Data Structures and Algorithm

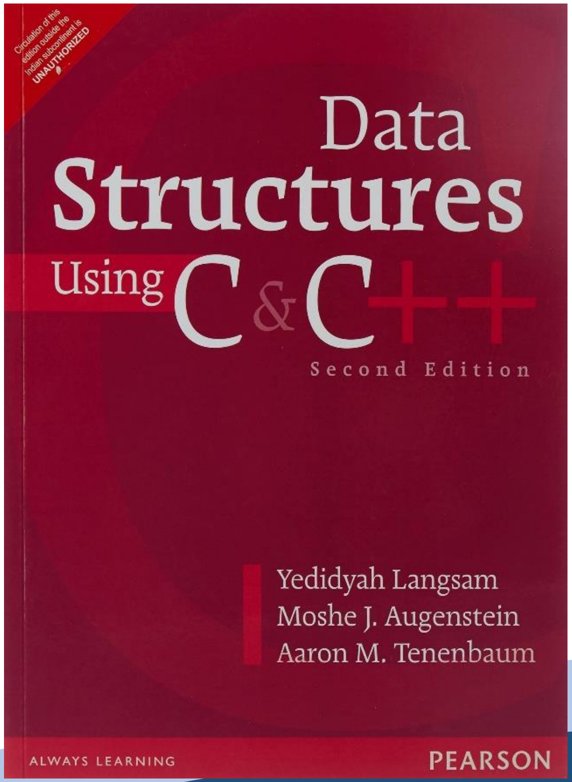
Year/Part: II/I
Faculty :BCA

Theory: 60+20
Practical: 20

References:
A. M Tenenbaum “Data Structures using C and C++

Course Distribution

Unit	Hour	Chapters
1	2	Introduction
2	3	Stack
3	3	Queue
4	3	List
5	5	Linked List
6	4	Recursion
7	5	Trees
8	5	Sorting
9	5	Searching
10	5	Graphs
11	5	Algorithms
Total	45	


Circulation of this edition outside the Indian subcontinent is UNAUTHORIZED
Data Structures Using C & C++
Second Edition
Yedidyah Langsam
Moshe J. Augenstein
Aaron M. Tenenbaum
ALWAYS LEARNING
PEARSON

UNIT- ONE
Introduction to Data Structures and Algorithm

Data and Data item:

- Data are values or set of values
- Data are simply collection of facts and figures
- Data items refers to a single unit of values. (eg. A set of characters which are used together to represent a specific data element is represented by the data item)
  - Element data item: can’t be further sub divided(eg. studentID)
  - Group Data item: can be further sub divided. (eg. DOB can be divided into YY, MM , DD)
- Records: collection of related data items(eg. Student record contains data fields as SID, Name, Roll, PhNo etc.)
- File: collection of logically related records.

Data Structure Definition:

- Structure means particular way of data organization.
- Data structure refers to the organization of data in computer memory or the way in which data is efficiently stored, processed and retrieved.
- Data structure is the structural representation of logical relationship between elements of data.
- Data structure is a group of elements grouped together under one name. These data elements known as members and can have different types and different length.
- Data structure is a logical model of a particular organization of data.
  - It must be able to represent the inherent relationship of the data in real world
  - It must be simple enough so that it can process efficiently as when necessary.

Need of a Data Structure

The study of data structures includes:

- Logical description of data structures
- Implementation of data structure
- Quantitative analysis of data structure

Area of Data Structures extensively applied

- Database Management System
- Compiler Design
- Network analysis
- Numerical Analysis
- Artificial Intelligence
- Simulation
- Operating System
- Graphics

Characteristic of various Data structure

Data Structure	Advantages	Disadvantages
Array	Quick insertion, fast access if index is known	Slow search, slow detection, fixed size
Ordered Array	Quick search	Slow insertion and deletion, fixed size
Stack	Last in first out	Slow access to other item
Queue	First in first out	Slow access to other item
Linked List	Quick insertion and deletion	Slow search
Binary Tree	Quick search , insertion and deletion if tree is in balanced condition	Deletion algorithm is complex
Hash table	Fast access, fast insertion	Slow deletion
Heap	Fast insertion deletion	Slow access to other thing
Graph	Models real world situation	Some of the algorithm are slow in access

