Analysis of Algorithms

Chapter 1.1, 1.2, 1.3, 1.4

Syllabus

FOLLOW THE WEB SITE FOR ANNOUNCEMENTS AND CHANGES

Web Page

http://pdc.ege.edu.tr/pdcworks/courses/352/352.html

Textbook

- Introduction to The Design and Analysis of Algorithms, Anany Levitin, Addison Wesley, 3rd edition.

Other Books

- Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press; 3rd edition, 2009 ISBN-10: 0262033844 ISBN-13: 978-0262033848

Syllabus-Catalog Description:

- Basic definitions and data structures.
- Introduction to analysis of algorithms.
- Standard algorithm design techniques;
  - divide-and-conquer,
  - greedy,
  - dynamic programming,
  - etc.
- Basic algorithms;
  - sorting and searching,
  - graph algorithms,
  - etc.
- Introduction to complexity classes.

Syllabus - Educational Objectives:

This course introduces basic algorithms, algorithm design and analysis techniques which can be used in designing solutions to real life problems. After this course, you will

- able to design a new algorithms for a problem using the methods discussed in the class
- able to analyze an algorithm with respect to various performance criteria such as memory use and running time
- able to choose the most suitable algorithm for a problem to be solved,
- able to implement an algorithm efficiently.

Syllabus - Attendance Policy:

- Class attendance is advised but will not be a part of your final grade. However, a minimum of 70% attendance is required. DO NOT SIGN FOR OTHER STUDENTS.

- Please be considerate of your classmates during class. Students are expected to show courtesy and respect toward their classmates.

- Please do not carry on side discussions with other students during lecture time – when you have a question, please raise your hand and ask the question so that everyone may benefit from it.

- Also, please make sure that your cellular phone and/or pager does not interrupt during lecture time, and especially during exams.

Bazı Önemli Uyarılar

- %70 devam şartı bulunmaktadır. Müfredat 15 hafta olduğundan toplam devamsızlık maksimum 5 hafta olabilir. <u>BAŞKA ARKADAŞLARINIZIN YERİNE İMZA ATMAYINIZ.</u>

- Sınıfa karşı saygılı olunuz. <u>DERSE ZAMANINDA GELİNİZ.</u> Ders başladıktan sonra gelip dikkat dağıtmayınız.

- <u>DERS ESNASINDA ARANIZDA TARTIŞMAYINIZ. SORUNUZ VARSA BANA SORUNUZ.</u>

- <u>DERS ESNASINDA TELEFONLARINIZI KAPATINIZ VEYA SESSİZE ALINIZ. TELEFONUNUZLA MEŞGUL OLMAYINIZ.</u>

Syllabus - Assessment:

MIDTERM **% 50**

FINAL EXAM **% 50**

ROAD MAP

- **Introduction**
  - **Definition and Properties of Algorithm**
  - **Fundamentals of Algorithmic Problem Solving**
  - **Important Problem Types**
- Fundamental Data Structures
  - Linear Data Structures
  - Graphs
  - Trees
- Mathematical Background

Introduction

- Why do you need to study algorithms?
- There are both practical and theoretical reasons to study algorithms.
- From a practical standpoint, you have to know a standard set of important algorithms from different areas of computing; in addition, you should be able to design new algorithms and analyze their efficiency.
- From the theoretical stand-point, the study of algorithms, sometimes called algorithmics, has come to be recognized as the cornerstone of computer science.

Introduction

Algorithms are “methods for solving problems which are suited for computed implementation.” [Sedgewick]

An algorithm is “a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.” [Aho, Hopcroft, & Ulman]

Introduction

"*Algorithmics* [defined as the study of algorithms -- A.L.] is more than a branch of computer science. <u>It is the core of computer science</u>, and, in all fairness, can be said to be relevant to most of science, business, and technology." [David Harel, "Algorithmics: The Spirit of Computing"]

What is an Algorithm ?

An *algorithm* is a finite, clearly specified sequence of instructions to be followed to solve a problem or compute a function

An *algorithm* generally

- takes some input
- carries out a number of effective instructions in a finite amount of time
- produces some output.

An effective instruction is an operation so basic that it is possible to carry it out using pen and paper.

Donald E. Knuth
Professor Emeritus of The Art of Computer Programming at Stanford University
Knuth has been called the "father" of the analysis of algorithms

- A person well-trained in computer science knows how to deal with algorithms:
  - how to construct them,
  - manipulate them,
  - understand them,
  - analyze them.
- It has often been said that a person does not really understand something until after teaching it to someone else.
- Actually, a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm . . .
- An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way.

Two main issues related to algorithms

- How to design algorithms

- How to analyze algorithm efficiency

Expressing Algorithms

Algorithms can be expressed in

- natural languages
  - verbose and ambiguous
  - rarely used for complex or technical algorithms
- pseudocode, flowcharts
  - structured ways to express algorithms
  - avoid ambiguities in natural language statements
  - independent of a particular implementation language
- programming languages
  - intended for expressing algorithms in a form that can be executed by a computer
  - can be used to document algorithms

Example:

Problem: Find the largest number in an (unsorted) list of numbers.

Idea: Look at every number in the list, one at a time.

Natural Language:

- Assume the first item is largest.
- Look at each of the remaining items in the list and if it is larger than the largest item so far, make a note of it.
- The last noted item is the largest in the list when the process is complete.

