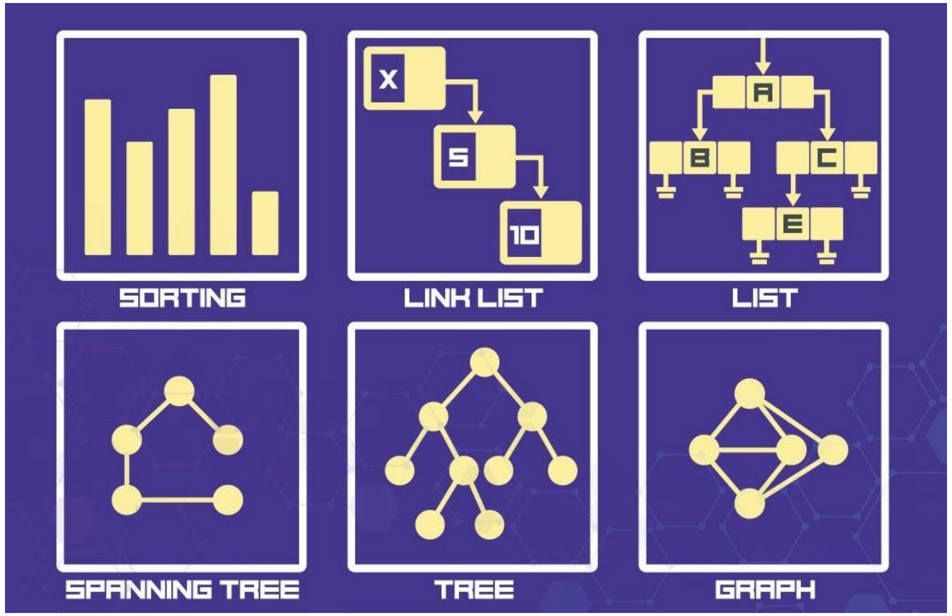
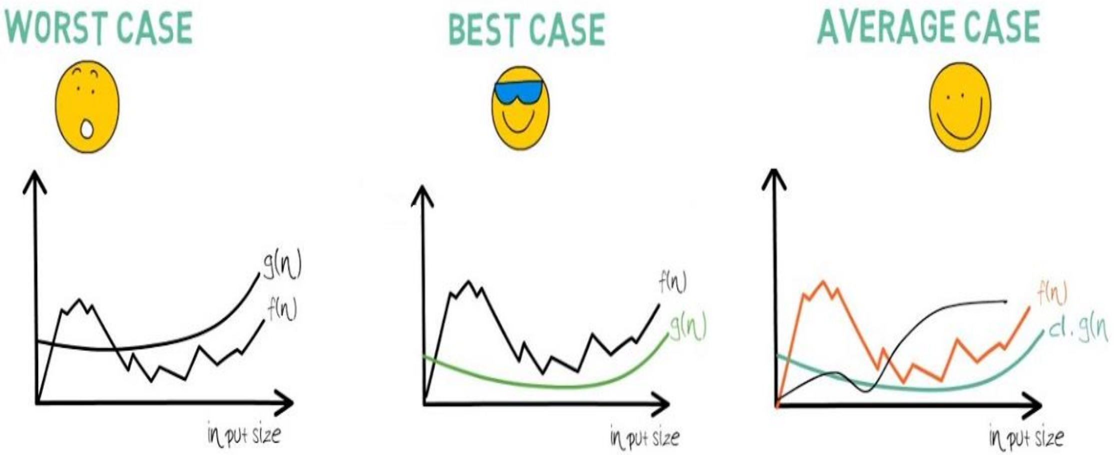
DSE 2256 DESIGN & ANALYSIS OF ALGORITHMS

Lecture 2 & 3 :

Review of Data Structures & Fundamentals of Analysis of Algorithms





Recap of L0 & L1

- What is an algorithm?
- Requirements of an algorithm
- GCD example
- Algorithm design & analysis process
- How good is the algorithm?
- Important problem types

