Data Structure

IBPS SO (IT- Officer) Exam 2017

Data Structure: In computer science, a data structure is a way of storing and organizing data in a computer's memory so that it can be used efficiently. Data may be organized in many different ways. The logical or mathematical model of a particular organization of data is called a data structure. The choice of a particular data model depends on the two considerations first; it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data whenever necessary.

Need of data structure-:

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

Selecting a data structure-: Selection of suitable data structure involve following steps –

- Analyze the problem to determine the resource constraints a solution must meet.
- Determine basic operation that must be supported. Quantify resource constraint for each operation
- Select the data structure that best meets these requirements.
- Each data structure has cost and benefits. Rarely is one data structure better than other in all situations.

Type of data structure-:

1. **Static data structure:** A data structure whose organizational characteristics are invariant throughout its lifetime. Such structures are well supported by high-level languages and familiar examples are arrays and records. The prime features of static structures are-
   - None of the structural information need be stored explicitly within the elements – it is often held in a distinct logical/physical header;
   - The elements of an allocated structure are physically contiguous, held in a single segment of memory.
   - All descriptive information, other than the physical location of the allocated structure, is determined by the structure definition.
   - Relationships between elements do not change during the lifetime of the structure.

2. Dynamic data structure-: A data structure whose organizational characteristics may change during its lifetime. The adaptability afforded by such structures, e.g. linked lists, is often at the expense of decreased efficiency in accessing elements of the structure. Two main features distinguish dynamic structures from static data structures. Firstly, it is no longer possible to infer all structural information from a header; each data element will have to contain information relating it logically to other elements of the structure. Secondly, using

a single block of contiguous storage is often not appropriate, and hence it is necessary to provide some storage management scheme at run-time.

Linear Data Structure-:

A data structure is said to be linear if its elements from any sequence. There are basically two ways of representing such linear structure in memory.

a) One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays**.**

b) The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists**.** The common examples of linear data structure are arrays, queues, stacks and linked lists.

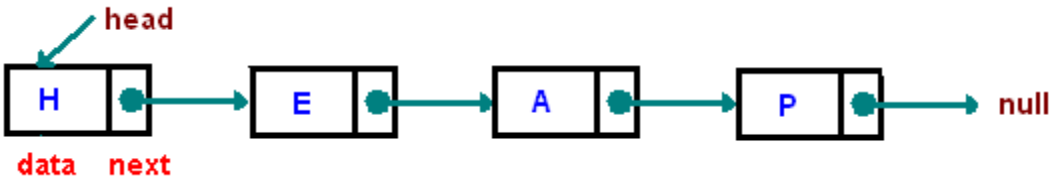
Non-linear Data Structure-:

This structure is mainly used to represent data containing a hierarchical relationship between elements. E.g. graphs, family trees and table of contents.

Array-:

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number *n* of similar data referenced respectively by a set of *n* consecutive numbers, usually 1, 2, 3 . . . . . *n*. if we choose the name **A** for the array, then the elements of **A** are denoted by subscript notation.

Linked List-: One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions. In this chapter we consider another data structure called Linked Lists that addresses some of the limitations of arrays. A linked list is a linear data structure where each element is a separate object.



Each element (we will call it a **node**) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

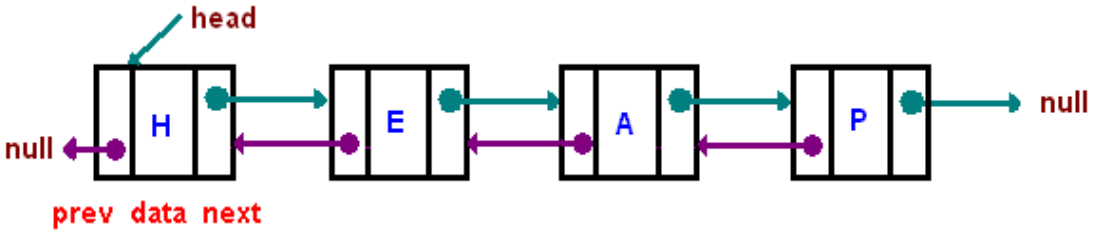
A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a reference to the next node.

Types of Linked Lists

1. A **singly linked list** is described above.
2. A **doubly linked list** is a list that has two references, one to the next node and another to previous node.



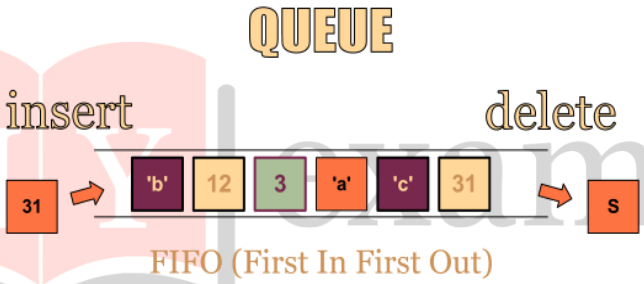
Another important type of a linked list is called a **circular linked list** where last node of the list points back to the first node (or the head) of the list.

