondary clustering.

12.3 Double Hashing

For double hashing, we use two auxillary hash functions h_1 and h_2 for the main hash function h, which is defined as

$$h(x,i) = (h_1(x) + i h_2(x)) \bmod m$$

The idea behind this choice of hash function is that we want to “jump by $h_2(x)$” starting from $h_1(x)$.

In practice, we choose h_2 such that $h_2(x)$ is relatively prime with m.

This generates $\Theta(m^2)$ probing sequences.

12.4 Deletion in Open Addressing

Although insertion to a hash table implemented using open addressing is relatively simple, it has its unique problem when it comes to deletion. We cannot simply set the value at a slot to NIL. If we did so, we will be unable to retrieve any key k during whose insertion probing we had skipped the slot previously occupied by the deleted element. To solve this, we use a method called “tombstone”. Essentially, we mark the deleted slot with a special marker DELETED known as the tombstone. The tombstone indicates that there used to be an item at that slot but is no longer there. We will still skip the tombstone during probing as if there is still an item.

Chapter 13 Bloom Filter

IV

Advanced Data Structures

14 Binomial Heaps 89
14.1 Mergeable Priority Queue ADT
14.2 Binomial Heap
14.3 Operations on Binomial Heap

15 Fibonacci Heaps 95

16 Disjoint Sets 97
16.1 Disjoint Set ADT
16.2 Data Structures for Disjoint Sets
16.3 Amortized Analysis of Disjoint Set

Chapter 14 Binomial Heaps

14.1 Mergeable Priority Queue ADT

Mergeable priority queue is a type of priority queues that support all the operations of a regular priority queue, in addition to the operation UNION

UNION(A,B): returns a new priority queue containing all elements from A and B. As a precondition, we may assume that A and B are disjoint.

Some implementations of a mergeable priority queue may also support the operation DECREASE-PRIORITY and DELETE.

14.2 Binomial Heap

In this section, we will examine a data structure that implements the mergeable priority ADT. It supports all operations in $\Theta(\log n)$ time.

> **Definition 14.2.1 — Binomial Tree.** The binomial tree B_k of degree k is defined recursively as follows:
>
> - B_0 consists of 1 node;
> - B_k consists of two binomial trees of degree $k-1$, where the root of one of the trees is made the first child of the root of the other.

A binomial tree B_k of degree k has 2^k nodes, and there are $\binom{}{ki}$ nodes at depth i for all $i = 0, \ldots, k$. The root of B_k has k children, each of which is a root of a binomial tree. The subtree rooted at the ith child is B_{k-i}. In other words, the children of the root of B_k are the roots of $B_{k-1}, \ldots, B_0$ in order from left to right.

Each node, in addition to its key, stores its degree, pointer to its parent, pointer to its leftmost child, and pointer to its next sibling.

A binomial heap is a linked list of binomial trees with increasing large degree such that each binomial tree satisfies the priority condition (i.e. the priority of a node is less than or equal to the priority of its parent). The next sibling field of the root of the binomial trees are used to link the binomial trees together into this linked list.

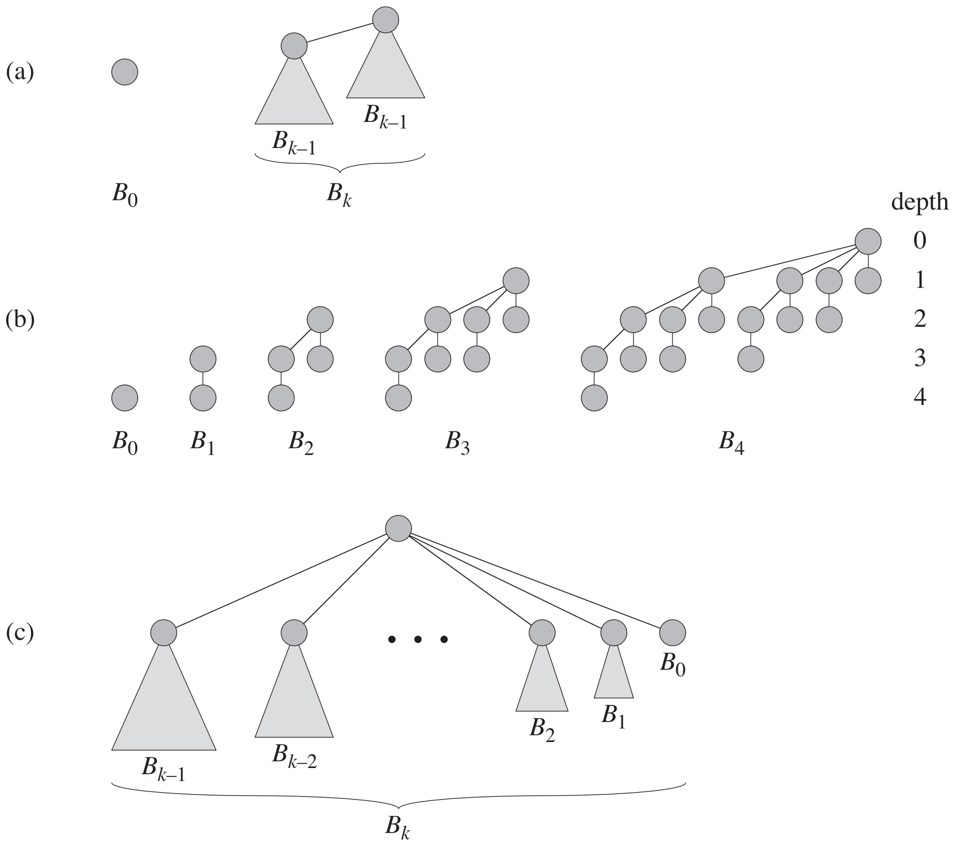

(a)
B_{k-1}
B_{k-1}
B_0
B_k
(b)
depth
0
1
2
3
4
B_0
B_1
B_2
B_3
B_4
(c)
B_{k-1}
B_{k-2}
B_2
B_1
B_0
B_k

Figure 14.1: (a) The recursive definition of binomial tree; (b) Binomial trees of degree 0-4; (c) Binomial heap viewed as rooted tree

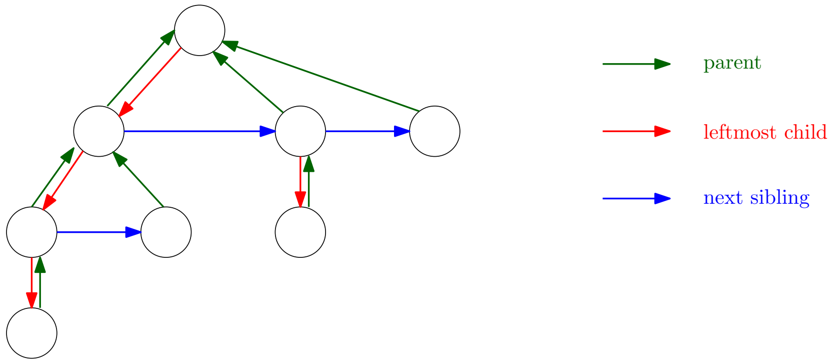

parent
leftmost child
next sibling

Figure 14.2: Pointers in a binomial tree.

There is at most one binomial tree of each degree in a binomial heap.

A binomial heap with $n > 1$ elements has at most $\lceil \log_2 n \rceil$ binomial trees. If n is a power of 2, then a binomial heap with n nodes consists of 1 binomial tree. For example, a binomial heap with 27 nodes consists of 4 binomial trees.

$$B_0 \to B_1 \to B_3 \to B_3 \qquad \Leftarrow \qquad \left(\underbrace{1}_{B_4}\underbrace{1}_{B_3}\underbrace{0}_{\text{nothing}}\underbrace{1}_{B_1}\underbrace{1}_{B_0}\right)_2 = 27$$

14.3 Operations on Binomial Heap

14.3.1 Linking Two Binomial Trees

Linking 2 binomial trees of the same degree is easy because linking B_k and B_k simply gives a new binomial tree of degree $k+1$. More specifically, we update the first child pointer and the degree of the root at one tree, and chagne the parent pointer and next sibling pointer of the root of the other tree. The linking operation takes $O(1)$ time.

```
BINOMIAL-LINK$(y, z)$
    $y.parent = z$
    $y.sibling = z.child$
    $z.child = y$
    $z.degree = z.degree + 1$
```

14.3.2 Union Two Binomial Heaps

To union two binomial heaps, we need to merge the two linked lists sorted by degree. We link the roots of equal degrees in increasing order until at most 1 root of each degree remains. This is somewhat similar to the merge step in merge sort and analogous to **addition of binary numbers**.

For example, starting with two linked lists of binomial trees

$$B_0 \to B_1 \to B_3 \to B_4 \qquad B_0 \to B_1 \to B_3 \to B_4 \to B_5$$

$$B_0 \to \boxed{B_1 \to B_1} \to B_3 \to B_3 \to B_4 \to B_4 \to B_5$$

$$B_0 \to B_2 \to \boxed{B_3 \to B_3} \to B_4 \to B_4 \to B_5$$

$$B_0 \to B_2 \to B_4 \to \boxed{B_4 \to B_4} \to B_5$$

$$B_0 \to B_2 \to B_4 \to \boxed{B_5 \to B_5}$$

$$B_0 \to B_2 \to B_4 \to B_6$$

If the first binomial heap H_1 consists of h_1 trees and H_2 consists of h_2 trees. In the worst case, we need to merge $O(h_1+h_2)$ binomial trees, each of which takes $O(1)$ time. After each link operation, the number of trees in the list decreases by 1, so there can be at most h_1+h_2-1 link operations. Therefore, the overall runtime of UNION is $O(h_1+h_2) \leq O(\log n)$.

The actual implementation of UNION is a little bit complicated as we need to consider a few different cases.

```
UNION($H_1$, $H_2$)
H = new BINOMIAL-HEAP()
H.head = MERGE($H_1$, $H_2$)          // same as MERGE in mergesort
if H.head == NIL
    return H
prev-x = NIL
x = H.head
next-x = x.sibling
while next-x ≠ NIL
    if x.degree ≠ next-x.degree or (next-x.sibling ≠ NIL and next-x.sibling.degree == x.degree)
        prev-x = x
        x = next-x
    elseif x.key ≤ next-x.key
        x.sibling = next-x.sibling
        BINOMIAL-LINK(next-x, x)
    else
        if prev-x == NIL
            H.head = next-x
        else
            prev-x.sibling = next-x
        BINOMIAL-LINK(x, next-x)
        x = next-x
    next-x = x.sibling
return H
```

14.3.3 Insertion

We can insert a new node into a binomial heap by first creating a binomial tree of degree 0, and then calling UNION(H, H') where H' is the newly created binomial tree. This is analogous to incrementing a binary number.

```
INSERT$(H,x)$
$H' =$ new BINOMIAL-HEAP$(H,x)$
$x.parent =$ NIL
$x.child =$ NIL
$x.degree == 0$
$H'.head = x$
$H =$ UNION$(H,H')$
```

The runtime of INSERT is $O(\log n)$.

14.3.4 Getting the Minimum

Search the list of roots to find the root with smallest priority. This takes $O(\log n)$ because H has $O(\log n)$ roots.

14.3.5 Extract Min

We need to find the minimum using the procedure described just now. And we remove the tree rooted at the node x with smallest priority from H. Let H' be the binomial heap consisting of the binomial trees rooted at the children of x in reverse order. We can put them back by calling MERGE(H,H'). Each step takes $O(\log n)$ so overall the algorithm still runs within a constant factor of logarithmic time.

14.3.6 Decreasing Priority

This is similar to the bubbling up procedure used in a normal min-heap. This operation also takes $O(\log n)$ time.

```
DECREASE-PRIORITY$(H,x,p)$
$x.key = p$
$y.x$
$z = y.parent$
while $z \neq$ NIL and $y.key < z.key$
    swap data in $y$ and $z$
    $y = z$
    $z = y.parent$
```

Chapter 15 Fibonacci Heaps

Chapter 16 Disjoint Sets

16.1 Disjoint Set ADT

A disjoint set is a collection of nonempty, pairwise disjoint sets of objects. Each set contains 1 special element called its representative. Sometimes, it may also be referred to as Union-Find.

A disjoint set ADT should support the following operations:

- MAKE-SET(x): Takes an object x that is not in any of the current sets and adds $\{x\}$ into the collection. The representative of the new set is x.
- FIND-SET(x): Given an element x (the pointer to x), return the representative of the set that contains x.
- UNION(x, y): Given (the pointers to) two elements $s \in S_x$ and $y \in S_y$, adds the set $S_x \cup S_y$ to the collection and removes S_x and S_y from the collection. Return the representative of the new set, which can be any element of the set. If x and y belong to the same set, UNION(x, y) has no effect.
- LINK(x, y): Same as UNION, except that x and y are the representatives of their sets.