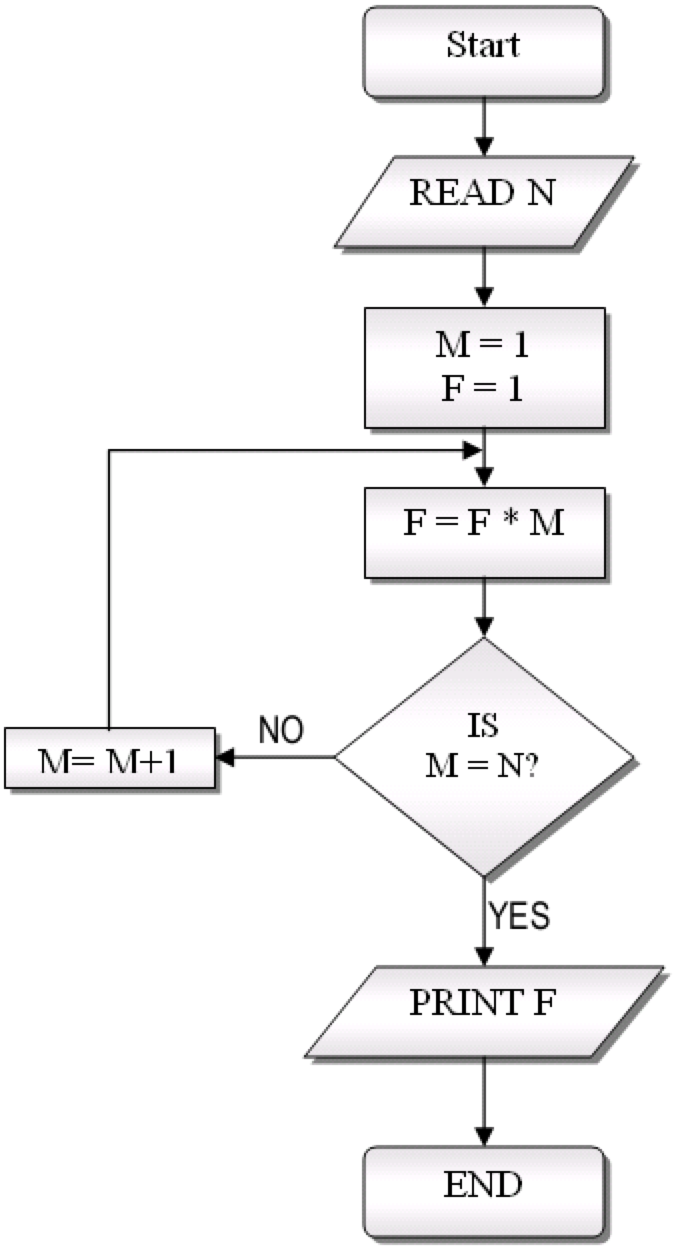
Example:

Pseudocode:

Algorithm LargestNumber
    Input: A non-empty list of numbers *L*.
    Output: The *largest* number in the list *L*.
    largest ← L_0
    for each *item* **in** the list $L_{i \geq 1}$, **do**
        if the *item* > *largest*, **then**
            largest ← the *item*
    return *largest*

Example:

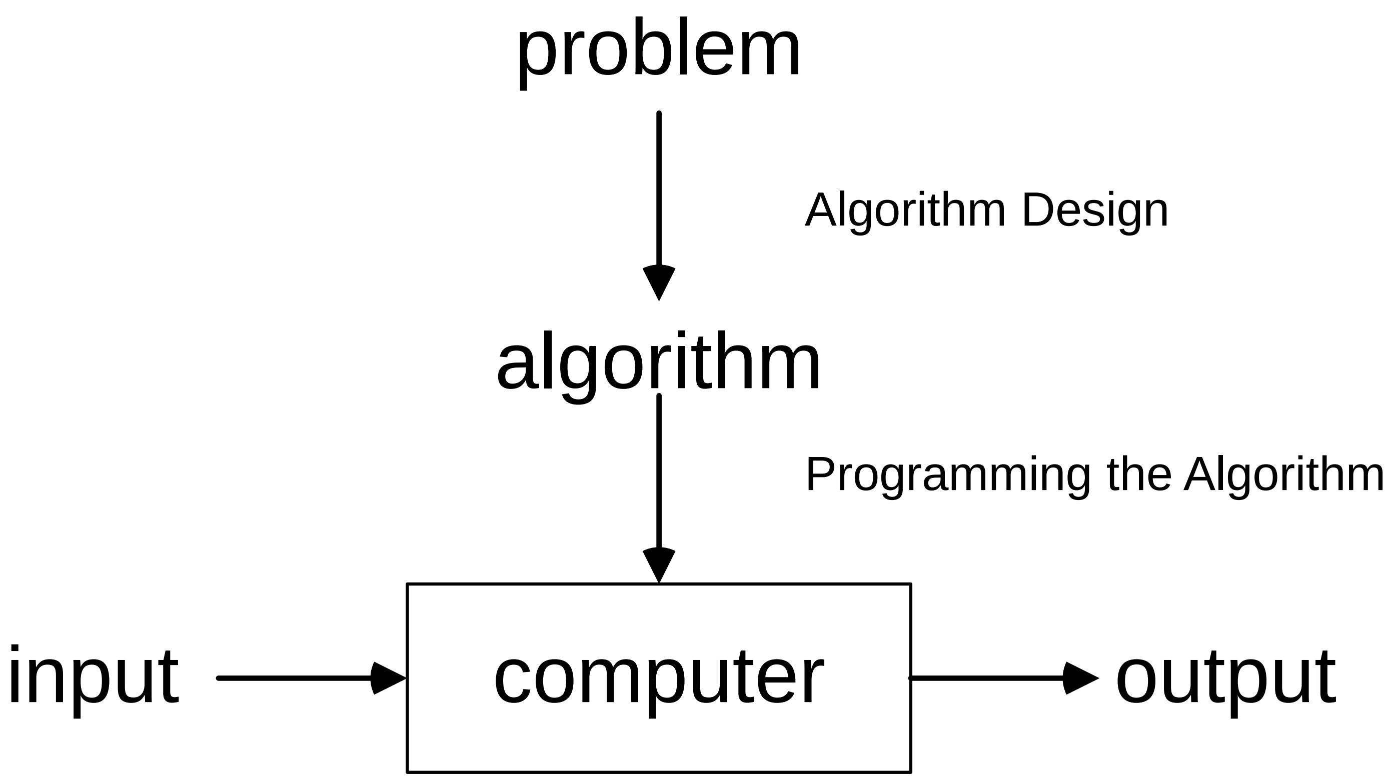
Flowchart:



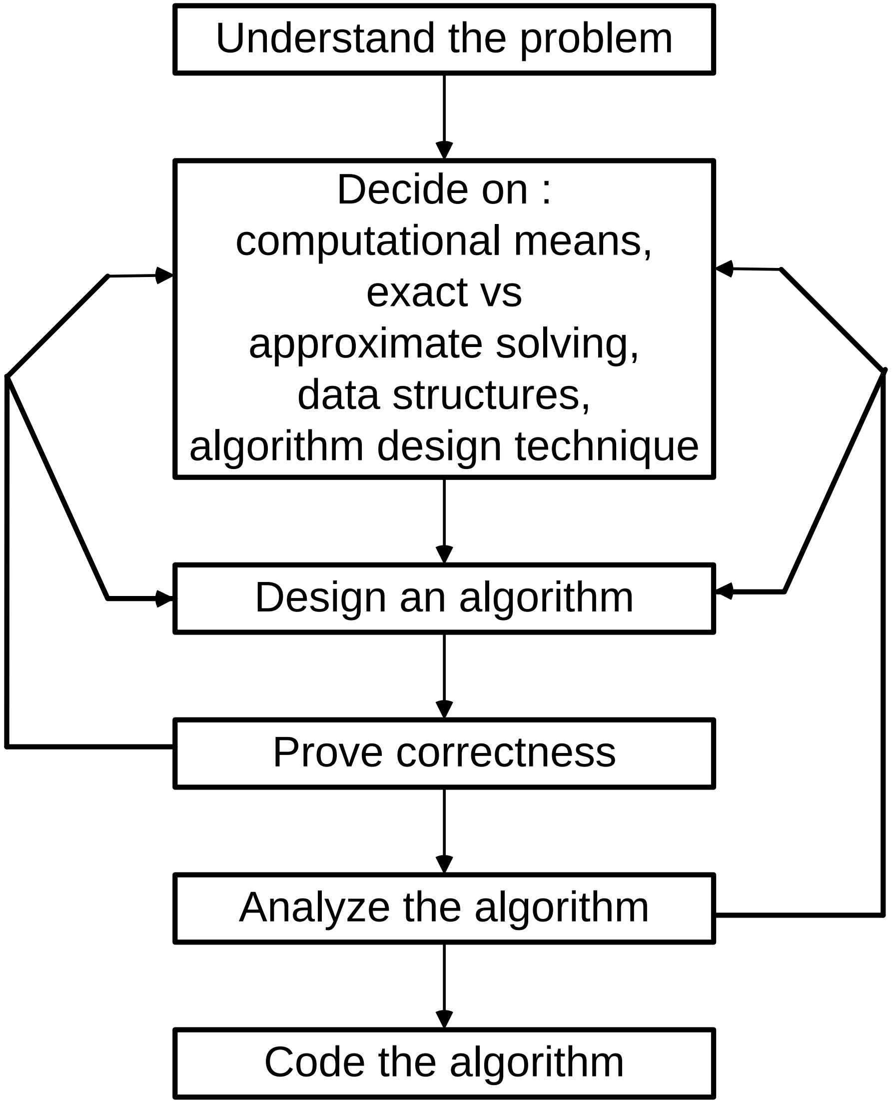
Properties of an Algorithm

- **Effectiveness**
  - Instructions are simple
    - can be carried out by pen and paper
- **Definiteness**
  - Instructions are clear
    - meaning is unique
- **Correctness**
  - Algorithm gives the right answer
    - for all possible cases
- **Finiteness**
  - Algorithm stops in reasonable time
    - produces an output

Notion of an Algorithm



Algorithm Design Process



Deciding on Appropriate Data Structures

Algorithms + Data Structures = Programs

What is an Algorithm ?

An *algorithm* is a finite, clearly specified sequence of instructions to be followed to solve a problem or compute a function

An *algorithm* generally

- takes some input
- carries out a number of effective instructions in a finite amount of time
- produces some output.

An effective instruction is an operation so basic that it is possible to carry it out using pen and paper.

Euclid’s Algorithm

- **Problem:** Find gcd(m,n), the greatest common divisor of two nonnegative, not both zero integers m and n

- Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

- Euclid’s algorithm is based on repeated application of equality

  gcd(m,n) = gcd(n, m mod n)

- until the second number becomes 0, which makes the problem trivial.

  Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

Structured Description of Euclid's Algorithm

- **Step 1** If $n = 0$, return m and stop; otherwise go to Step 2
- **Step 2** Divide m by n and assign the value to the remainder to r
- **Step 3** Assign the value of n to m and the value of r to n. Go to Step 1.