Abstract Data Type-:

It can be defined as a collection of data items together with the operations on the data. The word "abstract" refers to the fact that the data and the basic operations defined on it are being studied independently of how they are implemented. It involves **what** can be done with the data, not how has to be done. For ex, in the below figure the user would be involved in checking that what can be done with the data collected not how it has to be done.

Queue- :

A queue is a linear list of elements in which deletion can take place only at one end, called the ***front***, and insertions can take place only at the other end, called the ***rear***. The term "front" and "rear" are used in describing a linear list only when it is implemented as a queue. Queue is also called **first-in-first-out (FIFO)** lists. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enters a queue is the order in which they leave.



There are main two ways to implement a queue:

1. Circular queue using array
2. Linked Structures (Pointers)

Primary queue operations:

Enqueue : insert an element at the rear of the queue.

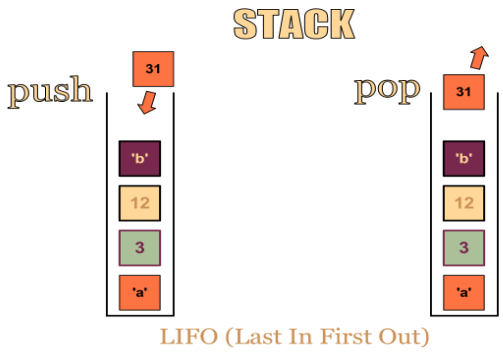
Dequeue : remove an element from the front of the queue Following is the algorithm which describes the implementation of Queue using an Array.

Priority queue-:

Priority queue is a linear data structure. It is having a list of items in which each item has associated priority. It works on a principle add an element to the queue with an associated priority and remove the element from the queue that has the highest priority. In general different items may have different priorities. In this queue highest or the lowest priority item are inserted in random order. It is possible to delete an element from a priority queue in order of their priorities starting with the highest priority. While priority queues are often implemented with heaps, they are conceptually distinct from heaps. A priority queue is an abstract concept like **"a list"** or **"a map"**; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods such as an unordered array.

Stack-:

It is an ordered group of homogeneous items of elements. Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (LIFO: Last In, First Out).



Stack Operations-:

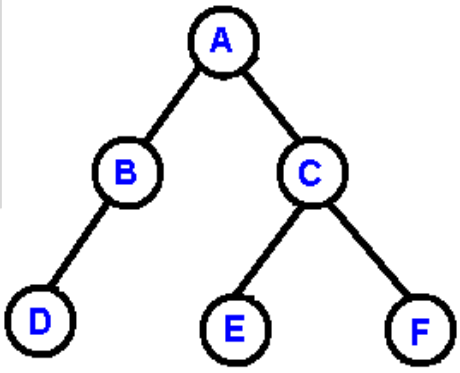
These are two basic operations associated with stack:

1. ***Push()*** is the term used to insert/add an element into a stack.
2. ***Pop()*** is the term used to delete/remove an element from a stack.

Tree - :

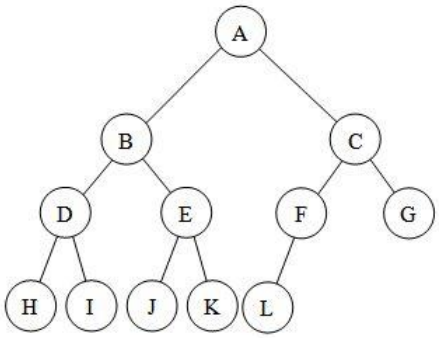
A node is a structure which may contain a value, a condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more **child nodes,** which are below it in the tree (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's **parent node** (or ancestor node, or superior). A node has at most one parent. Nodes that do not have any children are called **leaf nodes**. They are also referred to as terminal nodes. The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The **depth** of a node is the length of the path to its root (i.e., its root path).

Binary Tree -:



The binary tree is a fundamental data structure used in computer science. The binary tree is a useful data structure for rapidly storing sorted data and rapidly retrieving stored data. A binary tree is composed of parent nodes, or leaves, each of which stores data and also links to up to two other child nodes (leaves) which can be visualized spatially as below the first node with one placed to the left and with one placed to the right. It is the relationship between the leaves linked to and the linking leaf, also known as the **parent node**, which makes the binary tree such an efficient data structure. It is the leaf on the left which has a lesser key value (i.e, the value used to search for a leaf in the tree), and it is the leaf on the right which has an equal or greater key value. As a result, the leaves on the farthest left of the tree have the lowest values, whereas the leaves on the right of the tree have the greatest values. More importantly, as each leaf connects to two other leaves, it is the beginning of a new, smaller, binary tree. Due to this nature, it is possible to easily access and insert data in a binary tree using search and insert functions recursively called on **successive leaves**.

