Introduction

CMSC 142: Design and Analysis of Algorithms

Department of Mathematics and Computer Science
College of Science
University of the Philippines Baguio

Reference: Chapter 1: Sections 1.1 to 1.4 of Introduction to Design and Analysis of Algorithms 3rd Edition (*Levitin*, *2012*)

Overview

What is an Algorithm?

An **algorithm** is a sequence of unambiguous instructions for solving a problem, i.e. for obtaining a required output for any legitimate input in a finite amount of time.

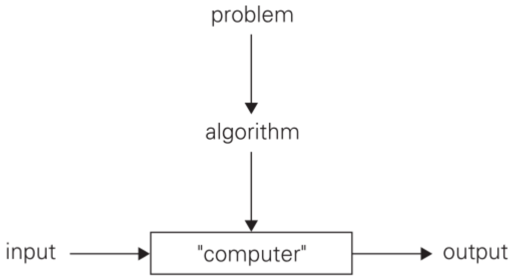


Figure: The notion of Algorithm

What is an Algorithm?

An algorithm satisfy the following requirements:

- finiteness, definiteness
- clearly specified input and expected output
- effectiveness

Remarks

1. The same algorithm can be represented in different ways.
2. Several algorithms for solving the same problem may exist.
3. Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Some Well-known Computation Problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Traveling salesman problem
- Knapsack problem
- Chess
- Tower of Hanoi
- Program termination

Some of these problems do not have efficient algorithms, or algorithms at all.

The GCD Problem

Recall that the greatest common divisor of two nonnegative, not both zero integers m and n, denoted $\gcd(m, n)$, is the largest integer that divides both m and n evenly, i.e. with a remainder of zero.

Euclid's Algorithm

Euclid's algorithm is based on repeated application of the equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

where $m \bmod n$ is the remainder of the division of m by n.

Illustration

Find the following using Euclid's algorithm: (a) $\gcd(60, 24)$, (b) $\gcd(8, 17)$

GCD Algorithm 1

Euclid's algorithm for computing $\gcd(m, n)$

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

GCD Algorithm 2

Consecutive integer checking algorithm for computing $\gcd(m, n)$

Step 1 Assign the value of $\min\{m, n\}$ to t.

Step 2 Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.

Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise proceed
to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2.

GCD Algorithm 3

High School procedure for computing $\gcd(m, n)$

Step 1 Find the prime factors of m.

Step 2 Find the prime factors of n.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Algorithm for Sieve of Eratosthenes

```
ALGORITHM  Sieve(n)
    //Implements the sieve of Eratosthenes
    //Input: A positive integer $n > 1$
    //Output: Array $L$ of all prime numbers less than or equal to $n$
      for $p \leftarrow 2$ to $n$ do $A[p] \leftarrow p$
      for $p \leftarrow 2$ to $\lfloor \sqrt{n} \rfloor$ do
          if $A[p] \neq 0$            //$p$ hasn't been eliminated on previous passes
              $j \leftarrow p * p$
              while $j \leq n$ do
                  $A[j] \leftarrow 0$    //mark element as eliminated
                  $j \leftarrow j + p$
      //copy the remaining elements of $A$ to array $L$ of the primes
      $i \leftarrow 0$
      for $p \leftarrow 2$ to $n$ do
          if $A[p] \neq 0$
              $L[i] \leftarrow A[p]$
              $i \leftarrow i + 1$
      return $L$
```

Overview

Fundamentals of Algorithmic Problem Solving

Algorithms are procedural solutions to problems.

- Understanding the Problem
- Ascertaining Capabilities of a Computational Device
- Choosing between Exact and Approximate Problem Solving
- Deciding on Appropriate Data Structures
- Algorithm Design Techniques
- Methods of Specifying an Algorithm
- Proving an Algorithm's Correctness
- Analyzing the Algorithm
- Coding the Algorithm

Algorithm Design Techniques/Strategies

- Brute force/ Exhaustive Search
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

Basic Issues Related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness
- Efficiency (or complexity) analysis
- Optimality

Why study algorithms?