Types of data Structures

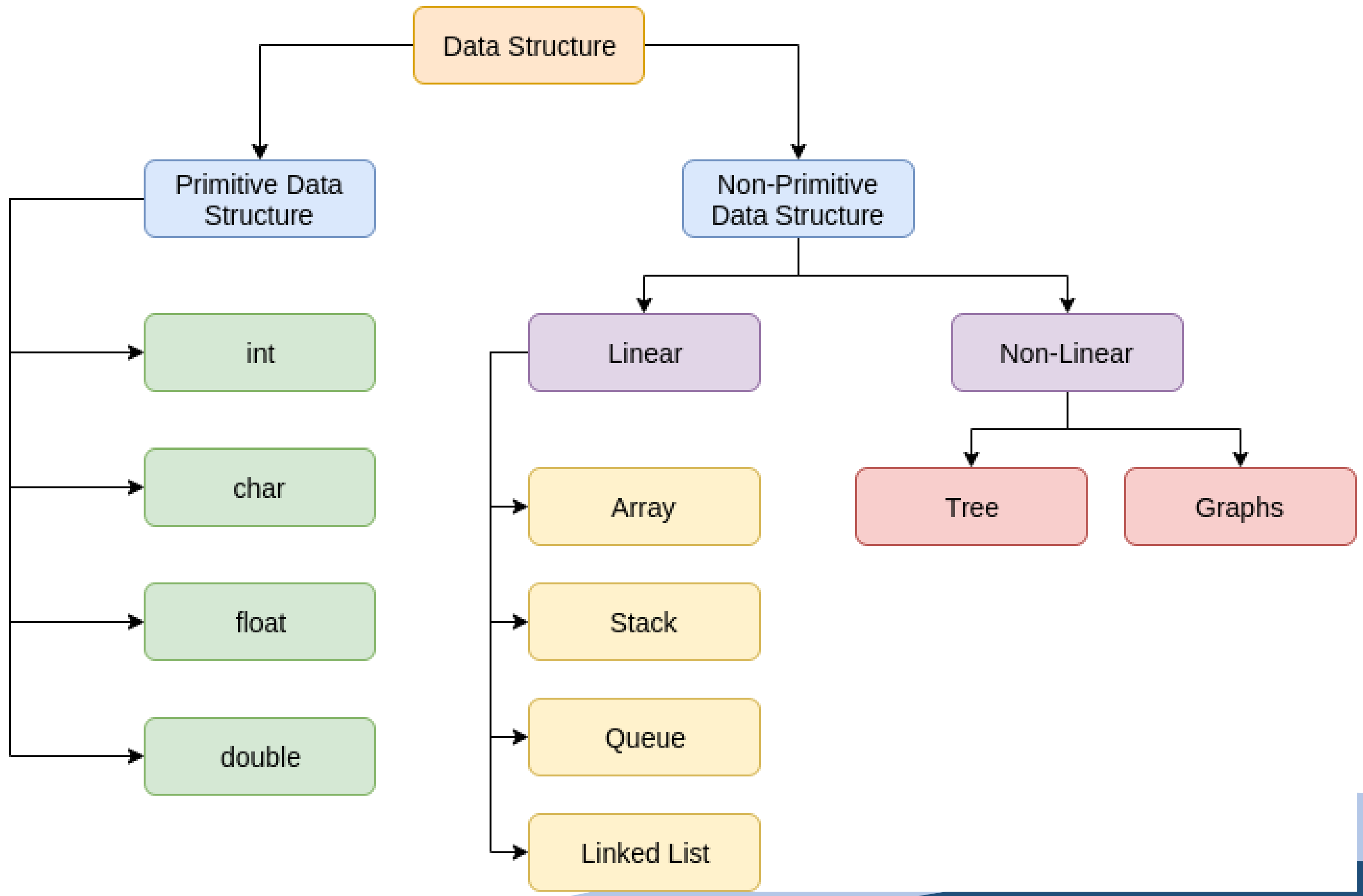
- According to Nature of Size:
  - Static Data Structure (eg. Array)
  - Dynamic Data Structure (eg. Link list, tree, graph)

- According to its occurance
  - Linear data Structure: data is stored in consecutive memory location (eg. Array, link list, queue, stack etc)
  - Non linear data Structure: data is stored in non consecutive memory location. (eg. Tree, graphs etc)
  -

Primitive and Non- Primitive data Structure

- Primitive data structure: are basic data structures and are directly operated upon by the machine instructions. (eg. Integers, floating point numbers, characters string constants and pointers etc)
- Non primitive data Structure: group of homogenous or heterogeneous data items
  - Are derived from primitive data structures (eg. Array, list, files, trees , graphs etc)

Types of data Structures



Data Types Vs Data Structures

- Data type of a variable is the set of values that the variable may assume. Eg Boolean, integer, float etc
  - Abstract data type: mathematical model together with various operations defined on model.
- A data structure is an actual implementation of a particular abstract data type. We may also call data structure as an extension of the concept of the data types.

