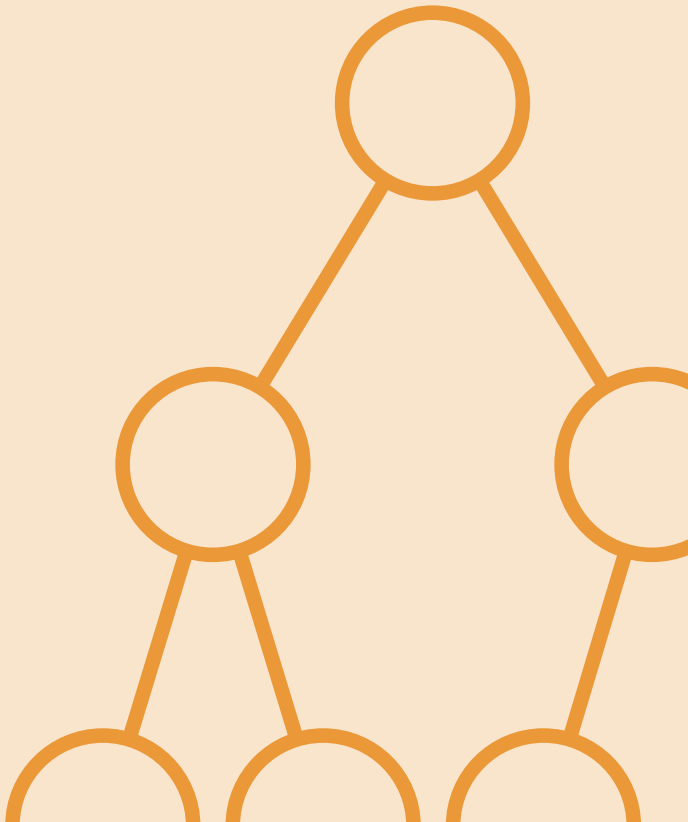
Data Structures and Analysis



Contents

I Data Structures

1 Abstract Data Types 9

2 Priority Queue and Heap 11

2.1 Priority Queue **11**

2.1.1 Priority Queue ADT 11

2.1.2 Primitive Implementation Using Linked Lists 11

2.2 Heap **11**

2.2.1 Types of Binary Trees 12

2.2.2 Heap Property 13

2.3 Maintaining the Heap Property **13**

2.3.1 Correctness of MAX-HEAPIFY 14

2.3.2 Running Time of MAX-HEAPIFY 14

2.4 Inserting Into Max-Heap **14**

2.5 Build Heap From Unsorted Array **14**

2.5.1 Running Time of BUILD-MAX-HEAP 14

2.6 Heapsort **15**

3 Mergeable Heaps 17

4 Balanced Binary Search Trees 19
5 Augmenting Data Structures 21
6 Hashing 23

II Algorithm Analysis

7 Randomized Algorithms 27
8 Amortized Analysis 29
8.1 Amortized Cost 29
8.2 Aggregate Method 30
8.3 Accounting Method 30
8.4 Potential Method 30

III Advanced Data Structures

9 Dynamic Tables 33
10 Disjoint Sets 35

IV Lower Bounds

11 Decision Trees 39
11.1 Lower Bounds 39
11.2 Comparison Model 39
11.3 Decision Tree 40
11.3.1 Intuition 40
11.3.2 Decision Tree for Searching 40
11.3.3 Decision Tree for Sorting 40
11.4 Sorting in Linear Time 40
12 Information Theory 41
13 Adversary Arguments 43
14 Reduction 45

V Graphs

15 Breadth-First Search 49
15.1 Definition 49
15.2 Representations of Graphs 49
15.2.1 Adjacency List 49
15.3 Breadth-first Search 49
16 Depth-First Search 51
17 Minimum Spanning Trees 53
18 Bellman-Ford's Algorithm 55
19 Dijkstra's Algorithm 57