Disjoint sets can be useful for the following applications:

- Classifying webpages and finding sets of duplicate pages;
- Reconstructing DNA sequences from small fragments (genome assembly);
- Graph algorithms such as finding connected components in a graph;
- Determining if two finite automata accepts the same language;
- Model checking for program verification.

16.2 Data Structures for Disjoint Sets

Consider a sequence of m MAKE-SET, LINK, and FIND-SET operations applied to an initially empty colleciton of sets. Let $n \leq m$ be the number of MAKE-SET operations in this sequence. Each UNION(x, y) operation can be replaced by

```
$x' =$ FIND-SET$(x)$
$y' =$ FIND-SET$(y)$
LINK$(x', y')$
```

In this section, we will take a look at a few data structures that implement the disjoint set ADT. For all the data structures described in this section, we would also need an auxillary data structure to keep track of the set/element pointers.

16.2.1 Circular Linked List

We can implement a disjiont set using a circular linked list. Each element of the disjoint set is a linked list node which contains a *next* pointer and a bit indicating whether this element is the representative of the set it belongs to. We can then represent each set in the collection by a circular singly linked list of its element nodes. In each linked list, exactly one node stores a 1 bit, indicating that node is the representative.

```
MAKE-SET($x$)
    Create a linked list where $x$ is the only node
    Set this node to 1 so that $x$ is the representative
```

Clearly, MAKE-SET runs in $O(1)$ time.

```
FIND-SET($x$)
    Follow pointers starting from $x$ until an element with bit 1 is found.
    Return the element with bit 1
```

The complexity of FIND-SET is $O(n)$ assuming all n elements are in the same set, or more generally, $O(\text{size of the set containing } x)$.

```
LINK($x, y$)
    if $x \neq y$
        Change the first element of each list to point to the second element
        of the other list
    The representative of the new set is $x$
```

LINK(x,y) would also run in $O(1)$ time.

The worst-case sequence complexity of a sequence of m MAKE-SET, LINK, and FIND-SET operations is $\Theta(mn)$. The upper bound follows from the fact that if each set has $O(n)$ elements, each operation takes $O(n)$ time and there are m operations. To obtain the matching lower bound, we perform n MAKE-SET operations, create one set by performing $n-1$ LINK operations, then finally perform $m-2n+1$ FIND-SET operations on the element that follows the representative in the list. Each of these operation takes $\Omega(2n-1)+\Omega(n(2m-2n+1)) = \Omega(mn)$ time.

16.2.2 Linked List With Back Pointers

Represent each set by a singly linked list of its elements. The first element in the list is the set representative. Each node contains a *next* pointer of a *representative* pointer.

For MAKE-SET(x), simply create a node with both the *next* and *representative* pointers pointing to itself. This takes $O(1)$ time.

For FIND-SET(x), follow the representative pointer in x, and this also takes $O(1)$ time.

```
LINK(x, y)
    Insert the list with representative y immediately after x
    Update the representative pointers of each element in list containing y to point to x
```

This will take $O(\text{length of list containing } y)$.

Now, let us consider the worst-case sequence complexity of the sequence of operations described at the beginning of the section (i.e. a sequence of m MAKE-SET, LINK, and FIND-SET operations and n is the number of MAKE-SET operations in the sequence). The worst-case sequence complexity is $\Theta(m+n^2)$. For the upper bound, there can be at most $n-1$ non-trivial LINK operations that takes $O(n)$ steps. All other operations takes $O(1)$ steps. For the lower bound, perform n MAKE-SET operations, followed by $n-1$ LINK operations. The LINK operations take in total $\sum_{i=2}^{n}(i-1) \in \Omega(n^2)$. In addition, there are m operations, each of which takes $\Omega(1)$ time. Hence, the lower bound of the worst-case sequence complexity is $\Omega(n^2+m)$.

The only expensive operation is LINK. If we call LINK on lists y of length $\Theta(n)$, then the time would be $O(mn)$. This is an overestimate since LINK(x,y) with $x \neq y$ can occur at most $n-1$ times.

16.2.3 Linked List With Back Pointers Using Union By Weight

In addition to the *back* and *representative* pointer, we store the size of each set at the head of the list.

The time to perform LINK is $O(\min\{|S_x|, |S_y|\})$. During LINK, the representative of the bigger set becomes the representative of the new set.

We claim that with union by weight, the worst-case sequence complexity is $\Omega(m+n\log n)$. For the lower bound, we can first perform $n=2^k$ MAKE-SET operations, and then repeatedly LINK sets of equal sizes. There are $n/2=2^{k-1}$ pair of lists each of length 2^0, and $n/4=2^{k-2}$ pairs of lists each of length 2^1, etc. The total cost is $\sum_{i=1}^{k}\frac{n}{2^i}\cdot 2^{i-1} = k\cdot n/2 \in \Omega(n\log n)$.

For the upper bound, consider the number of times the *representative* pointer changes for each element x. Whenever x's representative pointer is changed, $|S_x|$ is at least doubled. With $|S_x|=1$ initially, the maximum size of S_x is n. Hence, x's representative pointer changes at most by $\log_2 n$ times.

16.2.4 Trees

We can represent each set by a tree, where the root is the representative and each node has a *parent* pointer points to its parent. The root points to itself. There are no child pointers.

```
FIND-SET($x$)
    Follow the parent pointers from $x$ until reaching the root
```

This clearly takes $\Theta(T.height)$ time.

```
LINK($x,y$)
    Change the parent pointer to point to $x$
    Return $x$
```

This takes $O(1)$ time.

The worst-case sequence complexity is $\Omega(mn)$. To obtain a lower bound, we can make n sets by calling MAKE-SET n times. Then, link these sets by calling LINK $n-1$ time. Because of the implementation of LINK, this will essentially just create a linked list of length $n-1$. Finally, call FIND-SET $m-(2n-1)$ times, which in total gives us a $\Omega(mn)$ lower bound.

For the upper bound, each operation is $O(h) \leq O(n)$. With m operations, it takes $O(mn)$ in total. This is the same as a circular linked list, so this does not improve the performance. In fact, it might be better if we use a linked list with back pointers using union by height.

16.2.5 Trees With Union By Height

Store the height of each tree at the root. Representative of the taller tree becomes the new representative. If the heights are different, then the new height remains the maximum of the height of the two. If the two trees have the same height, the height of the new tree increases by 1.

> **Lemma 16.2.1** Any tree of height h has size of at least 2^h.

Proof. By induction on h. ■

> **Corollary 16.2.2** Any tree created by a sequence of n MAKE-SET and at most $n-1$ LINK operations has $O(\log n)$ height.

The upper bound on the worst-case sequence complexity is $O(m\log n)$ since the worst-case for FIND-SET is $O(\log n)$ and there are m operations in total. The lower bound can be achieved by constructing a binomial tree and then perform FIND-SET on a deepest node. So the worst-case sequence complexity is $\Theta(m \log n)$.

16.2.6 Trees With Path Compression

During FIND-SET(x), each node visited on the path from x to the root is made a child of the root. This at most doubles the running time of FIND-SET.

Claim. The worst-case sequence complexity is

$$\Theta\left(n + f \cdot \left(\frac{1 + \log n}{\log(2 + \frac{f}{n})}\right)\right)$$

where f is the number of FIND-SET operations.

16.2.7 Additional Implementations

Trees with union by weight and path compression.

Trees with union by height and path compression. The problem with this is that we cannot trivially maintain height information in the data structure when path compression is performed because path compression can change the height.

16.2.8 Trees With Union By Rank and Path Compression

In this implementation, we store the rank of each tree instead of its height. The rank can be thought as a "pseudoheight" that denotes the height of the tree if there is no path compression.

```
LINK$(x, y)$
if RANK$(x)$ == RANK$(y)$
    $y.parent = x$
    RANK$(x) =$ RANK$(x) + 1$
elseif RANK$(x) >$ RANK$(y)$
    $y.parent = x$
else
    $x.parent = y$
```

The rank of any tree created by a sequence of MAKE-SET and LINK operations is equal to its height and hence is in $O(\log n)$.

FIND-SET operations with path compression reduce the height of the tree but do not change the ranks. Maintain the rank of a tree takes fewer bits than maintaining the weight of a tree.

```
FIND-SET($x$)
if $x \neq x.p$
    $x.p = $ FIND-SET($x.p$)
return $x.p$
```

Finally, let us examine the sequence complexity of a sequence of operations performed on a disjoint set implemented using union-by-rank with path compression.

Definition 16.2.1 — Iterated Logarithm.

$$\log^*(n) = \min\{i \geq 0 \mid \log^{(i)}(n) \leq 1\}$$

where $\log^{(0)}(n) = n$, $\log^{(1)}(n) = \log n$, and $\log^{(i+1)}(n) = \log(\log^{(i)})n$. Intuitively, $\log^*$ is the number of times we need to apply log to n until we get a number of at most 1. For most practical purposes, $\log^*$ with base 2 has at most 5 (e.g. $\log^* 2^{65536} = 5$).

Theorem 16.2.3 The worst-case sequence complexity of a sequence of m MAKE-SET operations and FIND-SET operations with n MAKE-SET operations performed on a disjoint set implemented using union-by-rank with path compression is $O(m \log^* n)$.

Theorem 16.2.4 — A Tight Bound on Union-by-Rank With Path Compression. The worst-case sequence complexity of a disjoint set implemented using union-by-rank with path compression is $\Theta(m\alpha(n,m))$ where α is the inverse Ackermann function.

16.3 Amortized Analysis of Disjoint Set

The following proof was adopted from *Introduction to Algorithms (1st Edition)* by Cormen, Leiserson, and Rivest. The proof was first presented by Hopcroft and Ullman in the paper *Set Merging Algorithms* in SIAM J. Compt. in 1973.

Lemma 16.3.1 Any tree of rank r has size of at least 2^r.

Proof. By induction on the number of LINK operations. ■

Lemma 16.3.2 If p is the parent of x, then $rank(p) > rank(x)$.

Proof. By construction and the fact that rank is non-decreasing. The non-decreasing property of rank can be shown using induction on the number of operations. ■

Lemma 16.3.3 For any integer $r \geq 0$, there are at most $n/2^r$ nodes of rank r.

Proof. Fix $r \geq 0$. The trees rooted at any two nodes with rank r are disjoint. By Lemma 16.3.1, each of these trees contains at least 2^r nodes. Since there are at most n nodes in total, there are at most $n/2^r$ nodes of rank r. ■

Corollary 16.3.4 Every node has rank at most $\lfloor \lg n \rfloor$

Proof. Let $r > \lg n$, then there are at most $n/2^r < 1$ nodes of rank r. ■

Now we proceed to prove Theorem.

Proof. We partition the ranks into $\log^* n$ blocks where rank r goes into block $\log^* r$.

For example,

- Block 0: rank 0 and 1
- Block 1: rank 2
- Block 2: rank $3 = 2^1 + 1$ to $4 = 2^2$
- Block 3: rank $5 = 2^2 + 1$ to $16 = 2^{2^2}$
- Block j: rank $T(j-1)+1$ to $T(j)$ where $T(j)$ denotes a height j tower of 2.

Note that the last block is Block $\log^*(\log n) = \log^* n - 1$.

The analysis uses the aggregate method. Consider the charges corresponding to the actual cost of each operation. MAKE-SET and LINK are each associated with one charge since both are $O(1)$ operation.

For FIND-SET, we associate it with two types of charges: block charge and path charge. Suppose that the FIND-SET operation starts at a node x_0 and the path it takes to the root x_l consists of nodes $x_0, x_1, \ldots, x_l$, where each of x_i is the parent of x_{i-1}. By Lemma 16.3.2, we know that $rank(x_0) < rank(x_1) < \cdots < rank(x_l)$.

There is one block charge for the last node in the sequence $x_0 \ldots x_l$ whose rank is in Block j for $j \geq 0$. Additionally, give a block charge to the child of the root r_{l-1}, it it does not already have a block charge. This will cover the cost of calling FIND-SET on the child of root.

All other nodes have a path charge. Note that by definition, x_l has a block charge. For example, if we have the find path $x_0, x_1, x_2, x_3, x_4, x_5$ with ranks $2, 3, 4, 6, 8, 9$, respectively, the block charges and path charges are assigned as follows (□ represents a block charge, $\rightsquigarrow$ represents a path charge).

Node			□ x_0	$\rightsquigarrow$ x_1	□ x_2		$\rightsquigarrow$ x_3		□ x_4	□ x_5		
Rank	0	1	2	3	4	5	6	7	8	9	$\cdots$	16
Block	Block 0		Block 1	Block 2		Block 3						

Node that the total number of charges is still equal to the total number of nodes visited in all FIND-SET operations in the sequence of m operations.