Data Type: Permitted values + Operations

Data Structure= Original data + allowed Operation

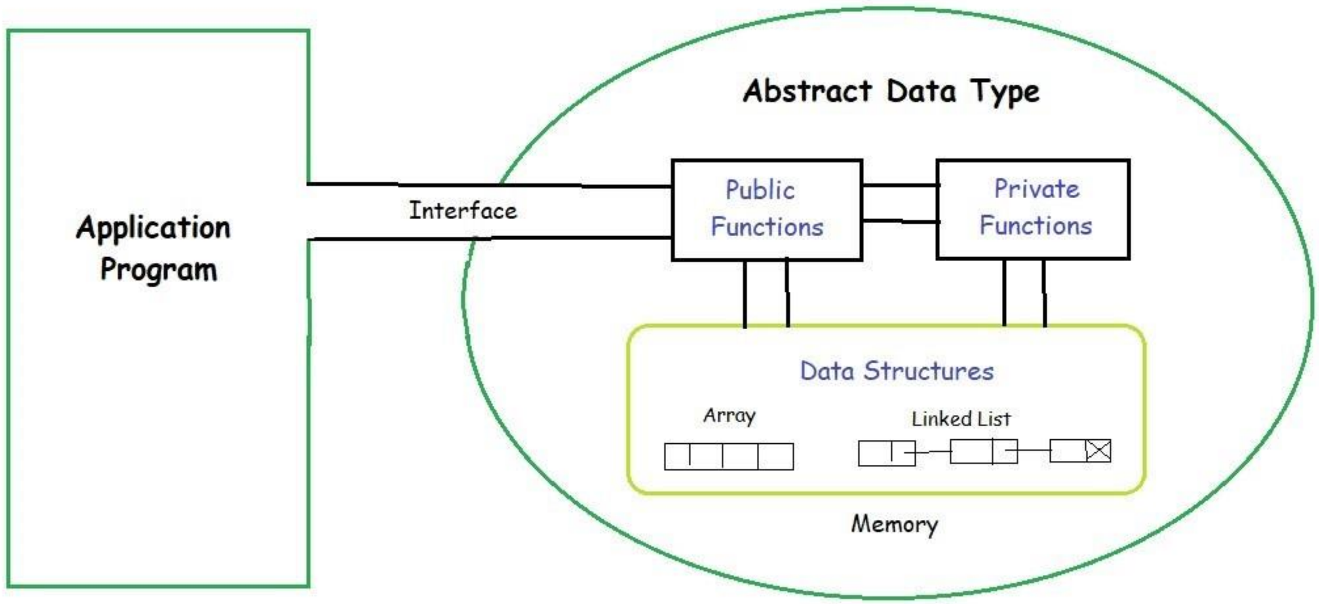
Data Structure Operations

- Creating: initialize and reserve memory location for the data elements
- Inserting: adding new records to the structure.
- Deleting: removing a record from the structure
- Updating: change data values of the data structure
- Traversing: accessing each records
- Searching: finding the location of the record
- Sorting: Arranging the data elements in some logical order
- Merging: Combine the data elements in two different sorted setsinto a single sorted set
- Destroying: last operation of data structure and are applied when no longer need of the data structure.

Abstract Data Types

- An Abstract Data Type (ADT) is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.
- Sets of integers, together with the operation of union, intersection and set differences form a simple example of an ADT.
- Useful tool for specifying the logical properties of a data type.
- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view.

Example: Abstraction- implementation independent view



- The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.
- So a user only needs to know what a data type can do, but not how it will be implemented. We can think of ADT as a black box which hides the inner structure and design of the data type. Example:
- The **List ADT Functions** is given below:
  - get() – Return an element from the list at any given position.
  - insert() – Insert an element at any position of the list.
  - remove() – Remove the first occurrence of any element from a non-empty list.
  - removeAt() – Remove the element at a specified location from a non-empty list.
  - replace() – Replace an element at any position by another element.
  - size() – Return the number of elements in the list.
  - isEmpty() – Return true if the list is empty, otherwise return false.
  - isFull() – Return true if the list is full, otherwise return false.