Fundamental data structures I

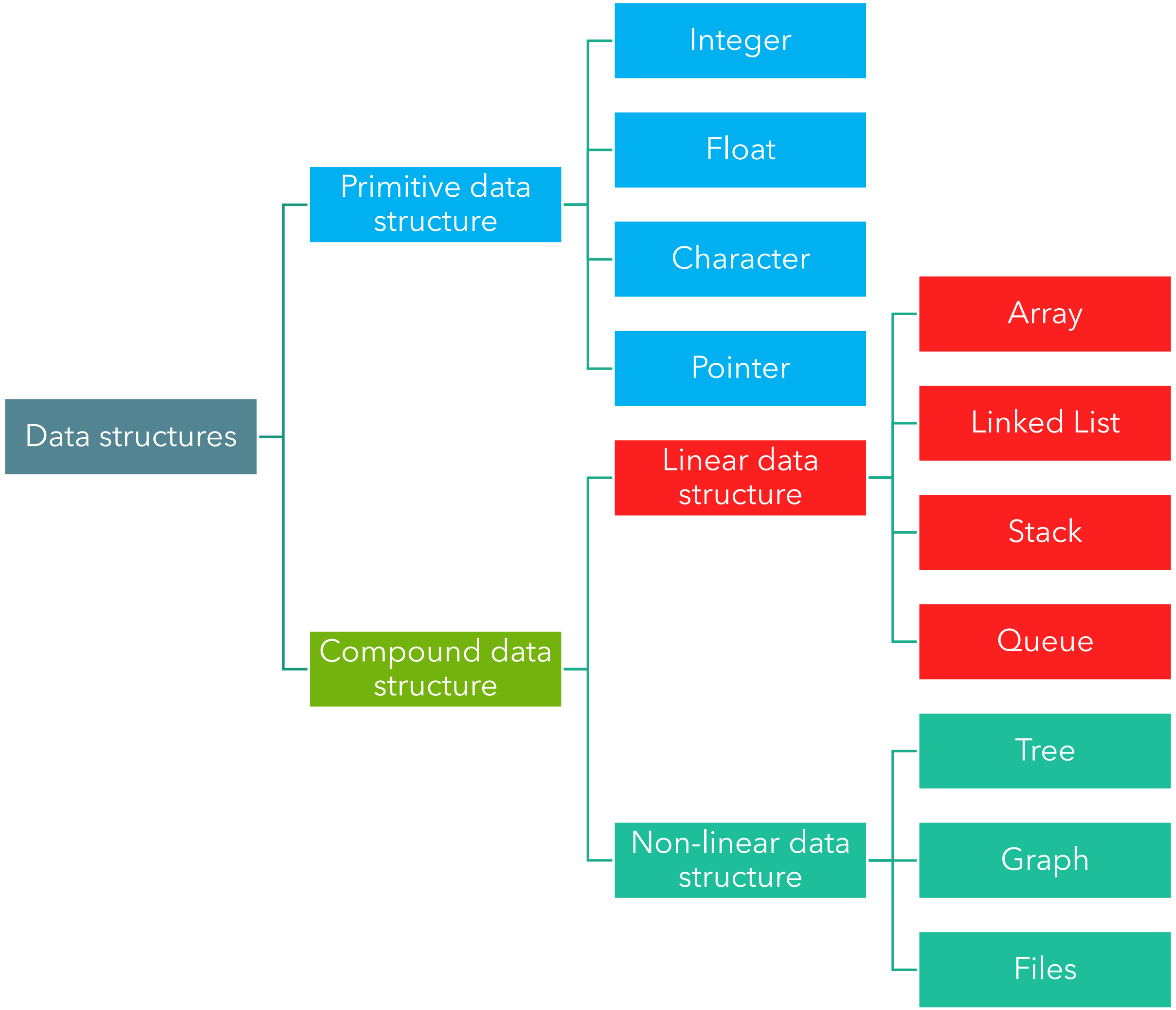
- A **data structure** can be defined as a particular scheme of organizing related data items.