Appendix

Axioms & Theorems 61
Basic Prerequisite Mathematics 65
Proof Templates 71
Index 83
Bibliography 85
Courses 85
Books 85

Tutorial

Data Structures

1 Abstract Data Types 9

2 Priority Queue and Heap 11
2.1 Priority Queue
2.2 Heap
2.3 Maintaining the Heap Property
2.4 Inserting Into Max-Heap
2.5 Build Heap From Unsorted Array
2.6 Heapsort

3 Mergeable Heaps 17

4 Balanced Binary Search Trees 19

5 Augmenting Data Structures 21

6 Hashing 23

Lecture 1 Abstract Data Types

Lecture 2 Priority Queue and Heap

2.1 Priority Queue

2.1.1 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.

2.1.2 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 $\Theta(\log n)$.

2.2 Heap

2.2.1 Types of Binary Trees

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

Definition 2.2.1 — 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 2.2.2 — Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

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

Definition 2.2.3 — Perfect 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.

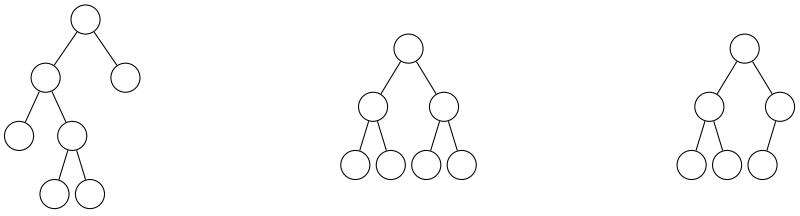


Figure 2.1: From left to right: full binary tree, complete binary tree, perfect binary tree.

Conveniently, a complete binary tree can be represented as an array.

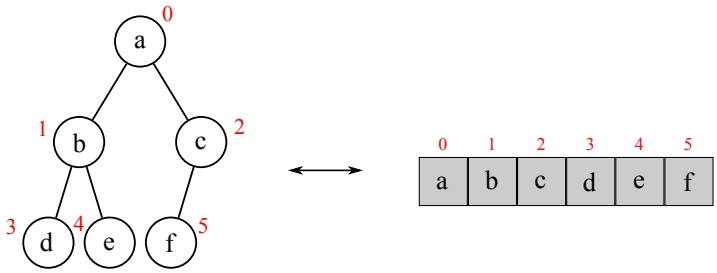


Figure 2.2: A complete binary tree and its corresponding array representation

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

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

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

PARENT(i) = $\lfloor i/2 \rfloor$
LEFT(i) = $2i$
RIGHT(i) = $2i+1$

2.2.2 Heap Property

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

Definition 2.2.4 — 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,

$$A[\text{PARENT}(i)] \geq A[i].$$

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.