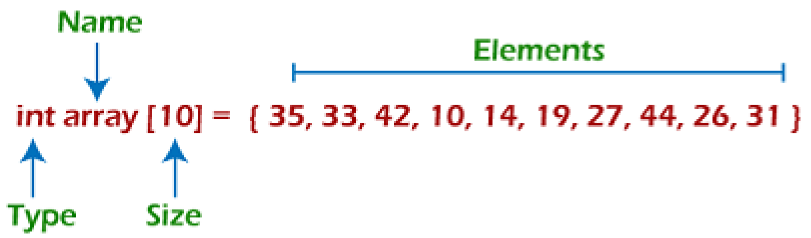
Features of ADT:

- **Abstraction:** The user does not need to know the implementation of the data structure.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.

Various Data Structures:

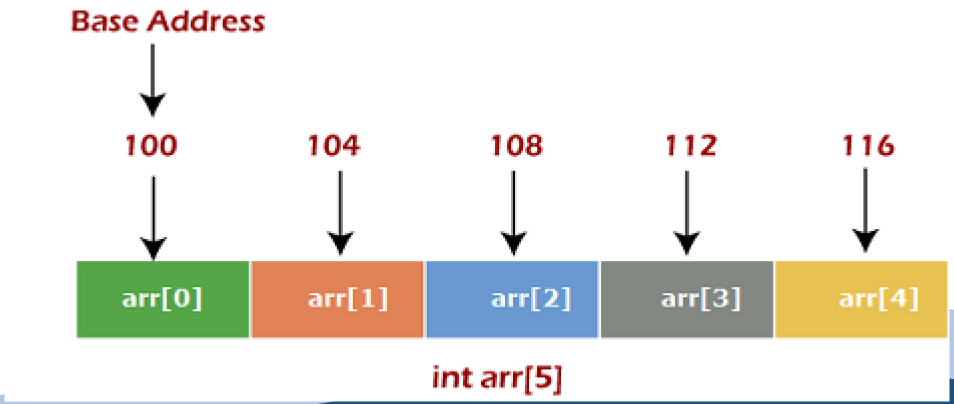
❖Array: Array is defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

- Example: Array representation



- Memory Allocation in Array:

int arr[5];



- Advantages of Array
  - Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
  - Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
  - Any element in the array can be directly accessed by using the index.
- Disadvantages of Array
  - Array is homogenous. It means that the elements with similar data type can be stored in it.
  - In array, there is static memory allocation that is size of an array cannot be altered.
  - There will be wastage of memory if we store less number of elements than the declared size.

❖Stack: A stack is an ordered list in which items are inserted and removed at only one end called TOP. Only two operations are possible on stack i.e, PUSH and POP.

- An example of stack is pile of CD disks
- Stack uses Last In First Out list.

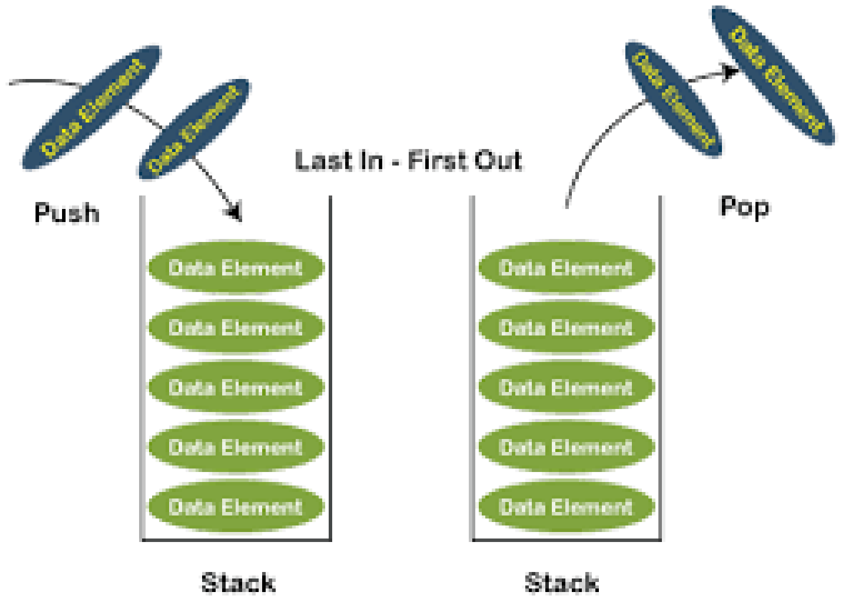


Fig: Graphical representation of stack

❖Queue: A queue is an ordered list in which all insertion can take place at one end called REAR and all deletion take place at the other end called FRONT.