- **Linear Data Structures:** A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached.
  - Continuous arrangement of data elements in the memory.
  - Relationship of adjacency is maintained between the data elements.
  - Eg: Arrays, Linked Lists

Fundamental data structures II

- **Non-Linear Data Structures:** Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure.
    - Collection of randomly distributed set of data item joined together by using a special pointer (tag).
    - Relationship of adjacency is not maintained between the data elements.
    - Eg: Trees, Graphs

Fundamental data structures III



List of fundamental data structures

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

Arrays

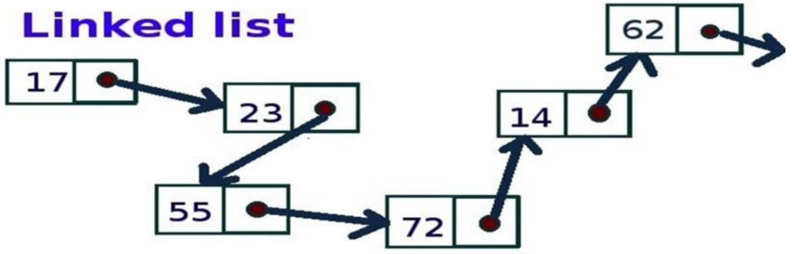
- **Arrays**: A sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index.
  - fixed length (need preliminary reservation of memory)
  - contiguous memory locations
  - direct access

Array of Integers									
0	1	2	3	4	5	6	7	8	9
5	6	4	3	7	8	9	2	1	2

Array of Character									
0	1	y	3	4	5	h	7	8	9
a	6	4	k	7	8	9	q	1	2

Linked List

- **Linked List**: A sequence of zero or more nodes each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.
  - Singly linked list (next pointer)
  - Doubly linked list (next + previous pointers)



- Arrays
  - fixed length (need preliminary reservation of memory)
  - contiguous memory locations
  - direct access
  - Insert/delete
- Linked Lists
  - dynamic length
  - arbitrary memory locations
  - access by following links
  - Insert/delete

Stacks and Queues

Stacks

Eg: A stack of plates

Insertion/deletion can be done only at the top.

LIFO

Two operations (push and pop)

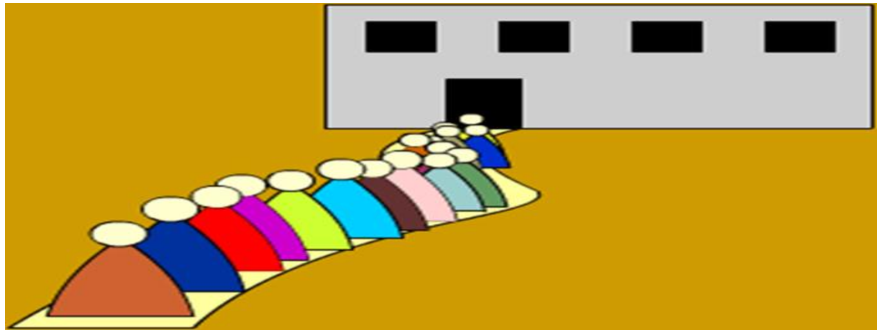
Queues

Eg: A queue of customers waiting for services

Insertion/enqueue from the rear and deletion/dequeue from the front.

FIFO

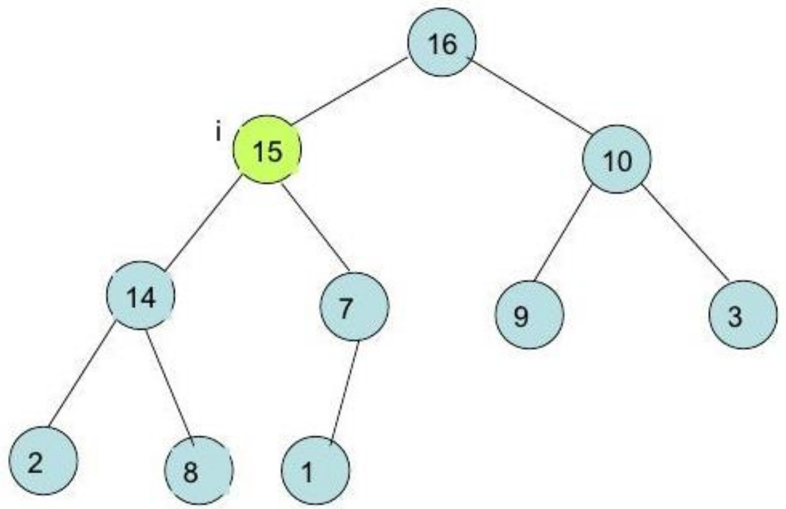
Two operations (enqueue and dequeue)