Complete Binary Tree -:

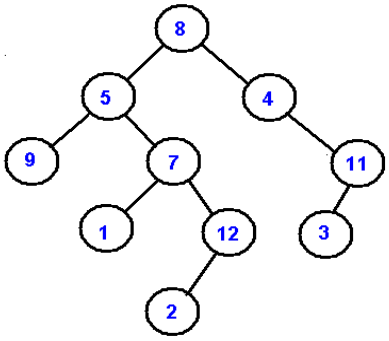


A binary tree T with n levels is complete if all levels except possibly the last are completely full, and the last level has all its nodes to the left side. A complete binary tree has 2k nodes at every depth k < n and between 2n and 2n+1-1 nodes altogether. It can be efficiently implemented as an array, where a node at index i has children at indexes 2i and 2i+1 and a parent at index i/2, with one-based indexing. If child index is greater than the number of nodes, the child does not exist. A complete binary tree can be represented in an array in the following approach. If root node is stored at index i, it's left, and right children are stored at indices 2*i+1, 2*i+2 respectively.

Traversal:-

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal.

There are three different types traversals,

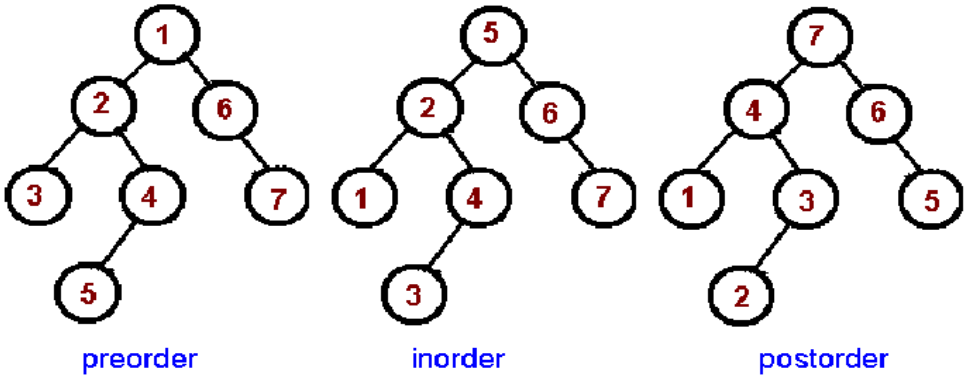


- **PreOrder traversal** - visit the parent first and then left and right children.
- **InOrder traversal** - visit the left child, then the parent and the right child.
- **PostOrder traversal** - visit left child, then the right child and then the parent. There is only one kind of breadth-first traversal--the level order traversal.

This traversal visits nodes by levels from top to bottom and from left to right. As an example consider the following tree and its four traversals:

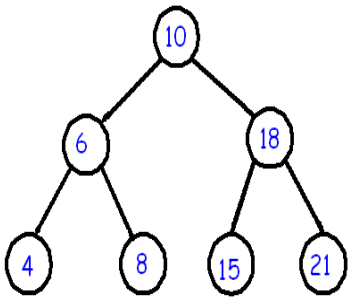
PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3,11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11,4, 8

In the next picture we demonstrate the order of node visitation. Number 1 denotes the first node in a particular traversal and 7 denote the last node.



Binary Search Tree-:

We consider a particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retrieving. A BST is a binary tree where nodes are ordered in the following way:



- Each node contains one key (also known as data)
- The keys in the left subtree are less than the key in its parent node, in short L < P.
- The keys in the right subtree are greater the key in its parent node, in short P < R.
- Duplicate keys are not allowed.

In the following tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10. Because both the left and right subtrees of a BST are again search trees; the above definition is recursively applied to all internal nodes:

Heaps-:

A **heap** is a binary tree where the elements are arranged in a certain order proceeding from smaller to larger. In this way, a heap is similar to a binary search tree (discussed previously), But the arrangement of the elements in a heap follows rules that are different from a binary search tree:

1. In a heap, the element contained by each node is greater than or equal to the elements of that node's children.
2. The tree is a complete binary tree, so that every level except the deepest must contain as many nodes as possible and at the deepest level, all the nodes are as far left as possible.

Linear Search-:

Linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Binary search-:

A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

Graph-:

Data sometimes contains a relationship between pairs of elements which is not necessarily hierarchical in nature, e.g. an airline flights only between the cities connected by lines. This data structure is called Graph.