- core of computer science
- Provides framework for designing and analyzing algorithms for new problems

Overview

Important Problem Types

These problems are used to illustrate different algorithm design techniques and methods of algorithm analysis.

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

Important Problem Types

The **sorting problem** asks to rearrange the items of a given list in ascending order.

- A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input.
- An algorithm is said to be **in place** if it does not require extra memory, except, possibly for a few memory units.

The **searching problem** deals with finding a given value, called a **search key** in a given set or a multiset.

Important Problem Types

String Processing

- A **string** is a sequence of characters from an alphabet (e.g. text strings, bit strings, gene sequences).

Graph Problems

- One of the oldest and most interesting areas in algorithmics.
- A **graph** is a collection of points called **vertices** some of which are connected by line segments called **edges**.

Important Problem Types

Combinatorial problems ask (explicitly or implicitly) to find a combinatorial object - such as a permutation, a combination or a subset - that satisfies certain constraints and has some desired property.

- **Geometric algorithms** deal with geometric objects such as points, lines and polygons.
- Two example of geometric problems are closest-pair problem and convex-hull problem.

Numerical problems involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integral, evaluating functions and so on.

Overview

Notion of an Algorithm

Algorithmic Problem Solving

Important Problem Types

Fundamental Data Structures: A Review

Fundamental Data Structures

A **data structure** can be defined as a particular scheme of organizing related data items.

Linear Data Structures

A (one-dimensional) **array** is a sequence of n items of the same data type that are stored contiguously in a computer memory and made accessible by specifying a value of the array's index.

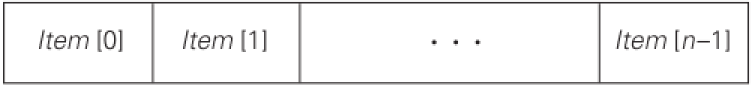


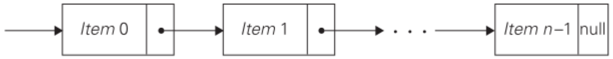
Figure: Array of n elements

Linear Data Structures

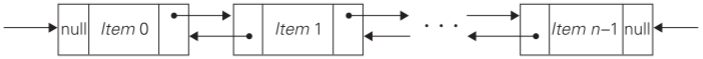
- A **linked list** is a sequence of zero or more elements called **nodes** each containing two kinds of information: some data and one or more links (called **pointers**) to other nodes of the link list.
- A special kind pointer called **null** is used to indicate the absence of a node's successor.

Linear Data Structures

In **singly linked list** each node except the last one contains a single pointer to the next element.



In a **doubly linked list**, every node, except the first and the last, contains pointers to both it successor and predecessor.



Linear Data Structures

- A **list** is a finite sequence of data items, i.e. a collection of data items arranged in a certain linear order.
- Two special types of lists are stacks and queues.
- A **stack** is a list in which insertions and deletions can be done only at the end.
- A **queue** is a list from which elements are deleted from one end of the structure, called the **front** (this operation is called **dequeue**) and new elements are added to the other end called the **rear** (this operation is called **enqueue**).

Graphs

A **graph** $G = \langle V, E \rangle$ is defined by a pair of two sets: finite set V of items called **vertices** and a set E of pairs of these items called **edges**.

- If a pair of vertices (u, v) is the same as the pair (v, u) we say that the vertices u and v are **adjacent** to each other and that the y are connected by the the **undirected edge** (u, v).
- A graph G is called **undirected** if every edge in it is undirected.

Graphs

- If a pair of vertices (u, v) is not the same as the pair (v, u) we say that the edge (u, v) is directed from the vertex u, called the **edge's tail** to the vertex v called the **edge's head**.
- A graph whose every edge is directed is called **directed graph** (or **digraph**).