Priority Queue and Heap

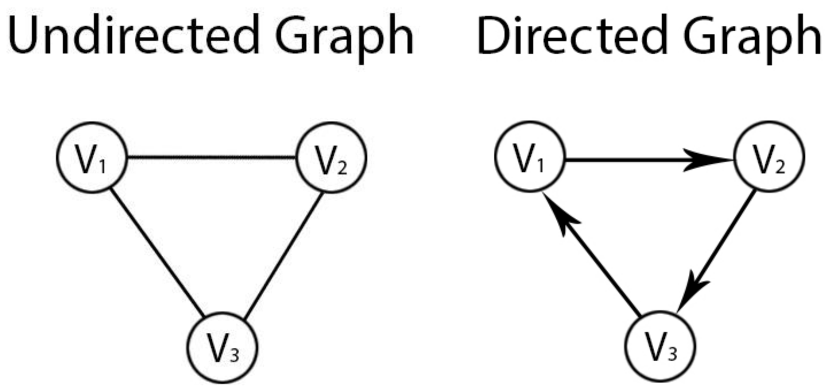
Priority queues (implemented using heaps)

- A data structure for maintaining a set of elements, each associated with a key/priority, with the following operations
  - Finding the element with the highest priority
  - Deleting the element with the highest priority
  - Inserting a new element
  - Scheduling jobs on a shared computer

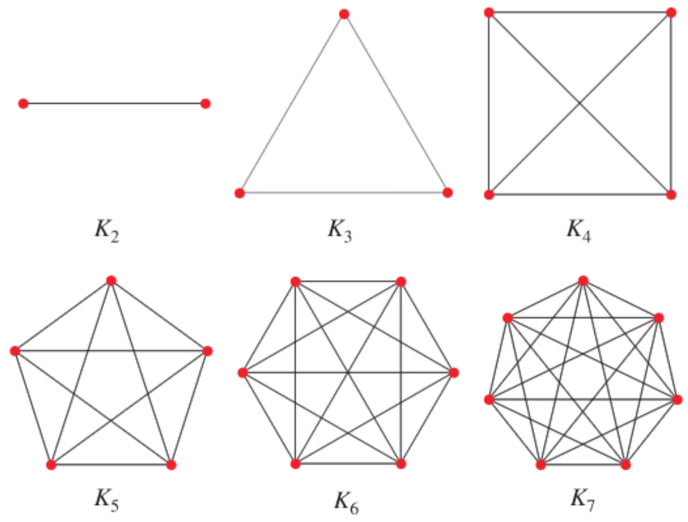


Graphs

- A graph $G = <V, E>$ is defined by a pair of two sets: a finite set V of items called vertices and a set E of vertex pairs called edges.
  - Undirected and directed graphs (digraphs).
  - Complete, dense, and sparse graphs



- A graph with every pair of its vertices connected by an edge is called complete graph, $K_{|V|}$



Graph Representation

Adjacency matrix

n x n boolean matrix if |V| is n.

The element on the ith row and jth column is 1 if there's an edge from ith vertex to the jth vertex; otherwise 0.

The adjacency matrix of an undirected graph is symmetric.

Adjacency linked lists

A collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex.

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

$$
\begin{array}{|c|l}
\hline
a & \rightarrow c \rightarrow d \\
\hline
b & \rightarrow c \rightarrow f \\
\hline
c & \rightarrow a \rightarrow b \rightarrow e \\
\hline
d & \rightarrow a \rightarrow e \\
\hline
e & \rightarrow c \rightarrow d \rightarrow f \\
\hline
f & \rightarrow b \rightarrow e \\
\hline
\end{array}
$$

Other graph types I

Weighted graphs

Graphs or digraphs with numbers assigned to the edges.