- Queue is also called First In First Out list.
- People standing in a queue at bank cash counter.

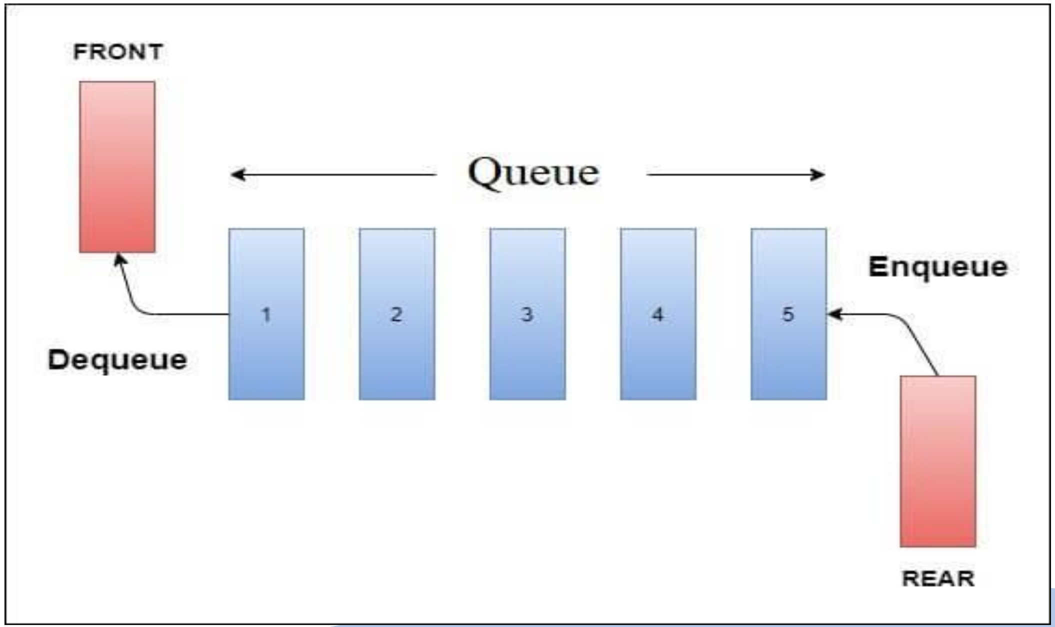
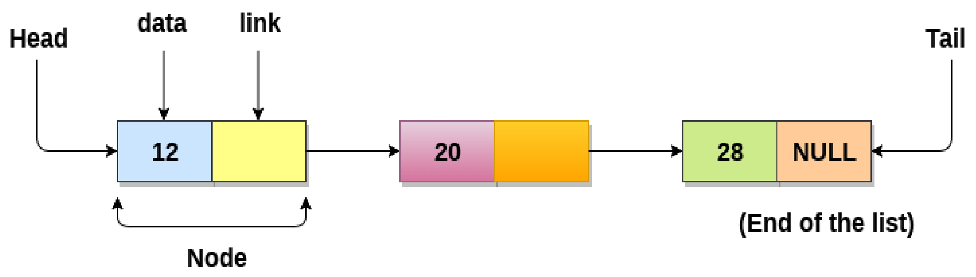


Fig: Graphical representation of queue

❖Linked list: Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.

- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.