Euclid's Algorithm (Pseudocode)

ALGORITHM *Euclid*(m, n)

//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while $n \neq 0$ **do**
$r \leftarrow m \bmod n$
$m \leftarrow n$
$n \leftarrow r$
return m

Consecutive integer checking algorithm

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

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

Step 3 Divide n by t. If the remainder is 0, return t and stop; otherwise, go to Step 4

Step 4 Decrease t by 1 and go to Step 2

Middle-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 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd*(m,n*)

60 = 2 x 2 x 3 x 5

24 = 2 x 2 x 2 x 3

gcd(60, 24) = 2 x 2 x 3 = 12

- *Is this an algorithm?*

Sieve of Eratosthenes

- **A simple Algorithm Generating Consecutive Primes Not Exceeding Any Given Integer n: Sieve of Eratosthenes**
- Example:

2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
2	3	x	5	x	7	x	9	x	11	x	13	x	15	x	17	x	19	x	21	x	23	x	**25**
2	**3**		5		7		x		11		13		x		17		19		x		23		**25**
2	**3**		**5**		**7**				**11**		**13**				**17**		**19**				**23**		**x**

Sieve of Eratosthenes

ALGORITHM *Sieve*(n)

```
    //Implements the sieve of Eratosthenes
    //Input: An integer n ≥ 2
    //Output: Array L of all prime numbers less than or equal to n
    for p ← 2 to n do A[p] ← p
    for p ← 2 to ⌊√n⌋ do    //see note before pseudocode
        if A[p] ≠ 0   //p hasn't been eliminated on previous passes
            j ← p * p
            while j ≤ n do
                A[j] ← 0    //mark element as eliminated
                j ← j + p
    //copy the remaining elements of A to array L of the primes
    i ← 0
    for p ← 2 to n do
        if A[p] ≠ 0
            L[i] ← A[p]
            i ← i + 1
    return L
```

Algorithm design techniques/strategies

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

Analysis of algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency

- Does there exist a better algorithm?
  - lower bounds
  - optimality

Important problem types

- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

ROAD MAP

- Introduction
  - Definition and Properties of Algorithm
  - Fundamentals of Algorithmic Problem Solving
  - Important Problem Types
- **Fundamental Data Structures**
  - **Linear Data Structures**
  - **Graphs**
  - **Trees**
- Mathematical Background

Fundamental data structures

- list
  - array
  - linked list
  - string
- stack
- queue
- priority queue
- graph
- tree
- set and dictionary

Fundamental Data Structures

- A ***data structure*** is a particular scheme of organizing related data items
  - Linear Data Structures
    - Array
    - Linked list
    - Stack
    - Queue
    - Prioritiy Queue
  - Graphs
  - Trees

Linear Data Structures

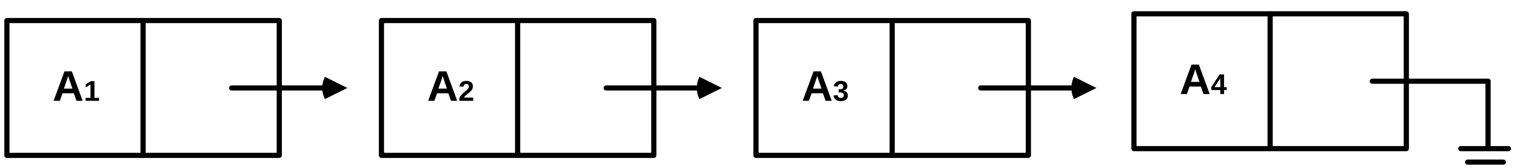
- **Array**
  - An *array* is a sequence of *n* items of the same data type that are stored contiguously in computer memory
  - Array is accessible by specifying a value of the array's *index*

Item[0]	*Item[1]*	…	*Item[n-1]*

Array of *n* elements

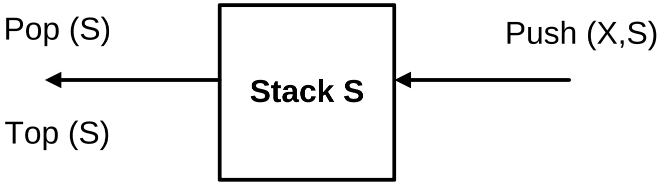
Linear Data Structures

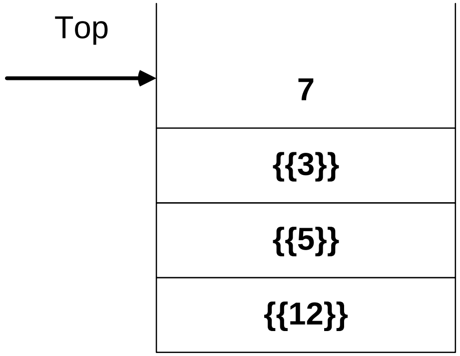
- **Linked List**
  - A *linked list* is a sequence of zero or more elements called *nodes*
  - Each node contains two kinds of information :
    - some data
    - one or more linkes called *pointers* to other nodes of the linked list



Linear Data Structures

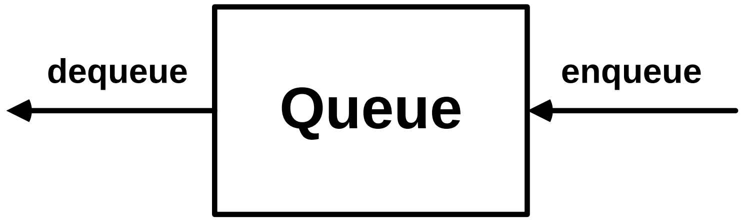
- **Stack**
  - A *stack* is a list in which insertions and deletions can be done only at the end (called *top)*
  - Last In First Out (LIFO)