Paths

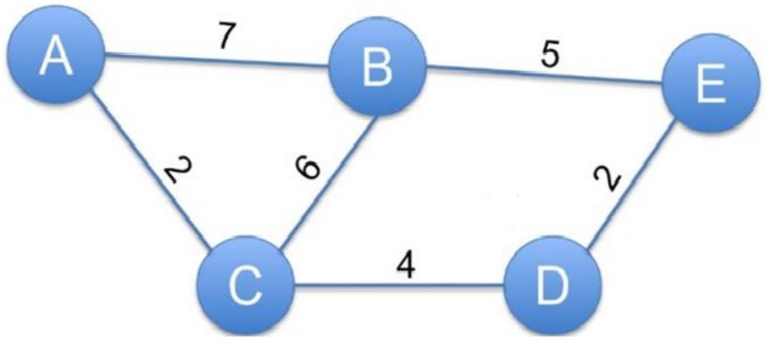
A path from vertex u to v of a graph G is defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.

Simple paths:

Each vertex is visited only once.

Path lengths:

the number of edges, or the number of vertices – 1.



Other graph types II

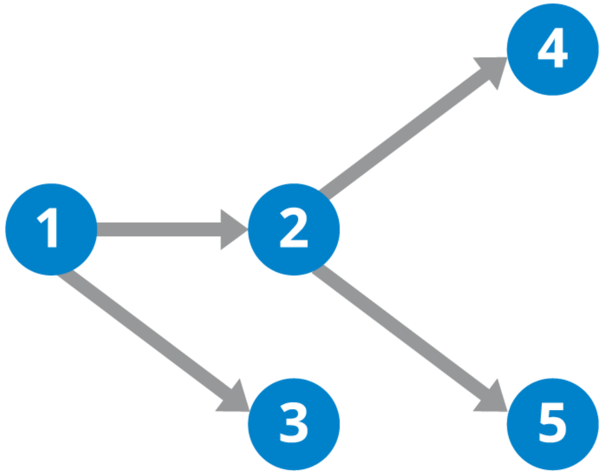
Cycle

A cycle is a path consisting of at least three vertices that starts and ends with the same vertex.

Acyclic graph

A graph without cycles

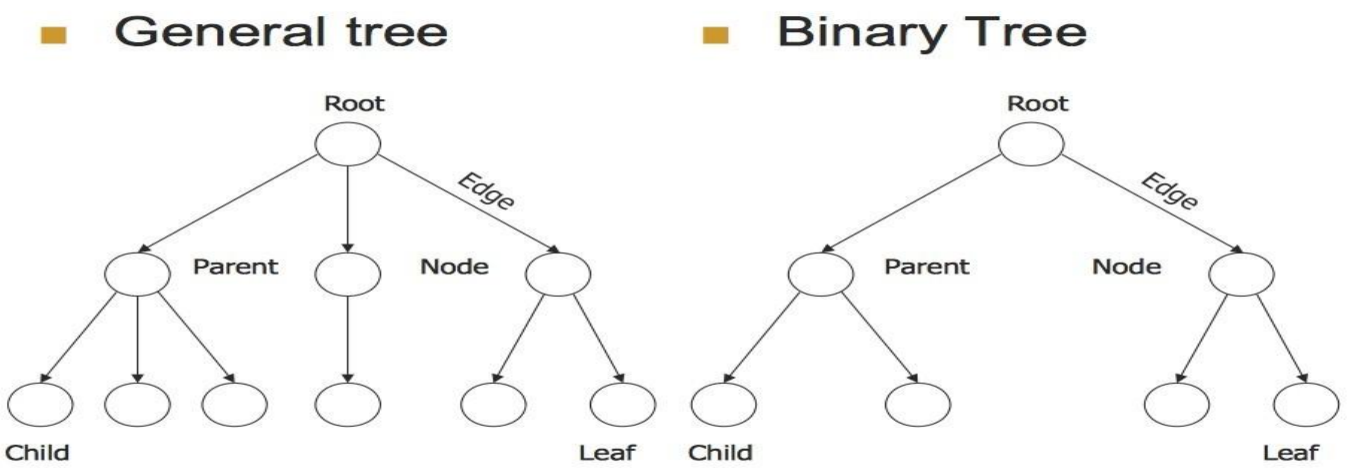
DAG (Directed Acyclic Graph)