In each FIND-SET operation, there is at most 1 block charge per block, plus one for the child of root. There are $\log^* n$ blocks, so there are at most $m[\log^*(n)+1] \in O(m \log^* n)$ block charges. Whenever we call FIND-SET on a child of the root, the cost is paid for by the block charges.

If x is not a root or a child of the root, then $rank(x)$ does not change during path compression. But it gains a parent of a larger rank. This implies the observation that if x is not root or child of the root and x gets a block charge in some FIND-SET operation, then it gets a block charge in all subsequent FIND-SET operations involving x. Once a node gets a block charge, it will no longer receive path charges.

Suppose that $rank(x)$ is in Block j, we want to know how many times x can get a path charge. Each time FIND-SET is called on x, x gets a parent with larger rank. Eventually, x will get a parent with rank in the next block and hence gets a block charge (x now sits on the boundary between two blocks). Recall that ranks contained in each block range from $T(j-1)+1$ to $T(j)$. It follows that when $j = 0, 2$, x get at most 1 path charge; when $j = 1$, x gets no block charge; and for all $j > 2$, at most $T(j) - T(j-1) - 1$ path charges. Once x gets the final path charge, it will gets a block charge and no more path charge in subsequent calls.

Next, we bound the number of nodes that have ranks in Block j in order to derive a bound on the number of path charges. Let $N(j)$ be the number of nodes whose ranks are in Block j. By Lemma 16.3.3

$$N(j) \le \sum_{r=T(j-1)+1}^{T(j)} \frac{n}{2^r}$$

For $j \ge 1$,

$$\begin{aligned} N(j) &\le \frac{n}{2^{T(j-1)+1}} \sum_{r=0}^{T(j)-T(j-1)-1} \frac{1}{2^r} \\ &< \frac{n}{2^{T(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\ &= \frac{n}{2^{T(j-1)}} \\ &= \frac{n}{T(j)} \end{aligned}$$

Finally, we can conclude that the total number path charges by summing over all blocks the product of the maximum number of nodes with ranks in that block and the maximum number of path charges per node in that block.

$$\begin{aligned}\sum_{j=0}^{\log^* n-1} N(j)\cdot(T(j)-T(j-1)-1) &\leq \sum_{j=0}^{\log^* n-1} N(j)\cdot T(j)\\ &\leq \sum_{j=0}^{\log^* n-1} \frac{n}{T(j)}\cdot T(j)\\ &= n\log^* n \in O(m\log^* n)\end{aligned}$$

■

A proof of the tight bound is presented in CLRS Chapter 21.4 using the potential method.

16.3.1 An Alternative Proof

I personally found this proof using block charges and path charges rather counterintuitive. Here, I present a more intuitive and straightforward proof, first discovered by Seidel and Sharir in their paper *Top-down Analysis of Path Compression*, published in SIAM J. Compt. in 2005.

V

Lower Bounds

17 Decision Trees **109**

17.1 Lower Bounds
17.2 Comparison Model
17.3 Comparison Tree

18 Information Theory **113**

18.1 Height of Trees
18.2 Lower Bounds for Searching
18.3 Sorting In Linear Time

19 Adversary Arguments **115**

19.1 Guess a Number Game
19.2 Proving Problem Lower Bound Using Adversary Argument

20 Reduction **117**

Chapter 17 Decision Trees

17.1 Lower Bounds

So far, we have been talking almost exclusively about how we can use different algorithms and data structures to solve certain problems as fast as possible. In this part, we will focus on proving certain problems cannot be solved as quickly as we might want. In other words, there is a limit to how good we can do. For example, in a comparison model, sorting can only be achieved at best in $\Omega(n \log n)$ time in the worst case.

Definition 17.1.1 Let P be a problem. The worst case complexity of problem P is

$$C(P) = \min\{C_A \mid A \text{ is an algorithm that solves } P\}$$

where $C_A : \mathbb{N} \to \mathbb{N}$ is the worst case complexity of algorithm A.

We can prove the upper-bound of the complexity of a problem by giving a specific algorithm A that solves Π, and faster algorithms give us smaller (tighter and better) upper bounds.

However, to prove that a problem has a certain lower bound, it is not enough to give an algorithm and say that we cannot do better. We have to show that every algorithm that solves Π has a worst-case running time $\Omega(f(n))$, or equivalently, that there is no algorithm that solves Π that runs in $o(f(n))$ time. To be more specific, we need to specify what kinds of algorithms we want to consider, which is formally known as model of computation. For example, the comparison model is one model that is used to solve sorting and searching problems.

Let $\mathcal{A}$ be a class of algorithms. The worst-case complexity of problem P in class $\mathcal{A}$ is

$$C(P) = \min\{C_A \mid A \in \mathcal{A} \text{ is an algorithm that solves } P\}$$

$\mathcal{A} \subseteq \mathcal{A}'$ implies that a lower bound on the worst-case complexity of P in class $\mathcal{A}'$ is also a lower bound on the worst-case complexity of P in class $\mathcal{A}$. In other words, lower bounds in more powerful models are better.

17.2 Comparison Model

In the comparison model, we consider all input items as black boxes, or more precisely, ADTs. The only operations allowed on the items are comparisons: $<, \leq, >, \geq, =$. Most searching and sorting algorithms we have been looking at so far use the comparison model: heap sort, merge sort, binary search and binary search tree, etc. In the comparison model, we count the number of comparisons and define it as the time cost of the algorithm.

17.3 Comparison Tree

17.3.1 Intuition

Any comparison algorithm can be viewed as a tree of all possible comparisons, the outcomes of the comparisons, and the resulting answer. This tree is called a decision tree. There is a different tree for each different input size.

For any particular n,

- *internal node* corresponds to a decision in the algorithm (in this case, comparisons)
- *edge* corresponds to a possible outcome of the comparison denoted by the node
- *leaf* corresponds to a possible answer (output) of the problem
- *root-to-leaf path* corresponds to an execution of the algorithm
- *length of the root-to-leaf path* corresponds to the time cost of the execution associated with that path
- *height of the tree* (or depth of the deepest leaf) corresponds to the worst-case running time.

17.3.2 Decision Tree for Searching

Consider a comparison tree for the binary search algorithm. An example is shown below for binary search for x in $A[1 \dots 3]$ using only $\leq$ comparisons and outputs i such that $x = A[i]$ or 0 if no such element exists.

The average case complexity the algorithm is the expected path length, that is the weighted average depths of the leaves.

$$\sum_{\ell} (\ell.depth) \cdot \text{Prob}[\text{input leads to } \ell]$$

Another example is for searching in $[2, 3, 5, 7, 11, 13, 17, 19]$ using $<$ and $=$ comparisons.

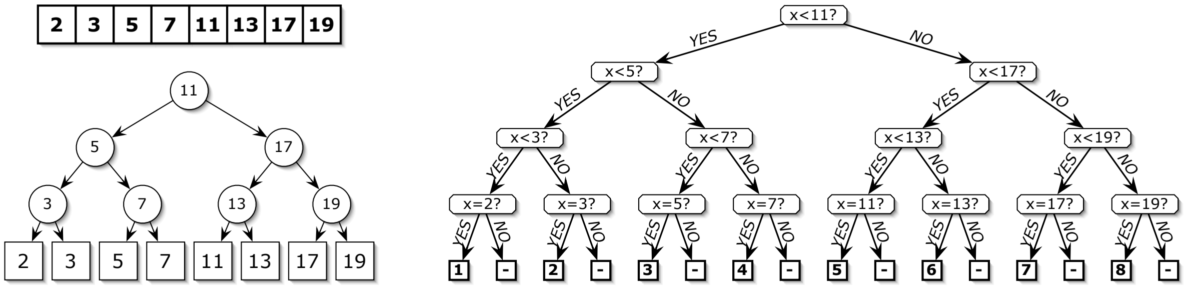

2 3 5 7 11 13 17 19
11
5
17
3
7
13
19
2
3
5
7
11
13
17
19
x<11?
YES
NO
x<5?
x<17?
x<3?
x<7?
x<13?
x<19?
x=2?
x=3?
x=5?
x=7?
x=11?
x=13?
x=17?
x=19?
1
2
3
4
5
6
7
8
-

Figure 17.1: Decision tree for searching the array containing $[2, 3, 5, 7, 9, 11, 13, 17, 19]$.

Chapter 18 Information Theory

18.1 Height of Trees

A t-ary tree of height h has at most t^h leaves. A t-ary tree with L leaves has height at least $\lceil \log_t L \rceil$.

Therefore, the worst-case number of t-way comparisons performed by a comparison tree T that solves P is at least $\lceil \log_t(\text{number of leaves in } T) \rceil$.

$$C(P) \geq \min\{\lceil \log_t(\text{number of leaves in } T) \rceil \mid T \text{ solves } P\}$$

If P has at least m different possible outputs, then every comparison tree that solves P has at least m leaves, and it follows that every comparison tree that solves P has height of at least $\lceil \log_t m \rceil$.

18.2 Lower Bounds for Searching

18.2.1 Searching Sorted List

Consider the problem of searchign a sorted list using only $\leq$ comparisons. The input for this problem is $A[1 \ldots n]$ where $A[1] \leq \cdots \leq A[n]$ and a search key x. The output is i such that $x = A[i]$ or 0 if does not exist.

Naturally, there are $n+1$ possible outputs. The information thoery lower bound for this problem is then

$$C(P) \geq \lceil \log_2(n+1) \rceil$$

Now let us consider the same problem, but with a slightly modified set of outputs. Instead of outputing 0, the lagorithm returns $-\infty$ if $x < A[i]$, ∞ if $x > A[i]$, or $(i, i+1)$ if $A[i] < x < A[i+1]$. Call this problem P'.

In this case, there are $2n+1$ different possible outputs. So the information theory lower bound gives $C(P') \geq \lceil \log_2(2n+1) \rceil$.

Any comparison tree that solves P' can be converted into a comparison tree by relabeling leaves. Conversely, any comparison tree using $\leq$ comparisons that solves P can be converted into a comparison tree that solves P' by relabeling leaves.

Lemma 18.2.1 In any comparison tree that solves P using $\leq$ comparisons, for any array A, if search keys $y < z$ go to the same leaf, then search key u also goes to that leaf, for $y < u < z$.

Proof. Consider the comparison $x \leq A[i]$. If this is true for $x = z$ then it is also true for $x = u$, since $u < z \leq A[i]$. If this test is false for $x = y$, then it is also false for $x = u$, since $u > y > A[i]$. ■

Hence $C(P) \geq \lceil \log_2(2n+1) \rceil$ where P is the problem for searching a sorted list using only $\leq$ comparisons. $\lceil \log_2(2n+1) \rceil$ is also an upper bound (consider binary search). So the lower bound is tight.

18.2.2 Searching Unsorted List

Now let's consider the problem U of searching an unsorted list using only $=$ comparisons. The input is an array $A[1 \ldots n]$ and a search key x. It should output i such that $x = A[i]$ or 0 if no such i exists.

There are $n+1$ possible outputs, so by information theory lower bound, $C(U) \geq \lceil \log_2(n+1) \rceil$.

As we will show later, this lower bound is not tight.

18.3 Sorting In Linear Time

Chapter 19 Adversary Arguments

19.1 Guess a Number Game

19.1.1 Battleship

19.2 Proving Problem Lower Bound Using Adversary Argument

To prove a lower bound $L(n)$ for the time complexity of problem P using an adversary argument, show that, for each algorithm A that solves P and for each input size n, an adversary can choose an input of size n (not necessarily in advance) on which algorithm A must take at least $L(n)$ steps.

This is usually done using proof by contradiction.

Template 19.1 — Lower Bound Proof Using Adversary Argument. We want to prove that the lower bound on the time complexity of problem P is $L(n)$.

Suppose there exists a comparison tree of height lower than $L(n)$ (if comparison-based problem). More generally, suppose there is an algorithm that solves the problem in fewer than $L(n)$ steps.

Devise an adversary strategy that gives answers consistent with its previous answers so that the algorithm is forced to take as many steps as possible. Choose and update input based on the adversary strategy and show that on such input, the comparison tree/algorithm gives the wrong answer. This contradicts the correctness of the comparison tree/algorithm.

Chapter 20 Reduction

VI

Graphs

21 Graph Representation 121
21.1 Graph
21.2 Operations on Graphs
21.3 Data Structures for Representing Graph
21.4 Incidence Matrix
21.5 Time Complexity

22 Breadth-First Search 125
22.1 Breadth-First Search (BFS)

23 Depth-First Search 129
23.1 Depth-First Search (DFS)
23.2 Applications of BFS/DFS

24 Minimum Spanning Trees 133
24.1 Spanning Forest and Spanning Tree
24.2 Computing the Minimum Spanning Tree
24.3 Kruskal's Algorithms
24.4 Prim's Algorithm

25 Single-Source Shortest Path Algorithms 139
25.1 Shortest Path
25.2 Dijkstra's Algorithm

Chapter 21 Graph Representation

21.1 Graph

Definition 21.1.1 — Undirected Graph. An undirected graph is the ordered pair $G = (V, E)$ where V is the set of nodes/vertices, and E is the set of edges defined by $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$.