Theorem
The number of edges $|E|$ possible in an undirected graph with $|V|$ vertices and no loops is given by the inequality

$$0 \le |E| \le |V|(|V| - 1)/2.$$

Graphs

- A graph with every pair of its vertices connected by an edge is called **complete**.
- A standard notation for the complete graph with $|V|$ vertices is $K_{|V|}$.

- A graph with relatively few possible edges missing is called **dense**.
- A graph with few edges relative to the number of its vertices is called **sparse**.

Graph Representations

- Graphs for computer algorithms can be represented in two principal ways: the adjacency matrix and the adjacency lists.

The **adjacency matrix** A of a graph with n vertices is an n-by-n boolean matrix with one row and one column for each graph's vertices, where

$$A[i,j] = \begin{cases} 1 & \text{if there is an edge from the } i^{th} \text{ vertex to the } j^{th} \text{ vertex} \\ 0 & \text{otherwise} \end{cases}$$

Graph Representations

- Graphs for computer algorithms can be represented in two principal ways: the adjacency matrix and the adjacency lists.

The **adjacency lists** of a graph or a digraph is a collection of linked lists, one for each vertex that contain all the vertices adjacent to the list's vertex.

Weighted Graphs

- A **weighted graph** is a graph with numbers assigned to its edges.
- These numbers are called **weights** or **costs**.

Remark

- If a weighted graph is represented by its adjacency matrix, then its element $A[i, j]$ will simply contain the weight of the edge from the i^{th} to the j^{th} vertex if there is such an edge and a special symbol, ∞, if there is no such edge.
- Adjacency list for a weighted graph have to include in their nodes not only the name of the an adjacent vertex but also the weight of the corresponding edge.

Graph Properties: Paths and Connectivity

- A **path** from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.
- If all vertices of a path are distinct, the path is said to be **simple**.

The **length** of a path is the total number of edges in the path.

A **directed path** is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next.

Graph Properties: Connectivity and Acyclicity

- A graph is said to be **connected** if for every pair of its vertices u and v there is a path from u to v.
- A **connected component** is a maximal connected subgraph of a given graph.

- A **cycle** is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once.
- A graph with no cycles is said to be **acyclic**.

Trees

- A **tree** (more accurately **free tree**) is a connected acyclic graph.
- A graph that has no cycles but is not necessarily connected is called a **forest**.

The number of edges in a tree is always one less than the number of its vertices.

Trees

- For every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other.
- This property makes it possible to select an arbitrary vertex in a free tree and consider it as the **root** of the so-called **rooted tree**.

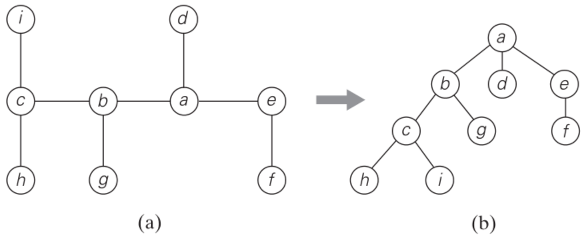


Figure: (a) Free Tree (b) Its transformation into a rooted tree

Trees and Terminologies

- For any vertex v in a tree T, all the vertices on the simple path from the root to that vertex are called **ancestors** of v.
- The vertex itself is usually considered its own ancestor; the set of ancestors that excludes the vertex itself is referred to as **proper ancestors**.
- If (u, v) is the last edge of the simple path from the root to vertex v (and $u \neq v$), u is said to be the **parent** of v and v is called a **child** of u.
- Vertices that have the same parent are said to be **siblings**.

Trees and Terminologies

- A vertex with no children is called a **leaf**; a vertex with at least one child is called **parental**.
- All the vertices for which a vertex v is an ancestor are said to be **descendants** of v; the **proper descendants** exclude the vertex v itself.
- All the descendants of a vertex v with all the edges connecting them form the **subtree** of T rooted at that vertex.
- The **depth** of a vertex v is the length of the simple path from the root to v.
- The **height** of a tree is the length of the longest simple path from the root to a leaf.

Ordered Trees

An **ordered tree** is a rooted tree in which all the children of each vertex are ordered.