Trees

Trees are natural structures for representing certain kinds of **hierarchical data.** (Eg: How our files get saved under hierarchical directories)

Tree is a data structure which allows you to associate a **parent-child relationship** between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion.



Properties of Trees I

Ancestors

For any vertex *v* in a tree *T*, all the vertices on the simple path from the root to that vertex are called ancestors.

Descendants

All the vertices for which a vertex *v* is an ancestor are said to be descendants of *v*.

Parent, child and siblings

If *(u, v)* is the last edge of the simple path from the root to vertex *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 called siblings.

Properties of Trees II

Leaves

A vertex without children is called a leaf.

Subtree

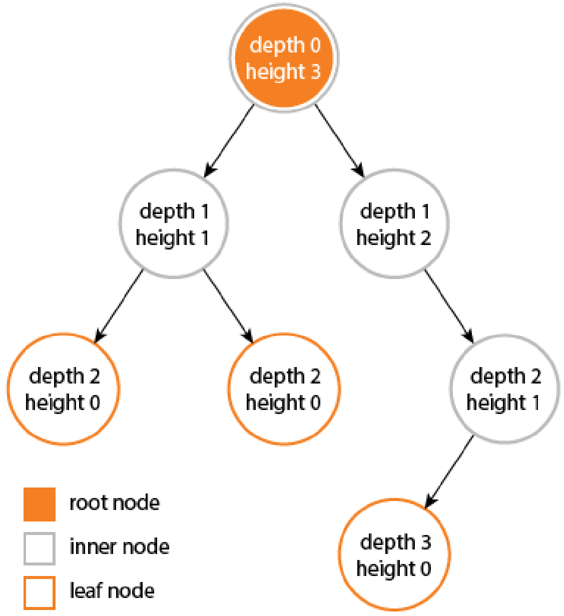
A vertex v with all its descendants is called the subtree of T rooted at v.

Depth of a vertex

The length of the simple path from the root to the vertex.

Height of a tree

The length of the longest simple path from the root to a leaf.



Types of Trees

Ordered trees

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

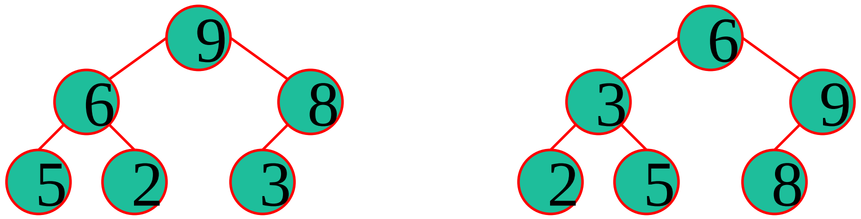
Binary trees

A binary tree is an ordered tree in which every vertex has no more than two children and each children is designated s either a left child or a right child of its parent.

Binary search trees

Each vertex is assigned a number.

A number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree.



Sets and Dictionaries I

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

- Plays important role in mathematics.
- Defined either by an explicit listing of its elements
  (e.g., $S = \{2, 3, 5, 7\}$)

  OR
- By specifying a property that all the set's elements and only they must satisfy
  (e.g., $S = \{n: n$ is a prime number smaller than $10\}$)

- Important set operations:
  - checking membership of a given item in a given set
  - finding the union of two sets
  - finding the intersection of two sets

Sets and Dictionaries II

Sets can be implemented in two ways:

1. Bit-vector representation

   Sets that are subsets of some large set *U*, called the universal set.