2.3 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 $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ 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)

```
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap − size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap − size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

2.3.1 Correctness of MAX-HEAPIFY

2.3.2 Running Time of MAX-HEAPIFY

2.4 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.

2.5 Build Heap From Unsorted Array

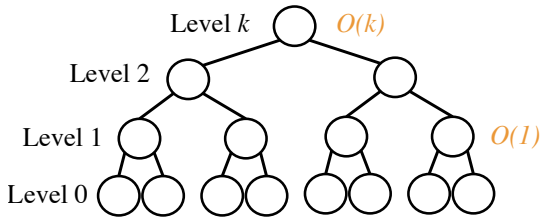
BUILD-MAX-HEAP(A)
1 $A.heapsize = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

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 complete binary tree with n nodes, there are only $\lfloor n/2 \rfloor$ internal nodes.

2.5.1 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,

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

2.6 Heapsort

HEAPSORT(A)

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

Lecture 3 Mergeable Heaps

Lecture 4 Balanced Binary Search Trees

Lecture 5 Augmenting Data Structures

Lecture 6 Hashing

II Algorithm Analysis

7 Randomized Algorithms 27

8 Amortized Analysis 29
8.1 Amortized Cost
8.2 Aggregate Method
8.3 Accounting Method
8.4 Potential Method

Lecture 7 Randomized Algorithms

Lecture 8 Amortized Analysis

8.1 Amortized Cost

Definition 8.1.1 — Amortized Cost. The amortized cost per operation for a sequence of n operations is the total cost of the operations divided by n.

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.

As a starter, 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.

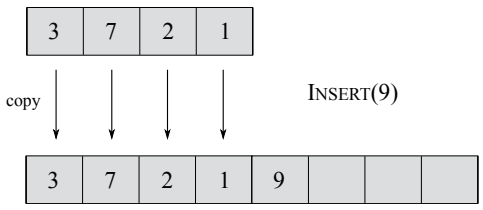


Figure 8.1 The dynamic array after the operation INSERT(9). The length of the new array is doubled, and old elements are copied to the new array.

8.2 Aggregate Method

8.3 Accounting Method

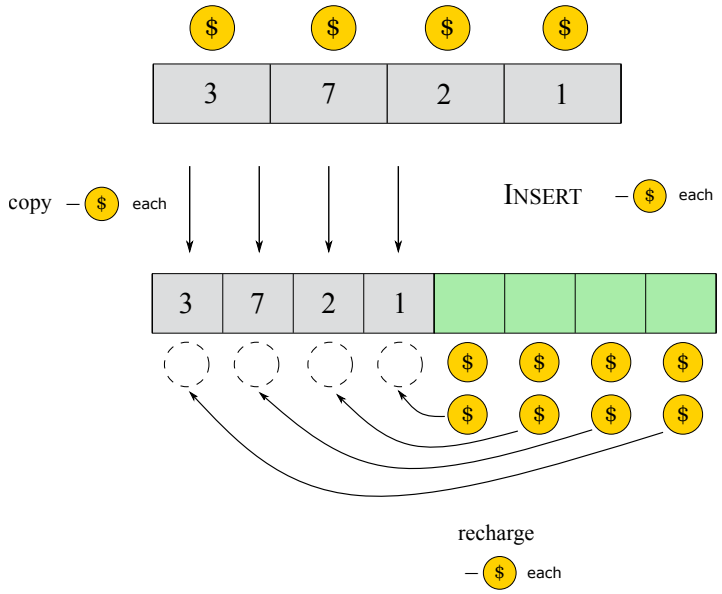


Figure 8.2: <caption>

8.4 Potential Method

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 $\hat{c}_i$ of the i-th operation with respect to the potential function Φ is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

And the total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

III Advanced Data Structures

9 Dynamic Tables 33

10 Disjoint Sets 35

Lecture 9 Dynamic Tables

Lecture 10 Disjoint Sets

IV Lower Bounds

11 Decision Trees 39
11.1 Lower Bounds
11.2 Comparison Model
11.3 Decision Tree
11.4 Sorting in Linear Time

12 Information Theory 41

13 Adversary Arguments 43

14 Reduction 45

Lecture 11 Decision Trees

11.1 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.

Let $T_A(X)$ be the running time of algorithm A given input X. Then, the worst-case running time is

$$T_A(n) = \max_{|X|=n} T_A(X)$$

The worst-case complexity of a **problem** Π is the worst-case running time of the fastest algorithm for solving it.

$$T_\Pi(n) = \min_{A \text{ solves } \Pi} T_A(n) = \min_{A \text{ solves } \Pi} \max_{|X|=n} T_A(X)$$

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, 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.

11.2 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.