A **binary tree** can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a **left child** or **right child** of its parents.

The subtree with its root at the left (right) child of a vertex is called the **left (right) subtree** of that vertex.

Trees

A binary tree in which a number assigned to each parental vertex that is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree is called **binary search trees**.

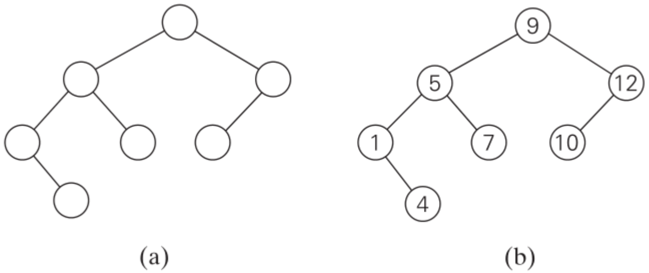


Figure: (a) Binary Tree (b) Binary search tree

Trees

The height h of a binary tree with n nodes is given by the inequality

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1.$$

Remark

- A binary tree is usually implemented for computing purposes by a collection of nodes corresponding to vertices of the tree.
- Each node contains some information associated with the vertex and two pointers to the nodes representing the left child and right child of the vertex, respectively.

Trees

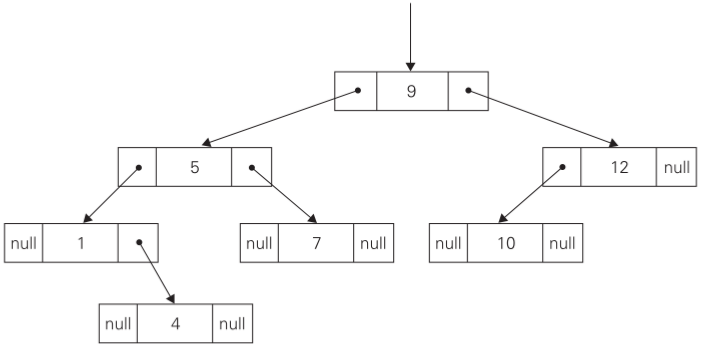
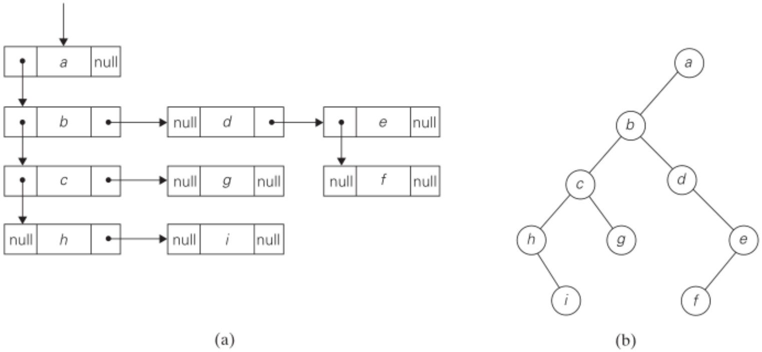


Figure: Standard implementation of the binary search tree

Trees

Figure: (a) First child-next sibling representation of the graph whose binary tree representation is (b)



Sets and Dictionaries

A **set** can be described as an unordered collection (possibly empty) of distinct items called **elements** of the set.

Sets can be implemented in computer applications in two ways:

- The first considers only sets that are subsets of some large set U, called the **universal set** (*bit string representation*).
- The second way to represent a set for computing purposes is to use the list structure to indicate the set's element.

Sets and Dictionaries

A data structure that implements the operations: searching for a given item, adding a new item and deleting an item from the collection, is called **dictionary**.

Abstract data type (ADT) is a set of abstract objects representing data items with a collection of operations that can be performed on them. The *list*, the *stack*, the *queue*, the *priority queue*, and the *dictionary* are important examples of abstract data types. Modern object-oriented languages support implementation of ADTs by means of **classes**.

End of Lecture