Definition 21.1.2 — Directed Graph. A directed graph is the ordered pair $G = (V, E)$ where E is the set of ordered pairs $E \subseteq V \times V$.

We use $n = |V|$ to denote the number of vertices, and $m = |E|$ to denote the number of edges in a graph. For an undirected graph, $\binom{|E| \leq}{n2}$ and $|E| \leq n^2$ for directed graph.

Definition 21.1.3 — Neighbors and Degrees. For any edge uv in an undirected graph, we call u neighbor of v and vice versa, and we say that u and v are adjacent. The degree of a node is its number of neighbors.

In directed graphs, for every edge $u \to v$, we call u a predecessor of v, and we call v a successor of u. The in-degree of a vertex is its number of predecessors; the out-degree is its number of sucessors.

Definition 21.1.4 — Subgraphs. A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A proper subgraph of G is any subgraph that is not G itself.

21.1.1 Other Types of Graphs

A **simple graph** has no multiple edges or self loops.

A **hypergraph** is a graph consists of vertices and hyperedges. A hyperedge is a nonempty set of vertices.

A **weighted graph** has a weight function $w: E \to \mathbb{R}$ that assign a weight to each edge. We can similarly define a weight function for the vertices.

An **subgraph** G' of G **induced** by V' is the graph (V', E') such that $E' = \{\{u, v\} \in E \mid u, v \in V'\}$.

21.2 Operations on Graphs

- Find the shortest path between $u, v \in V$.
- Cycle detection.
- Compute degree of a vertex ($\deg(v)$ is the number of neighbors of a vertex).
- Add/Delete vertices.
- Add/Delete edges.
- Check if there is a path between u and v.
- Topological sort of a directed acyclic graph (DAG)
- Find a clique (a set of nodes C such that $\forall u, v \in C. u \neq v$, $\{u, v\} \in E$)
- Color a graph.
- Find minimal spanning tree.
- Find all nodes u that are connected to some node v.
- Find all neighbors of v.
- Find all connected components in an undirected graph.
- Check if G is induced subgraph.
- Find maximal matching.
- Check isomorphism.
- Compute flows (minimum and maximum flow) in a weighted directed graph.

21.3 Data Structures for Representing Graph

21.3.1 List of Edges

We can simply store a graph as a list of all edges in the graph. It takes $O(m)$ words, each of $O(\lg n)$ bits. The list can either be sorted or unsorted.

21.3.2 Adjacency Matrix

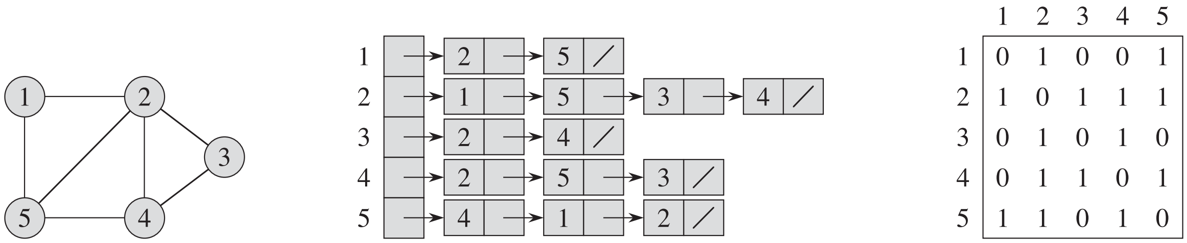
Definition 21.3.1 — Adjacency Matrix. An adjacency matrix is a matrix whose rows and columns are indexed by V where $A[u, v] = 1$ if and only if $\{u, v\} \in E$ for undirected graph or $(u, v) \in E$ for directed graph.

This representation takes $O(n^2)$ space. It is efficient if the grraph is dense (i.e. $m \in \Omega(n^2)$).

21.3.3 Adjacency List

Definition 21.3.2 — Adjacency List. The adjacency-list representation of a graph $G = (V, E)$ consists of an arrray *Adj* of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list *Adj[u]* contains all the vertices v such that there is an edge $(u, v) \in E$. That is, *Adj*[u] contains all the vertices adjacent to u in G.

It requires $O(n+m)$ words to store an adjacency list. There are in total $2m$ entries in these lists, and each word is $O(\lg n)$ bits. This representation is efficient if the graph is sparse (i.e. $m \in O(n)$ or $m \in O(n \log n)$).



(a) (b) (c)

Figure 21.1: Representations of undirected graph: (a) the graph; (b) adjacency list of the graph; (c) adjacency matrix of the graph.

21.4 Incidence Matrix

$M[v,e] = 0$ if v is not incident to e. For undirected graph, $M[v,e] = 1$ if v is incident to e. For directed graph, $M[v,e] = 1$ if v is the destination of e and -1 if v is the source of e. A self-loop is typically represented using a special value.

We can store the weight in the matrix if $w(v) \in \mathbb{R}^+$.

21.5 Time Complexity

The time complexity of operations depend on the representation. For example, consider the problem of checking whether or not $\{u,v\} \in E$.

- Adjacency matrix: $O(1)$
- Adjacency list: $O(n)$ or $O(\deg(u))$
- List of edges: $O(m)$ or $O(\log n)$
- Incidence matrix: $O(m)$

For the problem of finding the neighbors

- Adjacency matrix: $O(n)$
- Adjacency list: $O(1)$
- List of edges: $O(m)$
- Incidence matri: $O(m)$

Theorem 21.5.1 — Rivest-Vuiellemin Theorem. Any nontrivial monotone graph property that is invariant under isomorphism requires $\Omega(n^2)$ adjacency matrix probes. Monotone means the property does not change from True to False when more edges are added.

Chapter 22 Breadth-First Search

22.1 Breadth-First Search (BFS)

22.1.1 BFS Algorithm

Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth-first search systematically explores the edges of G to discover every vertex that is reachable from s. Breadth-first search expands the frontier between disvoered and undiscovered vertices uniformly across the breadth of the frontier. The algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k+1$.

BFS visits all the vertices in a connected undirected graph in order of their distance from the start vertex s. As we visit the nodes, we can also determine $\delta(s, v)$ which is the distance from s to v, that is, the number of edges on the shortest path from s to v.

```
BFS($s$)
$Q = \text{QUEUE}(\{\});\ d = []$
for $v \in V$
    $d[v] = 0$
    label $v$ as unvisited
visit $s$
$\pi[s] = \text{NIL}$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    $u = \text{HEAD}(Q)$
    for out neighbor $v$ of $u$
        if $v$ is unvisited
            ENQUEUE$(Q, v)$
            $d[v] = d[u] + 1$
            $\pi[v] = u$
    DEQUEUE$(Q)$
```

We keep track of the shortest length from s to each vertex in the array d. We will prove the correctness of BFS for calculating shortes path in a later subsection.

If G is not connected, this algorithm visits the connected component of G containing s. If G is directed, this visits all vertices reachable from s by a directed path.

22.1.2 Edge Classification

We can create a tree, known as BFS tree, by running the BFS algorithm on a graph. Each level of the BFS tree contains vertices reachable with one step from their corresponding parents at the previous level.

Using the BFS tree, we can classify edges in a graph into three categories:

- Tree edge (green): $\{v, v.parent\}$ or $(v.parent, v)$ if directed.
- Cross edge (orange): $\{u,v\}$ or (u,v) if directed where u is not an ancestor of descendant of v in the BFS tree.
- Back edge (red): edge from a node to one of its ancestors.

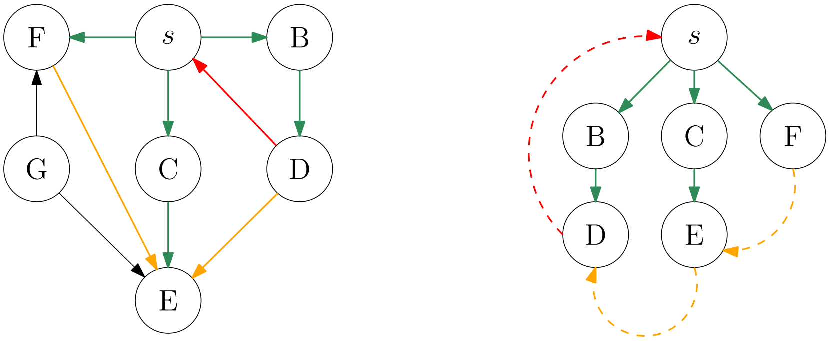


Figure 22.1: A graph and its corresponding BFS tree. Edges are colored based on their classification.

Lemma 22.1.1 If G is undirected, then all edges are tree edges or cross edges.

Proof. Suppose that $\{u,v\}$ is a back edge discovered when u is at the head of Q.

Since v is an ancestor of u, it was visited and hence enqueued before u. So v was the head of Q before u. Since $\{u,v\} \in E$, u was visited when v was at the head of Q.This means that $\{u,v\}$ is a tree edge. ■

22.1.3 Correctness of BFS

Claim. If G is connected, then at the end of BFS, $d[v] = \delta(s,v)$ for all $v \in V$.

Observation. $d[v] = 0$ if and only if $v = s$. If G is directed, $(\pi[v], v) \in E$. If G is undirected, $\{\pi[v], v\} \in E$. $d[v] = d[\pi[v]] + 1$.

From this observation, we have the following lemma.

Lemma 22.1.2 At the end of the BFS, $d[v] \geq \delta(s,v)$ for all nodes reachable from s.

Proof. $d[v]$ is the length of the path from s to v in the BFS tre since $d[v] = d[\pi[v]] + 1$ for all nodes $v \neq s$ reachable from s.

All these edges are in E, so $\delta(s,v) \leq d[v]$. ■

Lemma 22.1.3 Let $s = v_1, v_2, \cdots, v_n$ be the nodes in the order they are visited (enqueued). Then, $0 = d[v_1] \leq \cdots \leq d[v_n]$.

Proof. Suppose not. Let $j > 1$ be the smallest integer such that $d[v_{j-1}] > d[v_j]$.

Let $v_h = \pi[v_{j-1}]$ where $h < j-1$, and let $v_i = \pi[v_j]$ where $i < j$.

$d[v_{j-1}] = 1 + d[v_h]$ and $d[v_j] = 1 + d[v_i]$. Since $d[v_{j-1}] > d[v_j]$, then $d[v_h] > d[v_i]$. Since $h, i \leq j-1$ and $d[v_1] \leq \cdots \leq d[v_{j-1}]$, it follows that $i < h$. So, $d[v_j] \leq d[v_i]$ and $d[v_h] \leq d[v_{j-1}]$.

This means v_i is enqueued and hence dequeued before v_h, which then implies that v_j was enqueued before v_{j-1}. This is a contradiction. ■

Alternatively, this lemma can also be proved using induction on the number of queue operations (see CLRS pp 599).

Finally, using the lemmas that we have proved, we can show that BFS correctly calculates the shortest path from s to all vertices reachable from s.

Theorem 22.1.4 At the end of $\mathrm{BFS}(s)$, $d[v] = \delta(s,v)$ for all nodes v reachable from s.

Proof. Consider the shortest path (of length) $\delta(s,v)$ from s to v in G. Let $\{u,v\}$ be the last edge in this path where u is the parent of v on that path. $\delta(s,u) = \delta(s,v) - 1$. $\delta(s,s) = 0 = d[s]$.

Let v be a vertex where the theorem is not ture. That is, let v be a vertex with minimum value of $\delta(s,v)$ such that $d[v] > \delta(s,v)$. Let u be the predecessor of v on the shortest path from s to v. Then, we know that $\delta(s,u) = d[u]$ and $\delta(s,v) = d[v] = d[\pi[v]] + 1$. So, $d[v] > d[u] + 1$.

Consider the two cases: If v was unvisited, then $d[v] = d[u] + 1$ after the execution, which is a contradiction. If v has been visited, there exists a $\pi[v]$ in the BFS tree that was dequeued and thus enqueued before u. By Lemma, $d[\pi[v]] \leq d[u]$. Hence $d[v] = d[\pi[v]] + 1 \leq d[u] + 1$, which is a contradiction. No such v exists. Therefore, the theorem holds. ■

22.1.4 Running Time of BFS

If represented using an adjacency list, the BFS algorithm is $O(n+m)$. Each node v is enqueued at most once. Its adjacency list is examined at most once. If represnted using an adjacency matrix, it requires $\Theta(n^2)$ time.