11.3 Decision Tree

11.3.1 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.

For any particular n,

- *internal node* corresponds to binary decision in the algorithm (in this case, binary comparisons)
- *leaf* corresponds to a possible answer 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.

11.3.2 Decision Tree for Searching

In this subsection, we will look at the decision tree for binary search and use it to prove that the lower bound of searching under the comparison model is $\Omega(\log n)$.

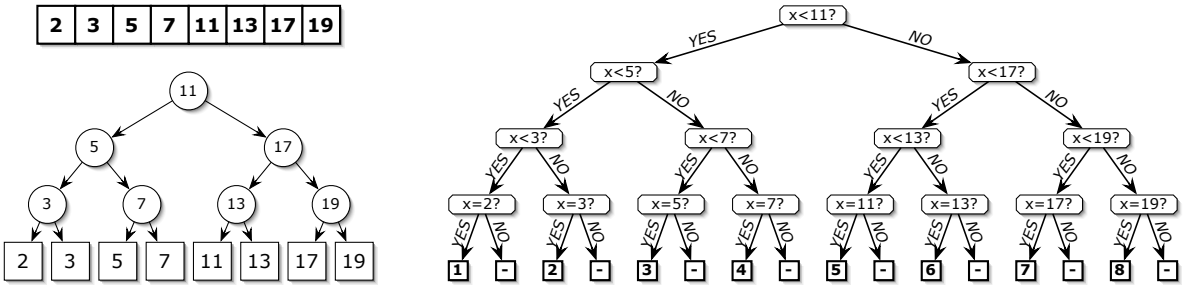


Figure 11.1: <caption>

11.3.3 Decision Tree for Sorting

11.4 Sorting in Linear Time

Lecture 12 Information Theory

Lecture 13 Adversary Arguments

Lecture 14 Reduction

V

Graphs

15 Breadth-First Search 49
15.1 Definition
15.2 Representations of Graphs
15.3 Breadth-first Search

16 Depth-First Search 51

17 Minimum Spanning Trees 53

18 Bellman-Ford's Algorithm 55

19 Dijkstra's Algorithm 57

Lecture 15 Breadth-First Search

15.1 Definition

Definition 15.1.1 — Graphs. A graph is defined as a tuple $G = (V, E)$ where $V$is an arbitrary non-empty finite set, whose elements are called vertices or nodes; and E is a set of pairs of elements of V, which we call edges. For an undirected graph, the edges are unordered pairs u, v. In a directed graph, the edges are ordered pairs (u, v).

Definition 15.1.2 — 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 15.1.3 — 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.

15.2 Representations of Graphs

15.2.1 Adjacency List

Definition 15.2.1 — 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.

15.3 Breadth-first Search

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 $k + 1$.

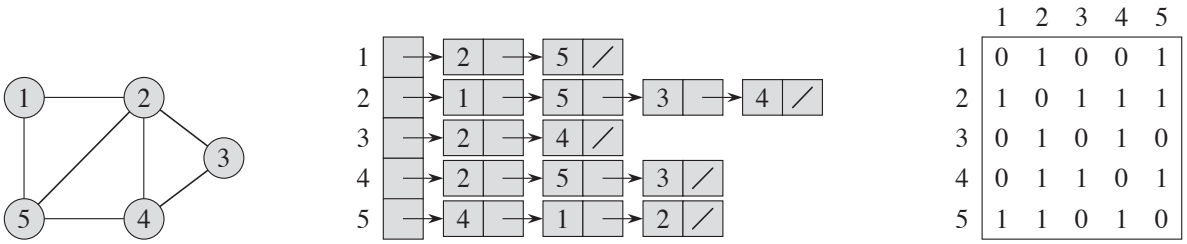


Figure 15.1: Representations of undirected graph: (a) the graph; (b) adjacency list of the graph; (c) adjacency matrix of the graph.

Lecture 16 Depth-First Search

Lecture 17 Minimum Spanning Trees

Lecture 18 Bellman-Ford's Algorithm

Lecture 19 Dijkstra's Algorithm

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. $((P \vee Q) \wedge (\neg P \vee R))$ IMPLIES $(Q \vee R)$

Laws of Logic

Axiom 9 — Implication Law. $(P$ IMPLIES $Q) \equiv (\neg P \vee Q)$

Axiom 10 — Distributive Law.

$$(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$$
$$(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R))$$

Axiom 11 — De Morgan's Law.

$$\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$$
$$\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$$

Axiom 12 — Absorption Law.

$$(P \vee (P \wedge Q)) \equiv P$$
$$(P \wedge (P \vee Q)) \equiv P$$

Axiom 13 — Commutativity of AND. $A \wedge B \equiv B \wedge A$

Axiom 14 — Associativity of AND. $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$

Axiom 15 — Identity of AND. $\mathbf{T} \wedge A \equiv A$

Axiom 16 — Zero of AND. $\mathbf{F} \wedge A \equiv \mathbf{F}$

Axiom 17 — Idempotence for AND. $A \wedge A \equiv A$

Axiom 18 — Contradiction for AND. $A \wedge \neg A \equiv \mathbf{F}$

Axiom 19 — Double Negation. $\neg(\neg A) \equiv A$

Axiom 20 — Validity for OR. $A \vee \neg A \equiv \mathbf{T}$