Fig: Graphical representation of Linked list



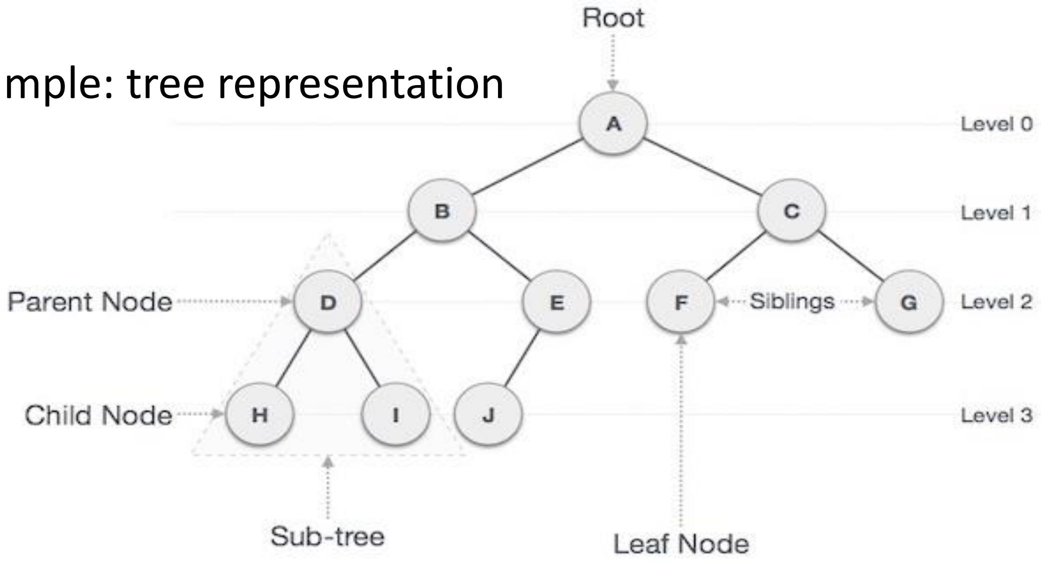
Uses of Linked List

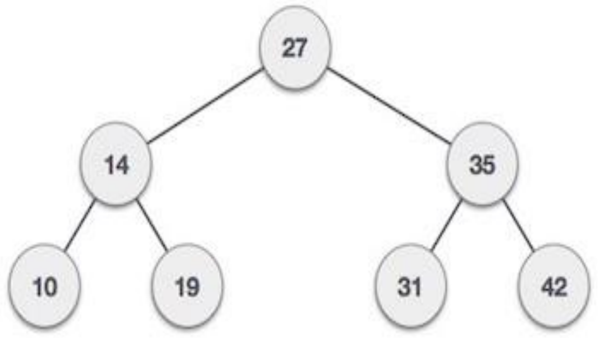
- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.

❖Tree: ***A tree*** is also one of the data structures that represent hierarchical data.

- Tree represents the nodes connected by edges.
  - Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

Example: tree representation



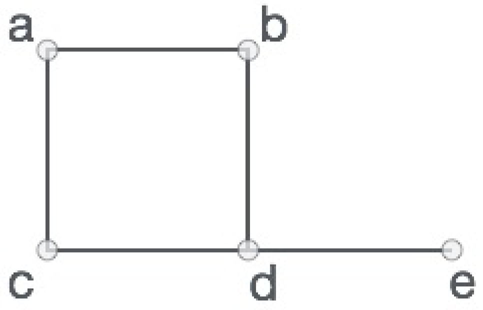


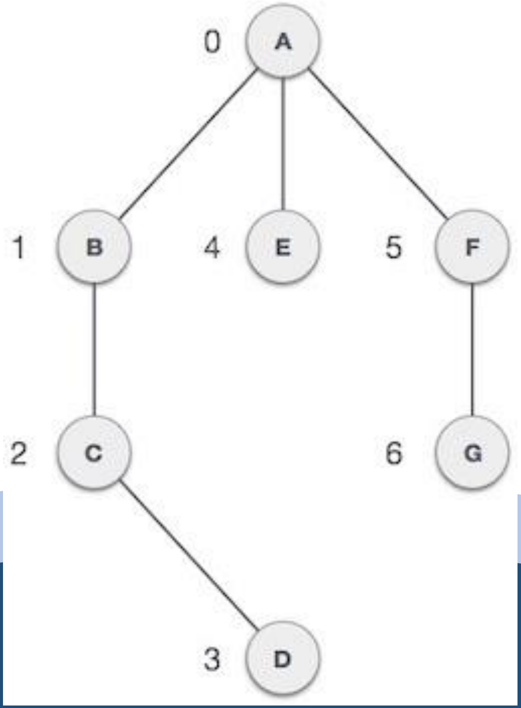
Example: binary tree

❖Graph: A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

- Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Example:

In the figure representing graph,
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}





Algorithm:

- An algorithm is a precise plan for performing a sequence of actions to achieve the intended purpose.

Algorithm Design Techniques:

- Top- down algorithm design approach: states that a program should be divided into a main module and its related sub modules.
- Each module/ specification is broken down into simpler pieces
- Example: a c program with main() functions and other user defined functions.
- Bottom-Up algorithm design approach: opposite to top- down design.
- Start the design with specific module and building them into more complex structure.