Chapter 23 Depth-First Search

23.1 Depth-First Search (DFS)

23.1.1 DFS Algorithms

```
DFS($s$)
$S = $ Stack($\{\}$)
Push($S, s$)
while $S \neq \emptyset$
    $u = $ Pop($S$)
    if $u$ is unvisited
        visit $u$
        for out neighbor $v$ of $u$
            if $v$ is unvisited
                Push($S, v$)
```

An equivalent recursive version of the algorithm is presented in CLRS. t is initialized to 0 prior to the first call to DFS-Visit.

```
DFS-Visit($s$)
visit $u$
$t = t + 1$;  $d[u] = t$
for out neighbor $v$ of $u$
    if $v$ is unvisited
        DFS-Visit($v$)
$t = t + 1$;  $f[u] = t$
DFS-Vist($s$)
```

The running time of the DFS algorithm is $O(m+n)$ if the graph is represented as an adjacency list, and $O(n^2)$ if represented as an adjacency matrix.

In DFS, we can keep track of the following two properties of each vertex:

- $d[v]$: discovery time of vertex v
- $f[v]$: finish time of vertex v

Theorem 23.1.1 — Parenthesis Theorem. In DFS of G, for all $u, v \in V$, either $(d[u], f[u])$ and $(d[v], f[v])$ are disjoint, or one is contained in the other. That is

$$d[u] < f[u] < d[v] < f[v] \text{ OR } d[u] < d[v] < f[v] < f[u] \text{ OR } d[v] < d[u] < f[u] < f[v]$$

Proof. Without loss of generality, suppose $d[u] < d[v]$. If $f[u] < d[v]$, then $d[u] < f[u] < d[v] < f[v]$. So suppose $d[v] < f[u]$, then v is a descendant of u in the DFS tree, so DFS-VISIT(v) finishes before DFS-VISIT(u), so $f[v] < f[u]$. Hence, $d[u] < d[v] < f[v] < f[u]$. The other case when $d[u] > d[v]$ is the mirror of this case. ■

Note that if a vertex v is visited after u, it will have a larger discovery time ($d[u] < d[v]$) but a smaller finish time ($f[v] < f[u]$) since we will not be able to finish u until we have visited all its descendants and travel back up from the recursion.

23.1.2 Edge Classification

Similar to BFS, we can construct a DFS tree and classify the edges into four categories by running the DFS algorithm on a graph.

- Tree edges (green): from $\pi(v)$ to v. $d[u] < d[v] < f[v] < f[u]$.
- Back edges (red): from a node to an ancestor. $d[v] < d[u] < f[u] < f[v]$.
- Forward edges (blue): a non-tree edge from a node to a descendant. $d[u] < d[v] < f[v] < f[u]$.
- Cross edges (orange): between two nodes of which is a descendant of the other. $d[v] < f[v] < d[u] < f[u]$.

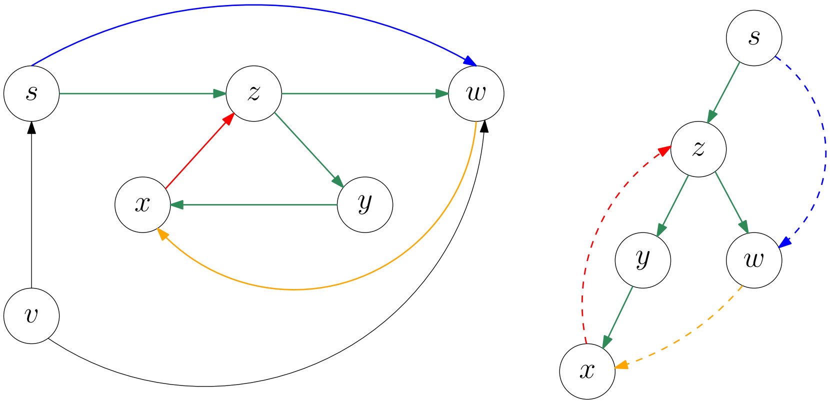


Figure 23.1: Example of a DFS on a graph.

Theorem 23.1.2 In an undirected graph, all edges are either tree edges or back edges (we do not differentiate between forward edge and back edge because the graph is undirected).

Proof. $\{u, v\} \in E$ and $d[u] < d[v]$. Since v is visited before DFS-VIST(u) terminates, so v is a descendant of u. So $\{u, v\}$ is a back edge or a tree edge. ■

23.2 Applications of BFS/DFS

- Broadcast information in a connected network.
- Detect if a graph contains a cycle. Every back edge is part of a cycle. If an undirected graph has no back edges. If all edges are tree edges, the graph is a tree and acylcic.

23.2.1 Cycle Detection

Lemma 23.2.1 A directed graph is acyclic if and only if a DFS of the graph has no back edges.

Proof. If there is a back edge, the graph has a cycle. Suppose the graph has a cycle. Before we finish the DFS of u, there is a predecessor v of u that is part of the cycle. $d[u] < d[v] < f[v] < f[u]$. ■

Lemma 23.2.2 An undirected graph contains a cycle if and only if its BFS tree contains a cross edge.

23.2.2 Determining If a Graph is Bipartite

Definition 23.2.1 A graph $G = (V, E)$ is a bipartite if and only if $V = A \cup B$ where $A, B \neq \emptyset$ and $A \cap B = \emptyset$ such that every edge $e \in E$ has 1 endpoint in A and the other endpoint is in B.

Do BFS, record the parity of the distance of each node from the source.

23.2.3 Topological Sort

The topological sort of a directed acyclic graph (DAG) $G = (V, E)$. Ordering of the vertices of V such that for if $(u, v) \in E$, then $u < v$. That is, u is before v in the ordering.

```
TOPOLOGICAL-SORT($G$)
$L = \{\}$
DFS($G$)
    after computing $f[v]$ for vertex $v$, insert $v$ at the front of $L$
return $L$
```

The algorithm sorts vertices in decreasing order of finish time.

Theorem 23.2.3 TOPOLOGICAL-SORT is correct.

Proof. Consider any edge $(u,v) \in E$, since G is acyclic, it has no back edges.

If (u,v) is a tree edge or forward edge,

$$d[u] < d[v] < f[v] < f[u]$$

which is equivalent to $u < v$ in finish time.

If (u,v) is a cross edge.

$$d[v] < f[v] < d[u] < f[u]$$

v is discovered and finished before the discovery and finish of u, so $u < v$ in finish time. ■

Chapter 24 Minimum Spanning Trees

24.1 Spanning Forest and Spanning Tree

A tree (V,T) is a connected graph with no cycles (connected acyclic graph). Every node is reachable from every other node in exactly one way. If (V,T) is connected, then (V,T) is acyclic if and only if $|T| = |V| - 1$.

Definition 24.1.1 — Spanning Forest and Spanning Tree. Let $G = (V,E)$ be an undirected graph. A spanning forest of $G = (V,E)$ is an acyclic graph (V,F) with $F \subseteq E$. It is a set of tree on disjoint sets of vertices.

A spanning tree of G is a connected spanning forest of G. It contains $|V| - 1$ edges.

Minimum spanning tree concerns weighted graphs. Consider a weight function $w : E \to \mathbb{R}$. A minimum spanning tree is the spanning tree (V,T) of G is a spanning tree of G that minimizes the weight of T where $w(T) = \sum\{w(e) \mid e \in T\}$.

24.2 Computing the Minimum Spanning Tree

Below is a generic algorithm (template) for finding a minimum spanning tree of a given graph $G = (V,E)$.

```
MST($G = (V,E), w$)
A = $\emptyset$
while $|A| < n - 1$
    find an edge $e \in E$ such that there is an MST containing $A \cup \{e\}$
    $A = A \cup \{e\}$
    return $(V,A)$
```

The algorithm maintains the loop invariant that there is an MST of G that contains the edges in A. On termination, $|A| = n - 1$ and then (V,A) is a minimum spanning tree since spanning trees of G have $n - 1$ edges.

But then, the question becomes: how to find an edge e that there is an MST containing $A \cup \{e\}$?

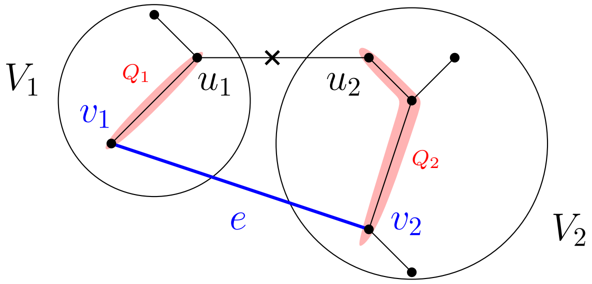
V_1, V_2 is a partition of V if $V_1 \cup V_2 = V$ and $V_1 \cap V_2 \neq \emptyset$. A **light edge** crossing the partition is an edge $\{v_1, v_2\} \in E$ with minimum weight such that $v_1 \in V_1$ and $v_2 \in V_2$.

$$w(\{v_1, v_2\}) = \min\{w(u_1, u_2) \mid u_1 \in V_1,\, u_2 \in V_2,\, \{u_1, u_2\} \in E\}$$

Theorem 24.2.1 Suppose that $A \subseteq E$ and there is an MST (V, T) of G such that $A \subseteq T$. Let V_1, V_2 be a partition such that for all $\{u, v\} \in A$, either $u, v \in V_1$ or $u, v \in V_2$. In other words, no edge in A crosses the partition V_1 and V_2. Let e be a light edge crossing this partition. Then, there is an MST that contains $A \cup \{e\}$.

Proof. If $e \in T$, the claim is true. So, suppose $e \notin T$. Without loss of generality, suppose $e = \{v_1, v_2\}$ where $v_1 \in V_1$ while $v_2 \in V_2$. Since G is connected, there is a simple path Q between v_1 and v_2 in T.

There is an edge $\{u_1, u_2\}$ on the path Q such that $u_1 \in V_1$ and $u_2 \in V_2$. Note that, by assumption, $\{u_1, u_2\} \notin A$. Let Q_1 be the part of the path Q from u_1 to v_1, and let Q_2 be the part of the path Q from u_2 to v_2, such that $Q = Q_1\{v_1, v_2\}Q_2$.



Let $T' = T \cup e - \{u_1, u_2\}$. $|T'| = |T| = n - 1$ and $A \cup \{\{v_1, v_2\}\} \subseteq T'$.

Claim: (V, T') is connected and a tree.

Let $x, y \in V$. Since (V, T) is connected, there is a path P in (V, T) connecting x and y. If P does not contain $\{u_1, u_2\}$ and u_1 comes before u_2 in P. Let P_1 be the subpath of P from $x \to u_2$, and let P_2 be the subpath of P from $u_2 \to y$ so that $P = P_1\{u_1, u_2\}P_2$. Then, $P_1, Q_1, \{v_1, v_2\}, Q_2, P_2$ is a path from x to y in (V, T'). Hence by generalization, (V, T') is connected. $|T'| = |T| = |V| - 1$ so (V, T') is a tree.

Claim: $w(T') = w(T)$.

Since T is an minimum spanning tree and T' is a spanning tree,

$$w(T) \leq w(T') = w(T) - w(\{u_1, u_2\}) + w(e) \leq w(T)$$

because $w(e) \leq w(\{u_1, u_2\})$, e is a light edge crossing the partition V_1, V_2 and $\{u_1, u_2\}$ crosses the partition. $\{u_1, u_2\} \notin A$. Hence T' is an MST that contains $A \cup \{e\}$. ■

24.3 Kruskal's Algorithms

Invariant: (V,A) is a spanning forest of $G=(V,E)$.

Initially, each vertex is in its own component.

```
KRUSKAL($V,A$)
$A = \emptyset$
$Q$ = PRIORITY-QUEUE(all edges $e$ where priority is $w(e)$)
for $v \in V$
    MAKE-SET($v$)
while $|A| < n-1$
    $\{u,v\}$ = EXTRACT-MIN($Q$)
    $u'$ = FIND-SET($u$)
    $v'$ = FIND-SET($v$)
    // if $u$ and $v$ are in different components of $(V,A)$
    if $u' \neq v'$
        add $\{u,v\}$ to $A$
        LINK($u',v'$)
```

Building the priority queue with m elements. This takes $O(m)$ time.

At most m EXTRACT-MIN operations. This takes $O(m \lg m)$ time.

Alternatively, these two steps can be combined into one step that sorts the edges by weights. If the range of size if $O(m)$, we can use a non-comparison based sorting algorithm like counting sort to achieve an $O(m)$ time bound.

To find connected components, we use disjoint sets. In particular, we have n MAKE-SET operations, at most $2m$ FIND-SET operations, and $n-1$ LINK operations. We can use union-by-rank with path compression to implement the disjoint set. The sequence complexity is $O(n+m\log^* n) = O(m\log^* n)$ where $n \leq m$. Alterenatively, we can use linked lists with union-by-size, which takes $O(m+n\log n)$. But no matter what implementation we use for the disjoint set, we still have the $O(m\log m)$ overhead from the heap/sorting.