Linear Data Structures

- **Queue**
  - A *queue* is a list whose elements are deleted from one end of the structure, called *front*
    - *dequeue operation*
  - New elements are added to the other end of the queue, called *rear*
    - *enqueue operation*
  - First In First Out (FIFO)



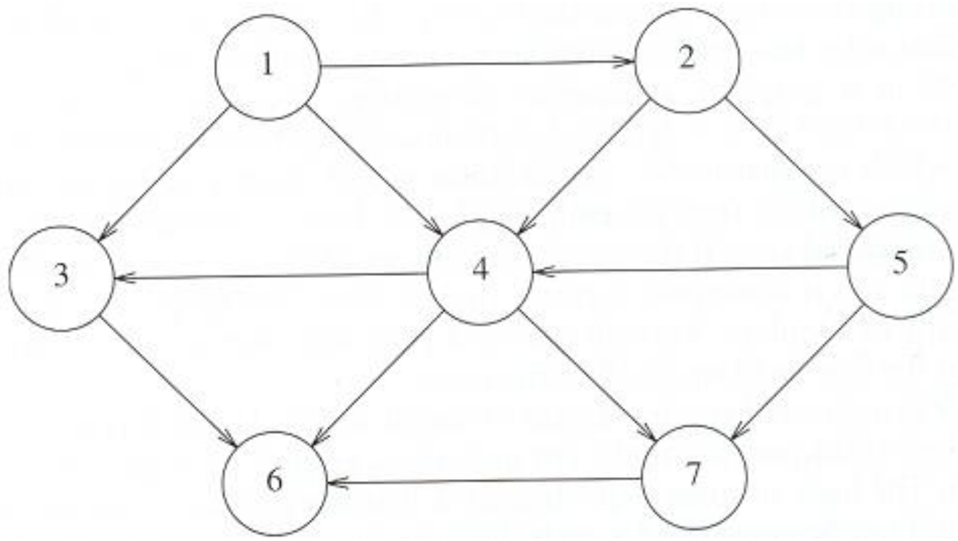
Linear Data Structures

- **Priority Queue**
  - A *priority queue* is a collection of data items from a totally ordered universe
    - e.g. integer or real numbers
  - Requires a selection of an item of the highest priority among a dynamically changing set of candidates
  - Principal operations:
    - Insert → adding a new element
    - Delete → deleting largest/smallest element
    - Search → find largest/smallest element
  - A better implementations of a priority queue is based on an ingenious data structure called ***heap***

Graphs

- A *graph* is a tuple *G=(V,E)*
  - V : set of *vertices* or *nodes*
  - E : set of *edges*
    - each edge is a pair (*v,w*), where *v,w* Є V
  - If the pairs are ordered, then the graph is *directed*
    - directed graphs are sometimes refered as *digraphs*
  - If the pairs are unordered, then the graph is *undirected*
  - Vertex *w* is adjacent to *v* iff *(v,w) Є E*