   - Eg: *U* = {1, 2, 3, 4, 5, 6, 7, 8, 9},
     then *S* = {2, 3, 5, 7} is represented by the bit string 011010100.
   - Limitation: requires large amount of storage.

2. Using the list structure to indicate the set's elements

   - name= { 'Arun' , 'Anjali', 'Akhil' }

Sets and Dictionaries III

Difference between set and list :

Set	List
Cannot contain identical elements	Can contain identical elements
Changing the order of elements does not change the set.	It is defined as an ordered collection of items and therefore do not accept the changes.

A set may be represented by a list, in a sorted order.

Sets and Dictionaries IV

Dictionaries: name_age = { 'Arun' : 20, 'Anjali': 10, 'Akhil' : 30 }

- A data structure which provides three important operations namely, search, add and delete.

- An efficient implementation of a dictionary has to strike a compromise between the efficiency of searching and the efficiencies of other two (add and delete) operations.

- Implementation: range from an unsophisticated use of arrays (sorted or not) to much more sophisticated techniques such as hashing and balanced search trees.

Analysis Framework I

- In what ways can we compare algorithms?
- Remember! from the previous class :

- How good is the algorithm?
    - Correctness
    - Time efficiency
    - Space efficiency

- Other Characteristics
    - Simplicity
    - Generality

Analysis Framework II

Analysis of algorithms means investigation of an algorithm's efficiency with respect running time and memory space.

Two kinds of efficiency :

1. Time efficiency – also called time complexity
   - indicates how fast an algorithm in question runs
2. Space efficiency – also called space complexity
   - the amount of memory units required by the algorithm in addition to the space needed for its input and output

Analysis Framework III

Efficiency considerations are of primary importance from a practical point of view when given the speed and memory of today's computers.

- Olden days: both time and space were important.
- Present days: Due to technological innovations computer speed and memory size have improved.
- But still time issue has not diminished.

Hence, primarily concentration given to time efficiency, but framework studied can be used for space efficiency also.

Measuring an Input's Size I

- All algorithms run longer on larger inputs

  Example: To sort larger arrays, multiply larger matrices, etc.

- Hence, an algorithm's efficiency may be decided as a function of some parameter ***n*** which indicates the algorithm's input size

Measuring an Input's Size II

- The choice of an appropriate size metric can be influenced by operations of the algorithm in question.

  - Example: how should we measure an input's size for a spell-checking algorithm?

    - If the algorithm examines individual characters of its input, we should measure the size by the number of characters.

    - If the algorithm works by processing words, we should count their number in the input.

Units for Measuring Running Time I

- Can we use standard units of time measurement? (Eg: a second, or millisecond)

Problems:

- Dependence on the speed of a particular computer.
- Dependence on the quality of a program implementing the algorithm and of the compiler used in generating the machine code.
- Difficulty of clocking the actual running time of the program.

Algorithm efficiency should not depend on these extraneous factors.

Units for Measuring Running Time II

- One possible approach is to count the number of times each of the algorithm’s operations is executed.
  - This approach is excessively difficult.
- The thing to do is to identify the most important operation of the algorithm, called the ***basic operation.***
  - ***Basic Operation:*** the operation contributing the most to the total running time.
- Compute the number of times the basic operation is executed.

Units for Measuring Running Time III

- Generally, the basic operation will be the most time-consuming operation in the algorithm's innermost loop.

Problem	Input size measure	Basic operation
Searching for key in a list of n items	Number of items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Units for Measuring Running Time Contd.

Thus, the established framework for the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size *n*.

Let **c_{op}** be the execution time of an algorithms basic operation on a particular computer, and let C(n) be the number of times this operation is run on input n, then we can estimate the running time T(n) of a program:

$$T(n) \approx c_{op}\, C(n)$$

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of **input size**

Basic operation: the operation that contributes the most towards the running time of the algorithm

input size

$$T(n) \approx c_{op}C(n)$$

running time

execution time
for basic operation
or cost

Number of times basic
operation is executed

Note: Different basic operations may cost differently!

Orders of Growth

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Thank you!

Any queries?