Graph representation: a graph is a collection of vertices (*or nodes)*, pairs of which are joined by edges (*or lines).*

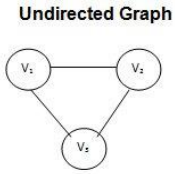


Figure 1: An Undirected Graph

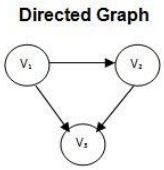


Figure 2: A Directed Graph

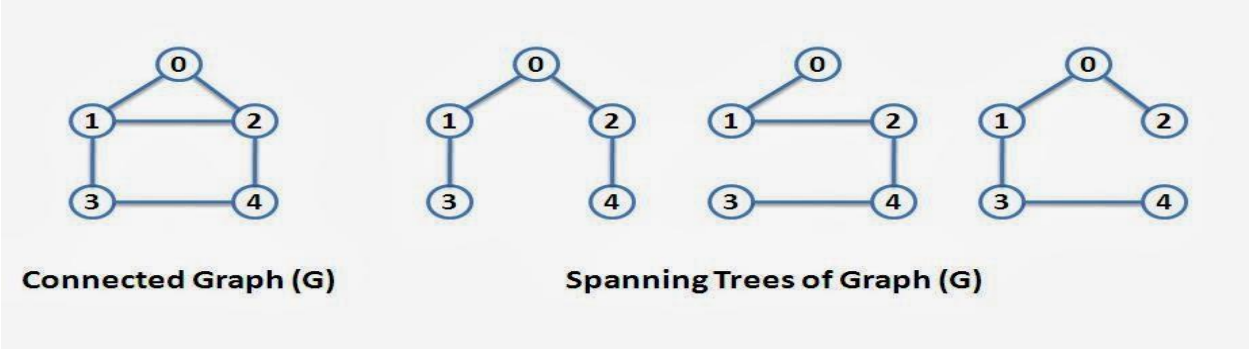
Graph operations and representation-:

1. **Path problems**: Since a graph may have more than one path between two vertices, we may be interested in finding a path with a particular property. For example, find a path with the minimum length from the root to a given vertex (node).
2. **Simple path:** a path in which all vertices, except possibly the first and last, are different.
3. **Undirected graph:** a graph whose vertices do not specify a specific direction.
4. **Directed graph:** a graph whose vertices do specify a specific direction.
5. **Connected graph:** there is at least one path between **every** pair of vertices.
6. **Bipartite graphs:** graphs that have vertexes that are partitioned into 2 subsets A and B, where every edge has one endpoint in subset A and the other endpoint in subset B.
7. **A complete graph:** an n-vertex undirected graph with n(n-1)/2 edges is a complete graph.
8. **A complete digraph:** (denoted as Kn) for n-vertices a complete digraph contains exactly n(n-1) directed edges
9. **Incident:** the edge (i, j) is incident on the vertices i and j (there is a path between i and j)
10. **In-degree:** the in-degree d of vertex i is the # of edges incident to i (the # of edges coming into this vertex)
11. **The out-degree:** the out-degree d of vertex i is the # of edges incident from vertex i (the # of edges leaving vertex i)
12. **The degree of a vertex**: the degree d of vertex i of an undirected graph is the number of edges incident on vertex i
13. **Connected component**: a maximal sub-graph that is connected, but you cannot add vertices and edges from the original graph and retain connectedness. A connected graph has EXACTLY one component
14. **Communication network**: Each edge is a feasible link that can be constructed. Find the components and create a small number of feasible links so that the resulting network is connected

15. **Cycles:** the removal of an edge that is on a cycle does not affect the networks connectedness (a cycle creates a sort of loop between certain vertices, for example there is a path that links vertex a to b to c and then back to a)

Spanning problems:

A spanning tree: is a sub-graph that includes all vertices of the original graph without cycles. Start a breadth-first search at any vertex of the graph. If the graph is connected, the n-1 edges are used to get to the unvisited vertices define the spanning tree (breadth-first Spanning tree)



Graph search methods:

A vertex *u* is reachable from vertex *b* iff there is a path from *u* to *b* A search method starts at a given vertex *v* and visits/labels/marks every vertex that is reachable from *v* Many graph problems are solved using search methods.

1. **Breadth-first search**:

- Visit the start vertex and use a FIFO queue
- Repeatedly remove a vertex from the queue, visit its *unvisited* adjacent vertices putting the newly visited vertices into the queue, when the queue is empty the search terminates
- All vertices that are reachable from the start vertex (including the start vertex) are visited

2. **Depth-first search**:

- Has the same complexity as breadth-first search
- Has the same properties with respect to path finding, connected components, and spanning trees
- Edges used to reach unvisited vertices define a depth-first spanning tree when the graph is connected

RECURSION-:

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it *is* similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call. One simple example is the idea of building a wall that is ten feet high; if I want to build a ten foot high wall, and then I will first build a 9 foot high wall, and then add an extra foot of bricks. Conceptually, this is like saying the "build wall" function takes a height and if that height is greater than one, first calls itself to build a lower wall, and then adds one a foot of bricks.