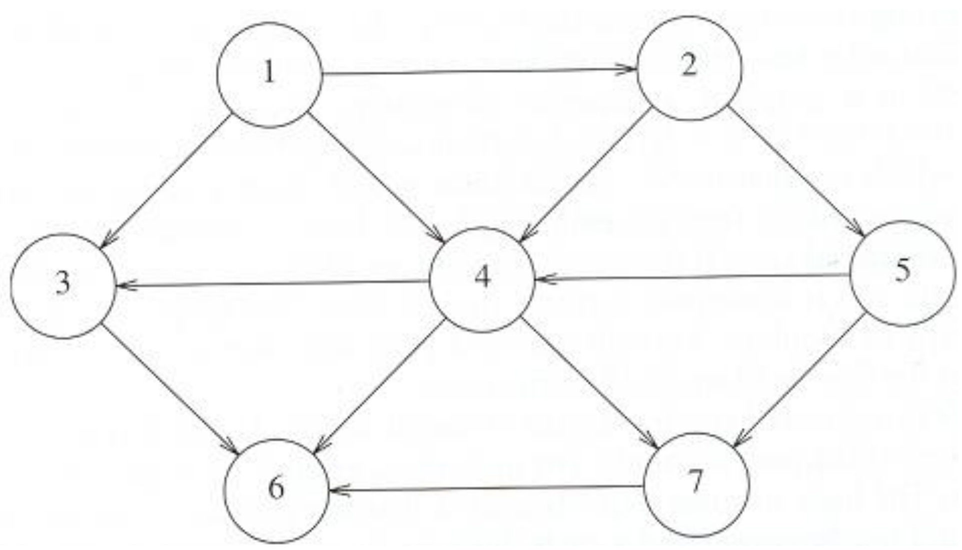
Graphs



A graph with 7 vertices and 12 edges

Graphs

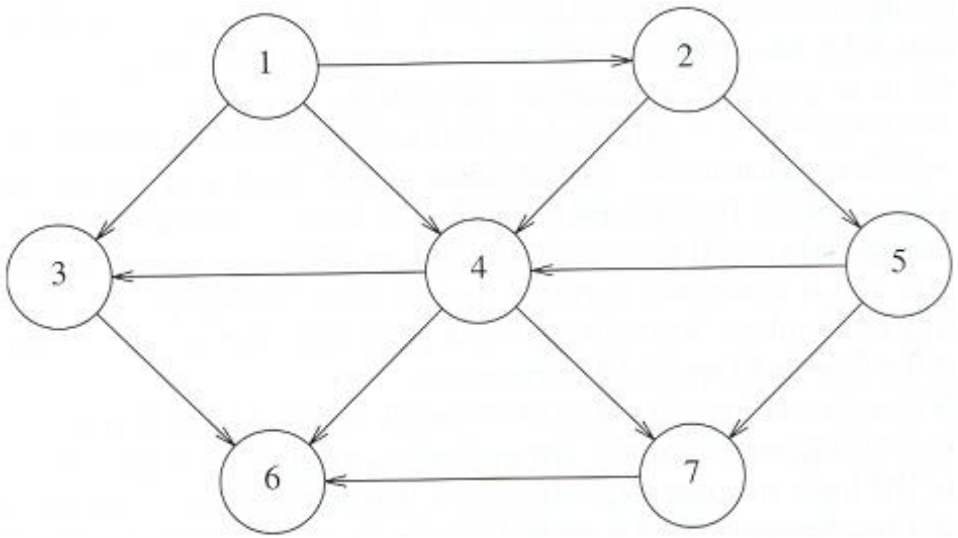
- A *path* is a sequence of vertices from *v* to *w*
- If all edges in a path are distinct the path is said to be *simple*
- *Lenght* is the total number of edges in a path
- A *cycle* is a path with *length ≥ 1* where *u=w*
- A graph is *acyclic* if it has no cycles



A graph with 7 vertices and 12 edges

Graphs

- A graph with every pair of its vertices connected by an edge is called *complete*
- 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 *sparce*
- A graph is *connected* if for every pair of vertices *u* and *v* there is a path from *u* to *v*



A graph with 7 vertices and 12 edges

Graph Representations

Adjacency Matrix Representation

- It is *n-by-n* boolean matrix with one row and one column for each of the graph's vertices
  - *ith* row and *jth* column is equal to 1 if there is an edge from the *ith* vertex to the *jth* vertex
  - *ith* row and *jth* column is equal to 0 if there is no such edge

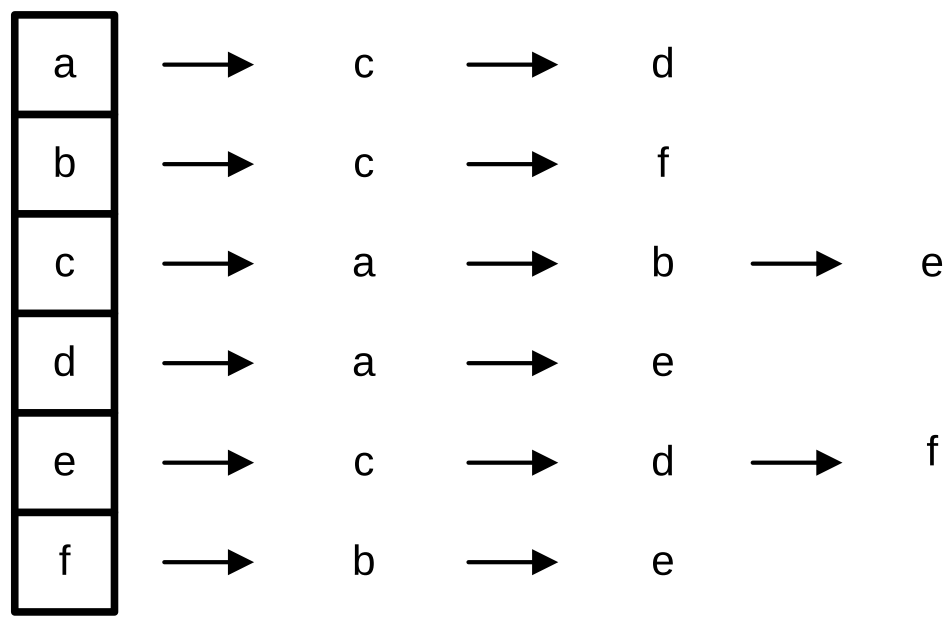
$$
\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array}
\begin{array}{c}
\begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$