24.4 Prim's Algorithm

The idea is to grow the MST from a single vertex r. V' is the set of nodes reachable from r via edges in A. We maintain the invariant that (V',A) is a tree.

Repeatedly add a light edge crossing the partition $(V', V-V')$ to A.

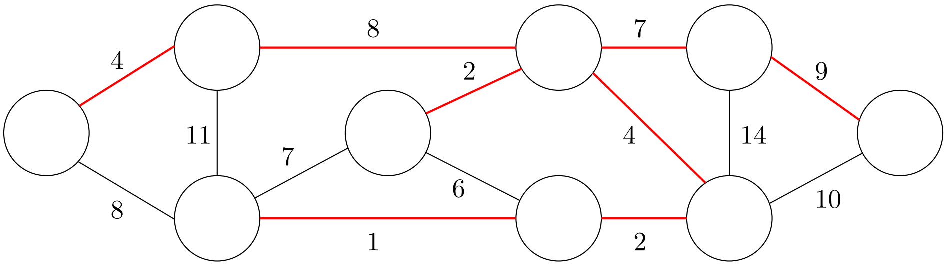


Figure 24.1: The minimum spaning tree resulted from running the Kruskal's algorithm. A step-by-step diagram can be found at pp 632 of CLRS.

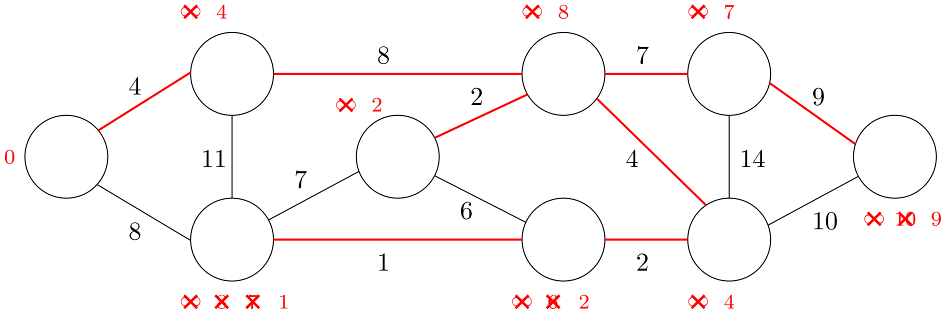


Figure 24.2: The minimum spaning tree resulted from running the Prim's algorithm. The priorities are labeled in red next to each node.

```
PRIM($V, r$)
for $v \in V - \{r\}$
    $v.priority = \infty$
    $v.parent = \text{NIL}$
$Q = \text{PRIORITY-QUEUE}(V - \{r\})$
$u = r$
while $Q \neq \emptyset$
    for neighbor $v$ of $u$
        if $v \in Q$ and $w(\{u,v\}) < v.priority$
            $v.priority = w(\{u,v\})$
            $\text{DECREAE-PRIORITY}(Q, v, priority = w(\{u,v\}))$
            $v.parent = u$
    $u = \text{EXTRACT-MIN}(Q)$
    add $\{u, u.parent\}$ to $A$
return $(V, A)$
```

Building the priority queue takes $\Theta(n)$ time. There are another $(n-1)$ EXTRACT-MIN operations, and m decrease priority operations. We can augment vertices so we know whether they are in Q or in V' with $O(1)$ per lookup.

If we implement the priority queue using a heap, both EXTRACT-MIN and decreasing priority are $O(\log n)$. If we use a Fibonacci heap, it takes $O(1)$ amortized for each DECREASE-PRIORITY operation, and EXTRACT-MIN still takes $O(\log n)$. Overall, the algorithm requires $O(m+n\log n)$ using a Fibonacci heap.

Both Kruskal and Prim's algorithms are greedy algorithms. At each step/iteration, it takes the locally optimal step. At each step of the algorithm, one of several possible choices must be made. In the context of finding the MST, the choices involve what edge to add to A. A greedy strategy chooses the edge which is best at the moment (locally optimal).

In Kruskal's algorithm, the choice is the edge of smallest weight that connects 2 different components. In Prim's algorithm, the choice is the edge of smallest weight that connects a vertex in $V-V'$ to a vertex in $V-V'$. The partition V' starts as a single node and grows to span all nodes in the graph.

By Theorem 24.2.1, both Kruskal's and Prim's algorithms guarantee that the result is a globally optimal minimum spanning tree. In general, greedy algorithms do not guarantee a globally optimal solution, but they often provide a good approximation to many hard-to-solve problems.

Chapter 25 Single-Source Shortest Path Algorithms

25.1 Shortest Path

25.2 Dijkstra's Algorithm

```
DIJKSTRA($G, s$)
for $v \in V - \{s\}$
    $d[v] = \infty$
    $v.parent = \infty$
$d[s] = 0$
$s.parent = s$
$Q =$ PRIORITY-QUEUE($V$, $key = d$)
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN($Q$)
    for neighbor $v$ of $u$
        if $v \in Q$ and $d[v] > d[u] + w(\{u, v\})$
            $d[v] = d[u] + w(\{u, v\})$
            DECREASE-PRIORITY($Q, v, priority = d[u] + w(\{u, v\})$)
            $v.parent = u$
```

Appendix

Commonly Used Axioms & Theorems

Rules of Inference

Axiom 1 — Modus Ponens. $(P \wedge (P \text{ IMPLIES } Q)) \text{ IMPLIES } Q$

Axiom 2 — Modus Tollens. $(\neg Q \wedge (P \text{ IMPLIES } Q)) \text{ IMPLIES } \neg P$

Axiom 3 — Hypothetical Syllogism (transitivity).

$((P \text{ IMPLIES } Q) \wedge (Q \text{ IMPLIES } R)) \text{ IMPLIES } (P \text{ IMPLIES } R)$

Axiom 4 — Disjunctive Syllogism. $((P \vee Q) \wedge \neg P) \text{ IMPLIES } Q$

Axiom 5 — Addition. $P \text{ IMPLIES } (P \vee Q)$

Axiom 6 — Simplification. $(P \wedge Q) \text{ IMPLIES } P$

Axiom 7 — Conjunction. $((P) \wedge (Q)) \text{ IMPLIES } (P \wedge Q)$

Axiom 8 — Resolution. $((P \lor Q) \land (\neg P \lor R))$ IMPLIES $(Q \lor R)$

Laws of Logic

Axiom 9 — Implication Law. $(P \text{ IMPLIES } Q) \equiv (\neg P \lor Q)$

Axiom 10 — Distributive Law.

$$(P \land (Q \lor R)) \equiv ((P \land Q) \lor (P \land R))$$
$$(P \lor (Q \land R)) \equiv ((P \lor Q) \land (P \lor R))$$

Axiom 11 — De Morgan's Law.

$$\neg(P \land Q) \equiv (\neg P \lor \neg Q)$$
$$\neg(P \lor Q) \equiv (\neg P \land \neg Q)$$

Axiom 12 — Absorption Law.

$$(P \lor (P \land Q)) \equiv P$$
$$(P \land (P \lor Q)) \equiv P$$

Axiom 13 — Commutativity of AND. $A \land B \equiv B \land A$

Axiom 14 — Associativity of AND. $(A \land B) \land C \equiv A \land (B \land C)$

Axiom 15 — Identity of AND. $\mathbf{T} \land A \equiv A$

Axiom 16 — Zero of AND. $\mathbf{F} \land A \equiv \mathbf{F}$

Axiom 17 — Idempotence for AND. $A \land A \equiv A$

Axiom 18 — Contradiction for AND. $A \land \neg A \equiv \mathbf{F}$

Axiom 19 — Double Negation. $\neg(\neg A) \equiv A$

Axiom 20 — Validity for OR. $A \lor \neg A \equiv \mathbf{T}$

Induction

Axiom 21 — Well Ordering Principle. Every nonempty set of nonnegative integers has a smallest element. i.e., For any $A \subset \mathbb{N}$ such that $A \neq \emptyset$, there is some $a \in A$ such that $\forall a' \in A. a \leq a'$.

Recurrences

Theorem 22 — The Master Theorem. Suppose that for $n \in \mathbb{Z}^+$.

$$T(n) = \begin{cases} c & \text{if } n < B \\ a_1 T(\lceil n/b \rceil) + a_2 T(\lfloor n/b \rfloor) + dn^i & \text{if } n \geq B \end{cases}$$

where $a_1, a_2, B, b \in \mathbb{N}$.

Let $a = a_1 + a_2 \geq 1$, $b > 1$, and $c, d, i \in \mathbb{R} \cup \{0\}$. Then,

$$T(n) \in \begin{cases} O(n^i \log n) & \text{if } a = b^i \\ O(n^i) & \text{if } a < b^i \\ O(n^{\log_b a}) & \text{if } a > b^i \end{cases}$$

Linear Recurrences:

A linear recurrence is an equation

$$f(n) = \underbrace{a_1 f(n-1) + a_2 f(n-2) + \cdots + a_d f(n-d)}_{\text{homogeneous part}} + \underbrace{g(n)}_{\text{inhomogeneous part}}$$

along with some boundary conditions.

The procedure for solving linear recurrences are as follows:

1. Find the roots of the characteristic equation. Linear recurrences usually have exponential solutions (such as x^n). Such solution is called the **homogeneous solution**.

$$x^n = a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_{k-1} x + a_k$$

2. Write down the homogeneous solution. Each root generates one term and the homogeneous solution is their sum. A non-repeated root r generates the term cr^n, where c is a constant to be determined later. A root with r with multiplicity k generates the terms

$$d_1 r^n \quad d_2 n r^n \quad d_3 n^2 r^n \quad \cdots \quad d_k n^{k-1} r^n$$

where $d_1, \cdots, d_k$ are constants to be determined later.

3. Find a **particular solution** for the full recurrence including the inhomogeneous part, but without considering the boundary conditions.
   If $g(n)$ is a constant or a polynomial, try a polynomial of the same degree, then of one higher degree, then two higher. If $g(n)$ is exponential in the form $g(n) = k^n$, then try $f(n) = ck^n$, then $f(n) = (bn + c)k^n$, then $f(n) = (an^2 + bn + c)k^n$, etc.
4. Write the **general solution**, which is the sum of homogeneous solution and particular solution.
5. Substitute the boundary condition into the general solution. Each boundary condition gives a linear equation. Solve such system of linear equations for the values of the constants to make the solution consistent with the boundary conditions.

Basic Prerequisite Mathematics

Basic Prerequisite Mathematics

SET THEORY

Common Sets

- $\mathbb{N} = \{0, 1, 2, \ldots\}$: the natural numbers, or non-negative integers. The convention in computer science is to include 0 in the natural numbers.
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$: the integers
- $\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$: the positive integers
- $\mathbb{Z}^- = \{-1, -2, -3, \ldots\}$: the negative integers
- $\mathbb{Q}$ the rational numbers, $\mathbb{Q}^+$ the positive rationals, and $\mathbb{Q}^-$ the negative rationals.
- $\mathbb{R}$ the real numbers, $\mathbb{R}^+$ the positive reals, and $\mathbb{R}^-$ the negative reals.

Notation

For any sets A and B, we will use the following standard notation.

- $x \in A$: "x is an element of A" or "A contains x"
- $A \subseteq B$: "A is a subset of B" or "A is included in B"
- $A = B$: "A equals B" (Note that $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.)
- $A \subsetneq B$: "A is a proper subset of B"
  (Note that $A \subsetneq B$ if and only if $A \subseteq B$ and $A \neq B$.)
- $A \cup B$: "A union B"
- $A \cap B$: "A intersection B"
- $A - B$: "A minus B" (*set* difference)
- $|A|$: "cardinality of A" (the number of elements of A)
- $\varnothing$ or $\{\}$: "the empty set"
- $\mathcal{P}(A)$ or 2^A: "powerset of A" (the set of all subsets of A)
  If $A = \{a, 34, \triangle\}$, then $\mathcal{P}(A) = \big\{\{\}, \{a\}, \{34\}, \{\triangle\}, \{a, 34\}, \{a, \triangle\}, \{34, \triangle\}, \{a, 34, \triangle\}\big\}$.
  $S \in \mathcal{P}(A)$ means the same as $S \subseteq A$.
- $\{x \in A \mid P(x)\}$: "the set of elements x in A for which $P(x)$ is true"
  For example, $\{x \in \mathbb{Z} \mid \cos(\pi x) > 0\}$ represents the set of integers x for which $\cos(\pi x)$ is greater than zero, *i.e.*, it is equal to $\{\ldots, -4, -2, 0, 2, 4, \ldots\} = \{x \in \mathbb{Z} \mid x \text{ is even}\}$.