Assignment 1.

- List out and describe different algorithm design approaches.

Analysis of an algorithm

- Algorithm analysis measures the efficiency of the algorithm. It can be checked by:
  - Correctness of an algorithm
  - Implementation of an algorithm
  - Simplicity of an algorithm
  - Execution time and memory requirements of an algorithm

❖Types of analysis:
  - Worst case running time: It defines the input for which the algorithm takes a huge time.
  - Average case running time: It defines the input for which it takes average time for the program execution.
  - Best case running time: It defines the input for which the algorithm takes the lowest time

Factors determining Efficiency of Algorithm

- Space Complexity
- Time Complexity

Space Complexity

- An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.
- For an algorithm, the space is required for the following purposes:
  - to store program instructions
  - To store constant values
  - To store variable values
  - To track the function calls, jumping statements, etc.
- Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.
  - **Space complexity = Auxiliary space + Input size.**

Time Complexity

- **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation.
- Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

```
sum=0;
for i=1 to n
{
 sum= sum+i;
}
return sum;
```

In this code, the time complexity of the loop statement will be atleast n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only.

Time- Space tradeoff

- The best algorithm or best program to solve a given problem is one that requires less space in memory and takes less time to execute its instruction or to generate output. But in practice, it is not always possible to achieve both of these objectives.
- The time- space tradeoff refers to a choice between algorithmic solutions of a data processing problem that allows one to decrease the running time of an algorithmic solution by increasing the space to store the data and vice versa.
- An algorithm A for a problem P is said to have time complexity of T(n) if the number of steps required to complete its run for an input of size n is always less than or equal to T(n)
- An algorithm A for a problem P is said to have space complexity of S(n) if the number of bits required to complete its run for an input of size n is always less than or equal to S(n)

Asymptotic Analysis

- Asymptotic analysis of algorithm is a method of defining the mathematical bound of its run-time performance.
- Using the asymptotic analysis, we can easily conclude the average-case, best-case and worst-case scenario of an algorithm.

Example: Let us consider f(n) denote the time complexity. And let,

$f(n) = 5n^2 + 6n + 12$

It is observed that for larger values of n, the squared term consumes almost 99% of the time. As the n^2 term is contributing most of the time, so we can eliminate the rest two terms.
Therefore time complexity $f(n) = 5n^2$

	% of running time due to		
n	$5n^2$	6n	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%

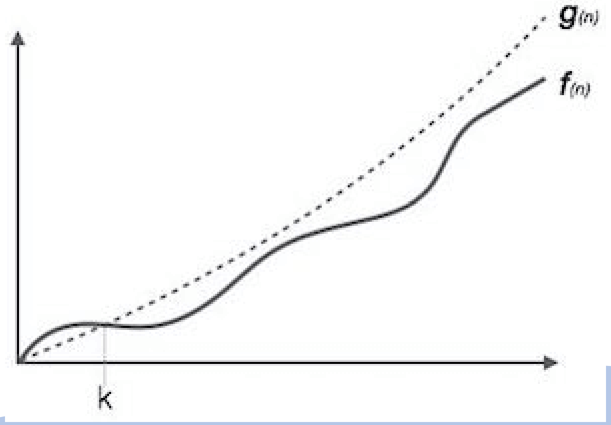
Asymptotic Notation

- Big oh Notation (O)
- Omega Notation (Ω)
- Theta Notation (θ)

Big oh Notation (O)

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.
- It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation.

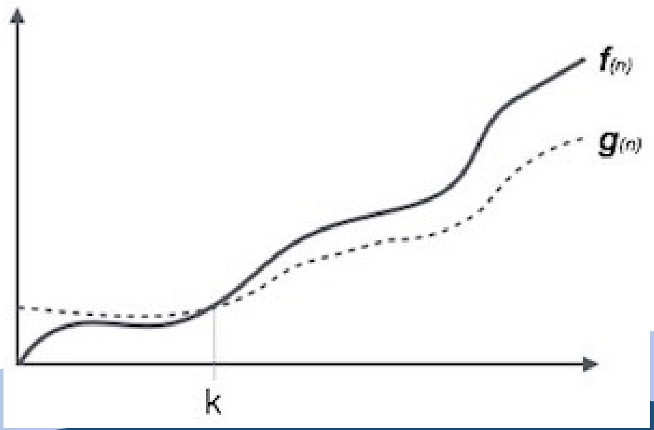
For example:
If **f(n)** and **g(n)** are the two functions defined for positive integers,
then **f(n)** = **O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:
f(n)≤c.g(n) for all n≥no



Omega Notation (Ω)

- It basically describes the best-case scenario which is opposite to the big oh notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.
- It is used to bound the growth of running time for large input size.