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 $\forall a' \in A . a \leq a'$.

Basic Prerequisite Mathematics

Basic Prerequisite Mathematics

SET THEORY

Common Sets

- $\mathbb{N} = \{0, 1, 2, \dots\}$: the natural numbers, or non-negative integers. The convention in computer science is to include 0 in the natural numbers.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$: the integers
- $\mathbb{Z}^+ = \{1, 2, 3, \dots\}$: the positive integers
- $\mathbb{Z}^- = \{-1, -2, -3, \dots\}$: 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 $\{\dots, -4, -2, 0, 2, 4, \dots\} = \{x \in \mathbb{Z} \mid x \text{ is even}\}$.

- $A \times B$: "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 $\lim_{n\to\infty} a_n = L$ Otherwise, we say that the sequence *diverges*.

If $\{a_n\}$ and $\{b_n\}$ are two sequences of real numbers such that $\lim_{n\to\infty} a_n = L_1$ and $\lim_{n\to\infty} b_n = L_2$, 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 $-1 < a < 1$, $\lim_{n\to\infty} a^n = 0$.
- For any $a \in \mathbb{R}^+$, $\lim_{n\to\infty} a^{1/n} = 1$.
- For any $a \in \mathbb{R}^+$, $\lim_{n\to\infty} (1/n)^a = 0$.
- $\lim_{n\to\infty}(1 + 1/n)^n = e = 2.71828182845904523536\ldots$

- 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**.

   ⋮
   L1. A
   ⋮
   L2. A IMPLIES B
   B; modus ponens: L1, L2

6. **Conjunction:** Proof of A AND B:

   ⋮
   L1. A
   ⋮
   L2. B
   A AND B; proof of conjunction; L1, 2

7. **Use of Conjunction**:

   ⋮
   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
      ⋮
      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.
      ⋮
      L2. B_1
      ⋮
      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.

    ⋮
    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.
        ⋮
        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).
        ⋮
        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.
    ⋮
    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 $\forall x \in D.(P(x)$ 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,
    $\forall x \in D.(P(x)$ 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 $\exists x \in D.(P(x)$ 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

    $\exists x \in D.(P(x)$ 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

adjacency-list, 49
amortized cost, 29

breadth-first search (BFS), 49

comparison model, 39
complete binary tree, 12

degree, 49
dynamic array, 29

edges, 49

full binary tree, 12

graphs, 49

heap, 11

in-degree, 49

lower bound, 39

max-heap, 13
max-heap property, 13
model of computation, 39

neighbors, 49

out-degree, 49

perfect binary tree, 12
potential function, 30
predecessor, 49
priority queue, 11

subgraphs, 49
successor, 49

vertices, 49

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.

[7] 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] Vassos Hadzilacos. *Course notes for CSC B36/236/240 Introduction to Theory of Computation*. University of Toronto. 2007.

[8] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. 8th edition. New York: McGraw-Hill Education, 2019.

Tutorial