- $A \times B$: "the cross product or Cartesian product of A and B"
  $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$.
  If $A = \{1, 2, 3\}$ and $B = \{5, 6\}$, then $A \times B = \{(1, 5), (1, 6), (2, 5), (2, 6), (3, 5), (3, 6)\}$.)

- A^n: "the cross product of n copies of A"
  This is set of all sequences of $n \geq 1$ elements, each of which is in A.

- B^A or $A \to B$: "the set of all functions from A to B."

- $f : A \to B$ or $f \in B^A$: "f is a function from A to B"
  f associates one element $f(x) \in B$ to every element $x \in A$.

NUMBER THEORY

For any two natural numbers a and b, we say that a *divides* b if there exists a natural number c such that $b = ac$. In such a case, we say that a is a *divisor* of b (*e.g.*, 3 is a divisor of 12 but 3 is not a divisor of 16). Note that any natural number is a divisor of 0 and 1 is a divisor of any natural number. A number a is *even* if 2 divides a and is *odd* if 2 does not divide a.

A natural number p is *prime* if it has exactly two positive divisors (*e.g.*, 2 is prime since its positive divisors are 1 and 2 but 1 is **not** prime since it only has one positive divisor: 1). There are an infinite number of prime numbers and any integer greater than one can be expressed in a unique way as a finite product of prime numbers (*e.g.*, $8 = 2^3$, $77 = 7 \times 11$, $3 = 3$).

Inequalities

For any integers m and n, $m < n$ if and only if $m + 1 \leq n$ and $m > n$ if and only if $m \geq n + 1$. For any real numbers w, x, y, and z, the following properties always hold (they also hold when $<$ and $\leq$ are exchanged throughout with $>$ and $\geq$, respectively).

- if $x < y$ and $w \leq z$, then $x + w < y + z$

- if $x < y$, then $\begin{cases} xz < yz & \text{if } z > 0 \\ xz = yz & \text{if } z = 0 \\ xz > yz & \text{if } z < 0 \end{cases}$

- if $x \leq y$ and $y < z$ (or if $x < y$ and $y \leq z$), then $x < z$

Functions

Here are some common number-theoretic functions together with their definitions and properties of them. (Unless noted otherwise, in this section, x and y represent arbitrary real numbers and k, m, and n represent arbitrary positive integers.)

- $\min\{x, y\}$: "minimum of x and y" (the smallest of x or y)

  Properties: $\min\{x, y\} \leq x$
  $\min\{x, y\} \leq y$

- $\max\{x, y\}$: "maximum of x and y" (the largest of x or y)

  Properties: $x \leq \max\{x, y\}$
  $y \leq \max\{x, y\}$

- $\lfloor x \rfloor$: "floor of x" (the greatest integer less than or equal to x, *e.g.*, $\lfloor 5.67 \rfloor = 5$, $\lfloor -2.01 \rfloor = -3$)

  Properties: $x - 1 < \lfloor x \rfloor \leq x$
  $\lfloor -x \rfloor = -\lceil x \rceil$
  $\lfloor x + k \rfloor = \lfloor x \rfloor + k$
  $\lfloor \lfloor k/m \rfloor /n \rfloor = \lfloor k/mn \rfloor$
  $(k - m + 1)/m \leq \lfloor k/m \rfloor$

- $\lceil x \rceil$: "ceiling of x" (the least integer greater than or equal to x, *e.g.*, $\lceil 5.67 \rceil = 6$, $\lceil -2.01 \rceil = -2$)

  Properties: $x \leq \lceil x \rceil < x + 1$
  $\lceil -x \rceil = -\lfloor x \rfloor$
  $\lceil x + k \rceil = \lceil x \rceil + k$
  $\lceil \lceil k/m \rceil /n \rceil = \lceil k/mn \rceil$
  $\lceil k/m \rceil \leq (k + m - 1)/m$

  Additional property of $\lfloor\ \rfloor$ and $\lceil\ \rceil$: $\lfloor k/2 \rfloor + \lceil k/2 \rceil = k$.

- $|x|$: "absolute value of x" ($|x| = x$ if $x \geq 0$; $-x$ if $x < 0$, *e.g.*, $|5.67| = 5.67$, $|-2.01| = 2.01$)

  BEWARE! The same notation is used to represent the cardinality $|A|$ of a set A and the absolute value $|x|$ of a number x so be sure you are aware of the context in which it is used.

- $m \operatorname{div} n$: "the quotient of m divided by n" (integer division of m by n, *e.g.*, $5 \operatorname{div} 6 = 0$, $27 \operatorname{div} 4 = 6$, $-27 \operatorname{div} 4 = -6$)

  Properties: If $m, n > 0$, then $m \operatorname{div} n = \lfloor m/n \rfloor$
  $(-m) \operatorname{div} n = -(m \operatorname{div} n) = m \operatorname{div} (-n)$

- $m \operatorname{rem} n$: "the remainder of m divided by n" (*e.g.*, $5 \operatorname{rem} 6 = 5$, $27 \operatorname{rem} 4 = 3$, $-27 \operatorname{rem} 4 = -3$)

  Properties: $m = (m \operatorname{div} n) \cdot n + m \operatorname{rem} n$
  $(-m) \operatorname{rem} n = -(m \operatorname{rem} n) = m \operatorname{rem} (-n)$

- $m \bmod n$: "m modulo n" (*e.g.*, $5 \bmod 6 = 5$, $27 \bmod 4 = 3$ $-27 \bmod 4 = 1$)

  Properties: $0 \leq m \bmod n < n$
  n divides $m - (m \bmod n)$.

- $\gcd(m, n)$: "greatest common divisor of m and n" (the largest positive integer that divides both m and n)

  For example, $\gcd(3, 4) = 1$, $\gcd(12, 20) = 4$, $\gcd(3, 6) = 3$

- $\operatorname{lcm}(m, n)$: "least common multiple of m and n" (the smallest positive integer that m and n both divide)

  For example, $\operatorname{lcm}(3, 4) = 12$, $\operatorname{lcm}(12, 20) = 60$, $\operatorname{lcm}(3, 6) = 6$

  Properties: $\gcd(m, n) \cdot \operatorname{lcm}(m, n) = m \cdot n$.

CALCULUS

Limits and Sums

An infinite sequence of real numbers $\{a_n\} = a_1, a_2, \ldots, a_n, \ldots$ *converges* to a limit $L \in \mathbb{R}$ if, for every $\varepsilon > 0$, there exists $n_0 \geq 0$ such that $|a_n - L| < \varepsilon$ for every $n \geq n_0$. In this case, we write $\lim\limits_{n\to\infty} a_n = L$ Otherwise, we say that the sequence *diverges*.

If $\{a_n\}$ and $\{b_n\}$ are two sequences of real numbers such that $\lim\limits_{n\to\infty} a_n = L_1$ and $\lim\limits_{n\to\infty} b_n = L_2$, then

$$\lim_{n\to\infty}(a_n + b_n) = L_1 + L_2 \quad \text{and} \quad \lim_{n\to\infty}(a_n \cdot b_n) = L_1 \cdot L_2.$$

In particular, if c is any real number, then

$$\lim_{n\to\infty}(c \cdot a_n) = c \cdot L_1.$$

The sum $a_1 + a_2 + \cdots + a_n$ and product $a_1 \cdot a_2 \cdots a_n$ of the finite sequence $a_1, a_2, \ldots, a_n$ are denoted by

$$\sum_{i=1}^{n} a_i \text{ and } \prod_{i=1}^{n} a_i.$$

If the elements of the sequence are all different and $S = \{a_1, a_2, \ldots, a_n\}$ is the set of elements in the sequence, these can also be denoted by

$$\sum_{a\in S} a \text{ and } \prod_{a\in S} a.$$

Examples:

- For any $a \in \mathbb{R}$ such that $-1 < a < 1$, $\lim\limits_{n\to\infty} a^n = 0$.
- For any $a \in \mathbb{R}^+$, $\lim\limits_{n\to\infty} a^{1/n} = 1$.
- For any $a \in \mathbb{R}^+$, $\lim\limits_{n\to\infty} (1/n)^a = 0$.
- $\lim\limits_{n\to\infty} (1 + 1/n)^n = e = 2.71828182845904523536\ldots$

- For any $a, b \in \mathbb{R}$, the *arithmetic* sum is given by:

$$\sum_{i=0}^{n}(a + ib) = (a) + (a + b) + (a + 2b) + \cdots + (a + nb) = \frac{1}{2}(n + 1)(2a + nb).$$

- For any $a, b \in \mathbb{R}^+$, the *geometric* sum is given by:

$$\sum_{i=0}^{n}(ab^i) = a + ab + ab^2 + \cdots + ab^n = \frac{a(1 - b^{n+1})}{1 - b}.$$

EXPONENTS AND LOGARITHMS

Definition: For any $a, b, c \in \mathbb{R}^+$, $a = \log_b c$ if and only if $b^a = c$.

Notation: For any $x \in \mathbb{R}^+$, $\ln x = \log_e x$ and $\lg x = \log_2 x$.

For any $a, b, c \in \mathbb{R}^+$ and any $n \in \mathbb{Z}^+$, the following properties always hold.

- $\sqrt[n]{b} = b^{1/n}$
- $b^a b^c = b^{a+c}$
- $(b^a)^c = b^{ac}$
- $b^a / b^c = b^{a-c}$
- $b^0 = 1$
- $a^b c^b = (ac)^b$
- $b^{\log_b a} = a = \log_b b^a$
- $a^{\log_b c} = c^{\log_b a}$
- $\log_b(ac) = \log_b a + \log_b c$
- $\log_b(a^c) = c \cdot \log_b a$
- $\log_b(a/c) = \log_b a - \log_b c$
- $\log_b 1 = 0$
- $\log_b a = \log_c a / \log_c b$

BINARY NOTATION

A *binary number* is a sequence of bits $a_k \cdots a_1 a_0$ where each bit a_i is equal to 0 or 1. Every binary number represents a natural number in the following way:

$$(a_k \cdots a_1 a_0)_2 = \sum_{i=0}^{k} a_i 2^i = a_k 2^k + \cdots + a_1 2 + a_0.$$

For example, $(1001)_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 1 = 9$, $(01110)_2 = 8 + 4 + 2 = 14$.

Properties:

- If $a = (a_k \cdots a_1 a_0)_2$, then $2a = (a_k \cdots a_1 a_0 0)_2$, *e.g.*, $9 = (1001)_2$ so $18 = (10010)_2$.
- If $a = (a_k \cdots a_1 a_0)_2$, then $\lfloor a/2 \rfloor = (a_k \cdots a_1)_2$, *e.g.*, $9 = (1001)_2$ so $4 = (100)_2$.
- The smallest number of bits required to represent the positive integer n in binary is called the *length* of n and is equal to $\lceil \log_2(n+1) \rceil$.

Make sure you know how to add and multiply two binary numbers. For example, $(1111)_2 + (101)_2 = (10100)_2$ and $(1111)_2 \times 101)_2 = (1001011)_2$.

Proof Templates

Proof Outlines

LINE NUMBERS: Only lines that are referred to have labels (for example, L1) in this document. For a formal proof, all lines are numbered. Line numbers appear at the beginning of a line. You can indent line numbers together with the lines they are numbering or all line numbers can be unindented, provided you are consistent.

INDENTATION: Indent when you make an assumption or define a variable. Unindent when this assumption or variable is no longer being used.

1. **Implication**: Direct proof of A IMPLIES B.

    L1. Assume A.
    ⋮
    L2. B
  A IMPLIES B; direct proof: L1, L2

2. **Implication**: Indirect proof of A IMPLIES B.

    L1. Assume $NOT(B)$.
    ⋮
    L2. $NOT(A)$
  A IMPLIES B; indirect proof: L1, L2

3. **Equivalence**: Proof of A IFF B.

    L1. Assume A.
    ⋮
    L2. B
  L3. A IMPLIES B; direct proof: L1, L2
    L4. Assume B.
    ⋮
    L5. A
  L6. B IMPLIES A; direct proof: L4, L5
  A IFF B; equivalence: L3, L6

4. **Proof by contradiction** of A.

    L1. To obtain a contradiction, assume $NOT(A)$.
    ⋮
        L2. B
    ⋮
        L3. $NOT(B)$
    L4. This is a contradiction: L2, L3
  Therefore A; proof by contradiction: L1, L4

5. **Modus Ponens**.

   $\vdots$
   L1. A
   $\vdots$
   L2. A IMPLIES B
   B; modus ponens: L1, L2

6. **Conjunction:** Proof of A AND B:

   $\vdots$
   L1. A
   $\vdots$
   L2. B
   A AND B; proof of conjunction; L1, 2

7. **Use of Conjunction**:

   $\vdots$
   L1. A AND B
   A; use of conjunction: L1
   B; use of conjunction: L1

8. **Implication with Conjunction**: Proof of (A_1 AND A_2) IMPLIES B.