Graph Representations

Adjacency List Representation

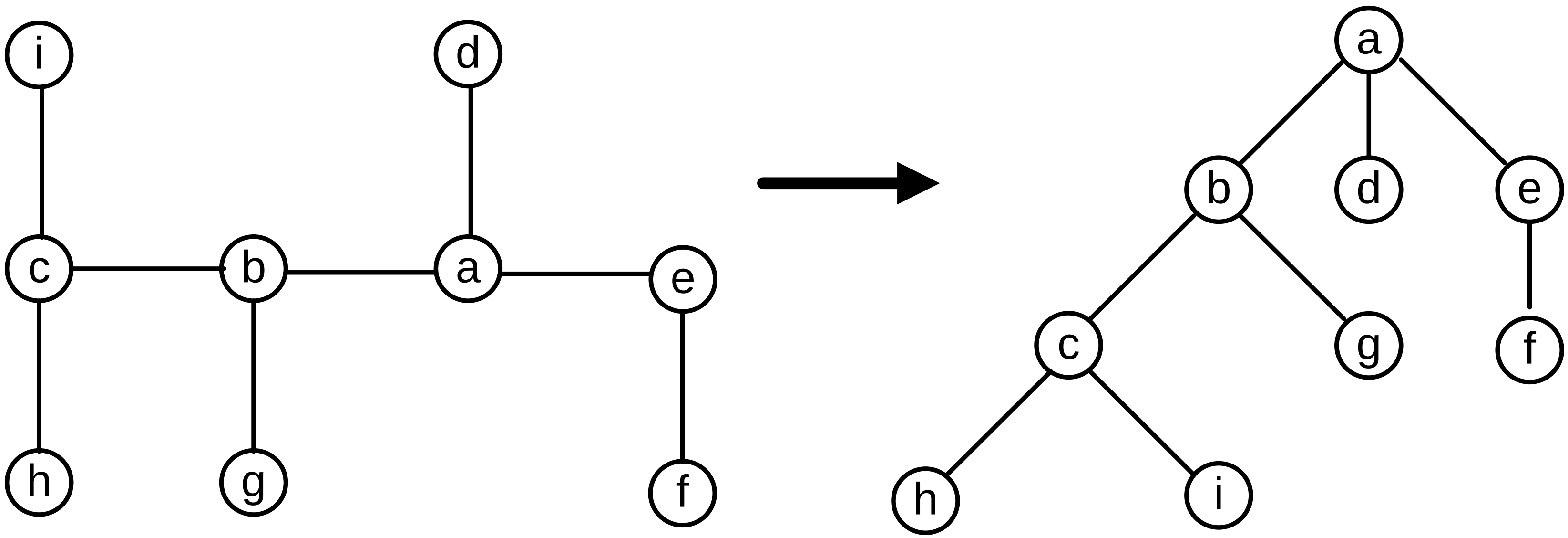
- Is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex

- If the graph is not dense (is *sparse*) adjacency list representation is a better solution



Trees

- A tree is a connected acyclic graph
    - *rooted tree*
        - Specialized node caled root



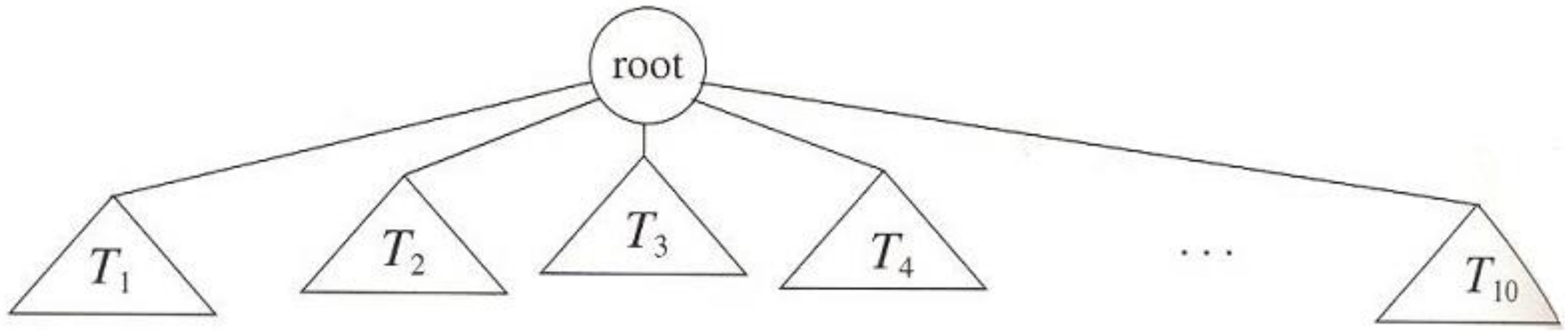
tree

rooted tree

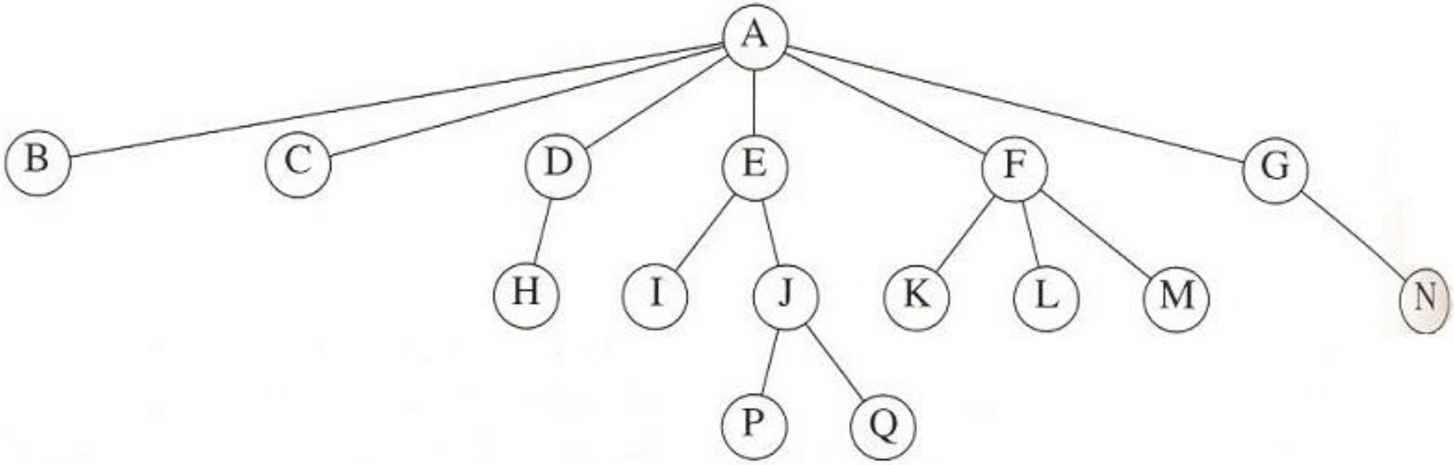
Trees

Recursive Definition of Rooted Trees:

- **Tree** is a collection of **nodes**
  - A tree can be empty
  - A tree contains zero or more **subtrees** T_1, T_2,… T_k connected to a **root node** by **edges**

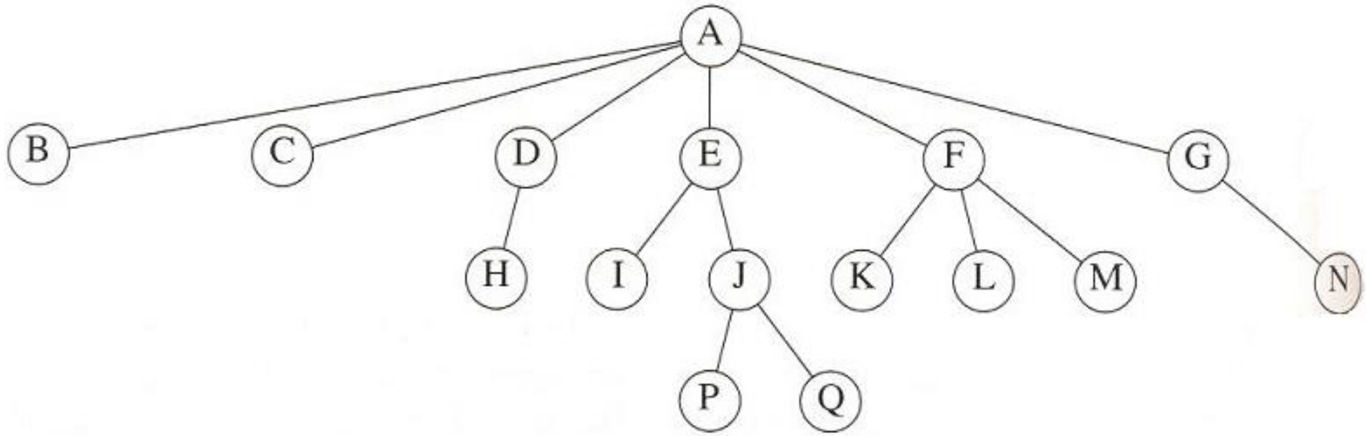


Trees - Terminology



Family Tree Terminology

- child → F is child of A
- parent → A is the parent of F
  - each node is connected to a parent except the root
- sibling → nodes with same parents (K, L, M)
- leaf → nodes with no children (P, Q)
- Ancestor / Descendant



- **Path :** a list of distinct vertices in which successive vertices are connected by edges in the tree. There is exactly one path between the root and the other nodes in tree.
- **Lenght :** number of edges on the path (k-1)
- **Depth :** depth of n_i is the lenght of unique path from the root to n_i
  - depth of root is 0
  - depth of a tree = depth of the deepest leaf
- **Height :** height of n_i is the lenght of the longest path from n_i to a leaf
  - height of a leaf is 0
  - height of a tree = height of the root = depth of the tree

Trees

- Ordered tree
  - A rooted tree in which all the children of each vertex are ordered
- Binary tree
  - An ordered tree in which every vertex has no more than two children
  - Each child designated as either a *left child* or a *right child* of its parent

Trees

- Binary Search Tree (BST)
  - A binary tree
    - No repeated element
  - Satisfies Search Tree Property
    - Elements on left subtree is smaller than root
    - Elements on right subtree is greater than root
    - Left and right subtrees are BST
- Heap
  - An implementation of priority queue
  - A binary tree
  - Satisfies heap order property
    - Each element is larger than the parent (min-heap)
    - Each element is smaller than the parent (max-heap)

ROAD MAP

- Introduction
  - Definition and Properties of Algorithm
  - Fundamentals of Algorithmic Problem Solving
  - Important Problem Types
- Fundamental Data Structures
  - Linear Data Structures
  - Graphs
  - Trees
- **Mathematical Background**

Mathematical Background

- Functions
- Logarithm
- Summation
- Probability
- Asymptotic Notations
- Recursion
  - Recurrence equation

Properties of Logarithms

1. $\log_a 1 = 0$
2. $\log_a a = 1$
3. $\log_a x^y = y \log_a x$
4. $\log_a xy = \log_a x + \log_a y$
5. $\log_a \dfrac{x}{y} = \log_a x - \log_a y$
6. $a^{\log_b x} = x^{\log_b a}$
7. $\log_a x = \dfrac{\log_b x}{\log_b a} = \log_a b \log_b x$

Important Summation Formulas

1. $\sum_{i=l}^{u} 1 = \underbrace{1 + 1 + \cdots + 1}_{u-l+1 \text{ times}} = u - l + 1$ (l, u are integer limits, $l \leq u$); $\sum_{i=1}^{n} 1 = n$

2. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$

3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

Important Summation Formulas

5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \ (a \neq 1); \quad \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

6. $\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$

7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma$, where $\gamma \approx 0.5772\ldots$ (Euler's constant)

8. $\sum_{i=1}^{n} \lg i \approx n \lg n$

Sum Manipulation Rules

1. $$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$$

2. $$\sum_{i=l}^{u}(a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$$

3. $$\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i, \text{ where } l \le m < u$$

4. $$\sum_{i=l}^{u}(a_i - a_{i-1}) = a_u - a_{l-1}$$