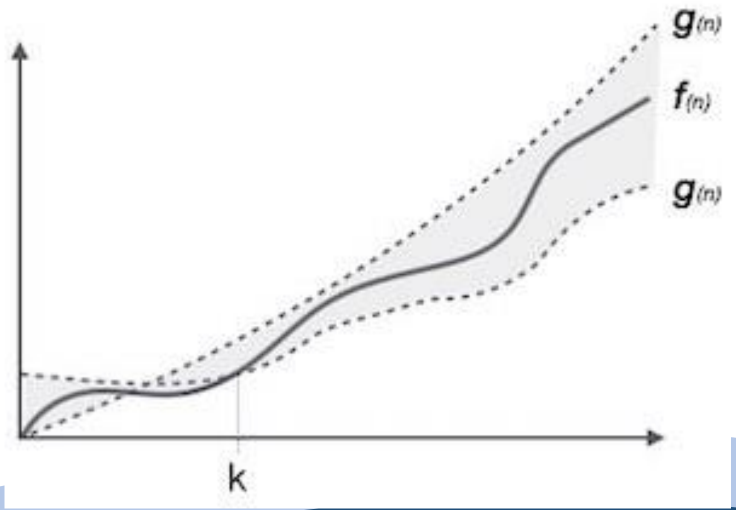
If **f(n)** and **g(n)** are the two functions defined for positive integers,
then **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:
f(n)>=c.g(n) for all n≥no and c>0



Theta Notation (θ)

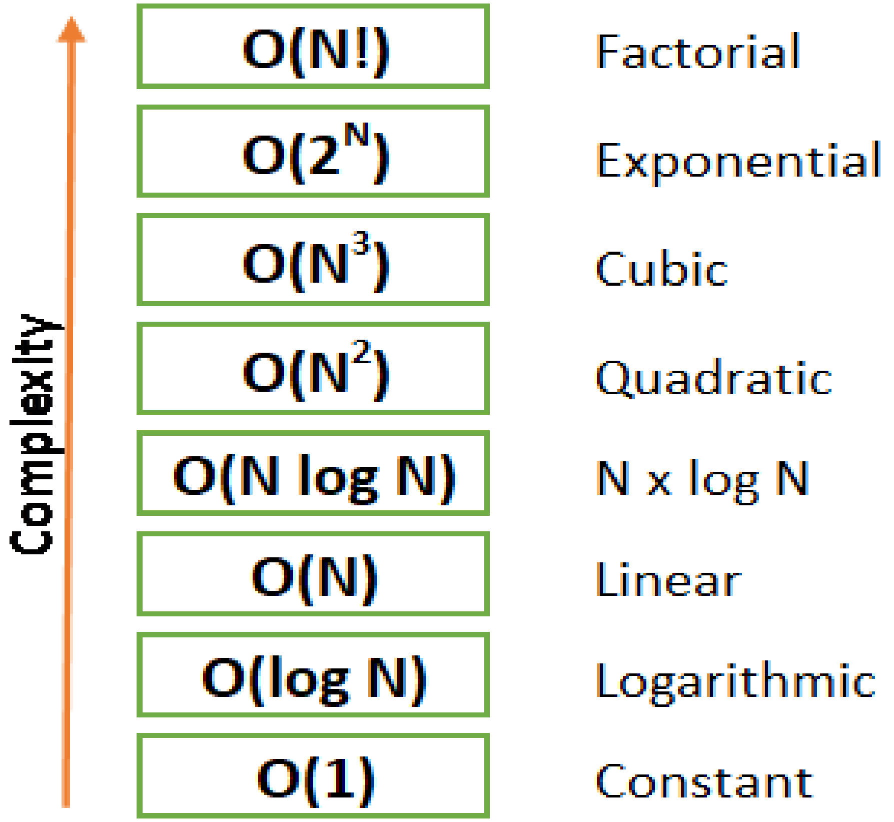
- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:
f(n)= θg(n)
The above condition is satisfied only if when
c1.g(n)<=f(n)<=c2.g(n)



Common Asymptotic Notations

- List out and describe different algorithm design approaches.



Any Queries?