   > L1. Assume A_1 AND A_2.
   > A_1; use of conjunction, L1
   > A_2; use of conjunction, L1
   > $\vdots$
   > L2. B

   (A_1 AND A_2) IMPLIES B; direct proof, L1, L2

9. **Implication with Conjunction**: Proof of A IMPLIES (B_1 AND B_2).

   > L1. Assume A.
   > $\vdots$
   > L2. B_1
   > $\vdots$
   > L3. B_2
   > L4. B_1 AND B_2; proof of conjunction: L2, L3

   A IMPLIES (B_1 AND B_2); direct proof: L1, L4

10. **Disjunction**: Proof of A OR B and B OR A.

    $\vdots$
    L1. A
    A OR B; proof of disjunction: L1
    B OR A; proof of disjunction: L1

11. **Proof by cases**.

    L1. C OR $NOT(C)$ tautology
    L2. Case 1: Assume C.
    $\vdots$
    L3. A
    L4. C IMPLIES A; direct proof: L2, L3
    L5. Case 2: Assume $NOT(C)$.
    $\vdots$
    L6. A
    L7. $NOT(C)$ IMPLIES A; direct proof: L5, L6
    A proof by cases: L1, L4, L7

12. **Proof by cases** of A OR B.

    L1. C OR $NOT(C)$ tautology
    L2. Case 1: Assume C.
    $\vdots$
    L3. A
    L4. A OR B; proof of disjunction, L3
    L5. C IMPLIES (A OR B); direct proof, L2, L4
    L6. Case 2: Assume $NOT(C)$.
    $\vdots$
    L7. B
    L8. A OR B; proof of disjunction, L7
    L9. $NOT(C)$ IMPLIES (A OR B); direct proof: L6, L8
    A OR B; proof by cases: L1, L5, L9

13. **Implication with Disjunction**: Proof by cases of (A_1 OR A_2) IMPLIES B.

    L1. Case 1: Assume A_1.
    $\vdots$
    L2. B
    L3. A_1 IMPLIES B; direct proof: L1,L2
    L4. Case 2: Assume A_2.
    $\vdots$
    L5. B
    L6. A_2 IMPLIES B; direct proof: L4, L5
    (A_1 OR A_2) IMPLIES B; proof by cases: L3, L6

14. **Implication with Disjunction**: Proof by cases of A IMPLIES (B_1 OR B_2).

    - L1. Assume A.
    - L2. C OR NOT(C) tautology
    - L3. Case 1: Assume C.
        - $\vdots$
        - L4. B_1
        - L5. B_1 OR B_2; disjunction: L4
    - L6. C IMPLIES (B_1 OR B_2); direct proof: L3, L5
    - L7. Case 2: AssumeNOT(C).
        - $\vdots$
        - L8. B_2
        - L9. B_1 OR B_2; disjunction: L8
    - L10. NOT(C) IMPLIES (B_1 OR B_2); direct proof: L7, L9
    - L11. B_1 OR B_2; proof by cases: L2, L6, L10

    A IMPLIES (B_1 OR B_2): direct proof. L1, L11

15. **Substitution of a Variable in a Tautology**:
    Suppose P is a propositional variable, Q is a formula, and R' is obtained from R by replacing *every* occurrence of P by (Q).

    L1. R tautology
    R'; substitution of all P by Q: L1

16. **Substitution of a Formula by a Logically Equivalent Formula**:
    Suppose S is a subformula of R and R' is obtained from R by replacing *some* occurrence of S by S'.

    L1. R
    L2. S IFF S'
    L3. R'; substitution of an occurrence of S by S': L1, L2

17. **Specialization**:

    L1. $c \in D$
    L2. $\forall x \in D.P(x)$
    $P(c)$; specialization: L1, L2

18. **Generalization**: Proof of $\forall x \in D.P(x)$.

    - L1. Let x be an arbitrary element of D.
    - $\vdots$
    - L2. $P(x)$

    Since x is an arbitrary element of D,
    $\forall x \in D.P(x)$; generalization: L1, L2

19. **Universal Quantification with Implication**: Proof of $\forall x \in D.(P(x) \text{ IMPLIES } Q(x))$.

> L1. Let x be an arbitrary element of D.
>
> > L2. Assume $P(x)$
> >
> > $\vdots$
> >
> > L3. $Q(x)$
>
> L4. $P(x)$ IMPLIES $Q(x)$; direct proof: L2, L3

Since x is an arbitrary element of D,
$\forall x \in D.(P(x) \text{ IMPLIES } Q(x))$; generalization: L1, L4

20. **Implication with Universal Quantification**: Proof of $(\forall x \in D.P(x))$ IMPLIES A.

> L1. Assume $\forall x \in D.P(x)$.
>
> $\vdots$
>
> L2. $a \in D$
> $P(a)$; specialization: L1, L2
>
> $\vdots$
>
> L3. A

Therefore $(\forall x \in D.P(x))$ IMPLIES A; direct proof: L1, L3

21. **Implication with Universal Quantification**: Proof of A IMPLIES $(\forall x \in D.P(x))$.

> L1. Assume A.
>
> > L2. Let x be an arbitrary element of D.
> >
> > $\vdots$
> >
> > L3. $P(x)$
>
> Since x is an arbitrary element of D,
> L4. $\forall x \in D.P(x)$; generalization, L2, L3

A IMPLIES $(\forall x \in D.P(x))$; direct proof: L1, L4

22. **Instantiation**:

L1. $\exists x \in D.P(x)$

> Let $c \in D$ be such that $P(c)$; instantiation: L1
>
> $\vdots$

23. **Construction**: Proof of $\exists x \in D.P(x)$.

> L1. Let $a = \cdots$
>
> $\vdots$
>
> L2. $a \in D$
>
> $\vdots$
>
> L3. $P(a)$

$\exists x \in D.P(x)$; construction: L1, L2, L3

24. **Existential Quantification with Implication**: Proof of $\exists x \in D.(P(x) \text{ IMPLIES } Q(x))$.

L1. Let $a = \cdots$
$\vdots$
L2. $a \in D$
L3. Suppose $P(a)$.
$\vdots$
L4. $Q(a)$
L5. $P(a)$ IMPLIES $Q(a)$; direct proof: L3, L4
$\exists x \in D.(P(x) \text{ IMPLIES } Q(x))$; construction: L1, L2, L5

25. **Implication with Existential Quantification**: Proof of $(\exists x \in D.P(x))$ IMPLIES A.

L1. Assume $\exists x \in D.P(x)$.
Let $a \in D$ be such that $P(a)$; instantiation: L1
$\vdots$
L2. A
$(\exists x \in D.P(x))$ IMPLIES A; direct proof: L1, L2

26. **Implication with Existential Quantification**: Proof of A IMPLIES $(\exists x \in D.P(x))$.

L1. Assume A.
L2. Let $a = \cdots$
$\vdots$
L3. $a \in D$
$\vdots$
L4. $P(a)$
L5. $\exists x \in D.P(x)$; construction: L2, L3, L4
A IMPLIES $(\exists x \in D.P(x))$; direct proof: L1, L5

27. **Subset**: Proof of $A \subseteq B$.

L1. Let $x \in A$ be arbitrary.
$\vdots$
L2. $x \in B$
The following line is optional:
L3. $x \in A$ IMPLIES $x \in B$; direct proof: L1, L2
$A \subseteq B$; definition of subset: L3 (or L1, L2, if the optional line is missing)

28. **Weak Induction**: Proof of $\forall n \in N.P(n)$

    Base Case:

    $\vdots$

    L1. $P(0)$
        L2. Let $n \in N$ be arbitrary.
            L3. Assume $P(n)$.
            $\vdots$
            L4. $P(n+1)$
        The following two lines are optional:
        L5. $P(n)$ IMPLIES $(P(n+1)$; direct proof of implication: L3, L4
    L6. $\forall n \in N.(P(n)$ IMPLIES $P(n+1))$; generalization L2, L5
    $\forall n \in N.P(n)$ induction; L1, L6 (or L1, L2, L3, L4, if the optional lines are missing)

29. **Strong Induction**: Proof of $\forall n \in N.P(n)$

        L1. Let $n \in N$ be arbitrary.
            L2. Assume $\forall j \in N.(j < n$ IMPLIES $P(j))$
            $\vdots$
            L3. $P(n)$
        The following two lines are optional:
        L4. $\forall j \in N.(j < n$ IMPLIES $P(j))$ IMPLIES $P(n)$; direct proof of implication: L2, L3
    L5. $\forall n \in N.[\forall j \in N.(j < n$ IMPLIES $P(j))$ IMPLIES $P(n)]$; generalization: L1, L4
    $\forall n \in N.P(n)$; strong induction: L5 (or L1, L2, L3, if the optional lines are missing)

30. **Structural Induction**: Proof of $\forall e \in S.P(e)$, where S is a recursively defined set

    Base case(s):
        L1. For each base case e in the definition of S
        L2. $P(e)$.
    Constructor case(s):
        L3. For each constructor case e of the definition of S,
            L4. assume $P(e')$ for all components e' of e.
            $\vdots$
            L5. $P(e)$
    $\forall e \in S.P(e)$; structural induction: L1, L2, L3, L4, L5

31. **Well Ordering Principle**: Proof of $\forall e \in S.P(e)$, where S is a well ordered set, i.e. every nonempty subset of S has a smallest element.

    L1. To obtain a contradiction, suppose that $\forall e \in S.P(e)$ is false.
    L2. Let $C = \{e \in S \mid P(e) \text{ is false}\}$ be the set of counterexamples to P.
    L3. $C \neq \phi$; definition: L1, L2
        L4. Let e be the smallest element of C; well ordering principle: L2, L3
            Let $e' = \cdots$
            $\vdots$
            L5. $e' \in C$
            $\vdots$
            L6. $e' < e$.
    L7. This is a contradiction: L4, L5, L6
    $\forall e \in S.P(e)$; proof by contradiction: L1, L7

Index

Index

2-3 tree, 28

abstract data type, 13
accounting method, 56
adjacency list, 122
adjacency matrix, 122
ADT, 13
aggregate method, 56
allocated charge, 56
amortized cost, 53, 56
average case time complexity, 42

back edge, 126, 130
bit vector direct access table, 17
breadth-first search (BFS), 125

comparison model, 109
complement (probability), 41
complete binary tree, 34
conditional probability, 42
credit invariant, 56
cross edge, 126, 130

data structure, 13
degree, 121
depth-first search (DFS), 129
direct access table, 16
directed graph, 121
dynamic array, 61

event, 41

FKS hashing, 76
forward edge, 130
full binary tree, 34

hashing, 71
heap, 34
heap shape, 34
height balanced, 21
hypergraph, 121

in-degree, 121
incidence matrix, 123

linear probing, 81
lower bound, 109

max-heap, 35
max-heap property, 35
model of computation, 109

neighbors, 121

open addressing, 81

out-degree, 121
outcome, 41

Parenthesis Theorem, 130
perfect binary tree, 34
perfect hashing, 74
potential function, 58
predecessor, 121
priority queue, 33
probability space, 41

quicksort, 44

randomized quicksort, 46
red-black tree, 21
Rivest-Vuiellemin Theorem, 123

sample space, 41
simple graph, 121
spanning forest, 133
spanning tree, 133
subgraphs, 121
successor, 121

topological sort, 131
tree edge, 126, 130

undirected graph, 121
universal hashing, 72

weight balanced, 21
weighted graph, 121

Bibliography

Courses

[2] Erik Demaine and Srini Devadas. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. Fall 2011.

[3] Erik Demaine, Srini Devadas, and Nancy Lynch. *6.046J Design and Analysis of Algorithms*. Massachusetts Institute of Technology. Spring 2015.

[4] Faith Ellen. *CSC240S1 Winter 2021*. University of Toronto. 2021.

[5] Faith Ellen. *CSC265F1 Fall 2021*. University of Toronto. 2021.

[10] Mauricio Karchmer, Anand Natarajan, and Nir Shavit. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. Spring 2021.

Books

[1] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[6] Jeff Erickson. *Algorithms*. 1st. 2019. ISBN: 978-1792644832.

[9] Vassos Hadzilacos. *Course notes for CSC B36/236/240 Introduction to Theory of Computation*. University of Toronto. 2007.

[11] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. 8th edition. New York: McGraw-Hill Education, 2019.

Journal Articles

[7] Michael Fredman, Janos Komlos, and Endre Szemeredi. “Storing a Sparse Table with O(1) Worst Case Access Time”. In: *Journal of the Association for Computing Machinery* 31.3 (July 1984), pages 538–544.

[8] Igal Galperin and Ronald L. Rivest. “Scapegoat Trees”. In: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (Jan. 1993), pages 165–174.