**Data Structures**

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Below is an overview of some popular linear data structures

1. Array

2. Linked List

3. Stack

4. Queue

**Array**

Array is a data structure used to store homogeneous elements at contiguous locations. Size of an array must be provided before storing data.

Let size of array be n.

Accessing Time: O(1) [This is possible because elements

are stored at contiguous locations]

Search Time: O(n) for Sequential Search:

O(log n) for Binary Search [If Array is sorted]

Insertion Time: O(n) [The worst case occurs when insertion

happens at the Beginning of an array and

requires shifting all of the elements]

Deletion Time: O(n) [The worst case occurs when deletion

happens at the Beginning of an array and

requires shifting all of the elements

**Linked List**

A linked list is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprising of two items – the data and a reference to the next node.

Types of Linked List :

1. Singly Linked List : In this type of linked list, every node stores address or reference of next node in list and the last node has next address or reference as NULL. For example 1->2->3->4->NULL

2. Doubly Linked List : In this type of Linked list, there are two references associated with each node, One of the reference points to the next node and one to the previous node. Advantage of this data structure is that we can traverse in both the directions and for deletion we don’t need to have explicit access to previous node. Eg. NULL<-1<->2<->3->NULL

3. Circular Linked List : Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. Advantage of this data structure is that any node can be made as starting node.

Accessing time of an element : O(n)

Search time of an element : O(n)

Insertion of an Element : O(1) [If we are at the position

where we have to insert

an element]

Deletion of an Element : O(1) [If we know address of node

previous the node to be

deleted]

**Stack**

A stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added. In stack both the operations of push and pop takes place at the same end that is top of the stack. It can be implemented by using both array and linked list.

Insertion : O(1)

Deletion : O(1)

Access Time : O(n) [Worst Case]

Insertion and Deletion are allowed on one end.

Eg  Stacks are used for maintaining function calls (the last called function must finish execution first), we can always remove recursion with the help of stacks. Stacks are also used in cases where we have to reverse a word, check for balanced parenthesis and in editors where the word you typed the last is the first to be removed when you use undo operation. Similarly, to implement back functionality in web browsers

**Queue**

A queue or FIFO (first in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: enqueue, the process of adding an element to the collection.(The element is added from the rear side) and dequeue, the process of removing the first element that was added. (The element is removed from the front side). It can be implemented by using both array and linked list.

Insertion : O(1)

Deletion : O(1)

Access Time : O(n) [Worst Case]

**Example S**o any situation where resources are shared among multiple users and served on first come first server basis. Examples include CPU scheduling, Disk Scheduling. Another application of queue is when data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

**Circular Queue** The advantage of this data structure is that it reduces wastage of space in case of array implementation, As the insertion of the (n+1)’th element is done at the 0’th index if it is empty.

**Binary Tree**  
Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.  
A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. It is implemented mainly using Links.

**Binary Tree Representation:** A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL. A Binary Tree node contains following parts.  
1. Data  
2. Pointer to left child  
3. Pointer to right child

A Binary Tree can be traversed in two ways:  
Depth First Traversal: *Inorder (Left-Root-Right), Preorder (Root-Left-Right) and Postorder (Left-Right-Root)*Breadth First Traversal: Level Order Traversal

**Binary Tree Properties:**

The maximum number of nodes at level ‘l’ = 2l-1.

Maximum number of nodes = 2h – 1.

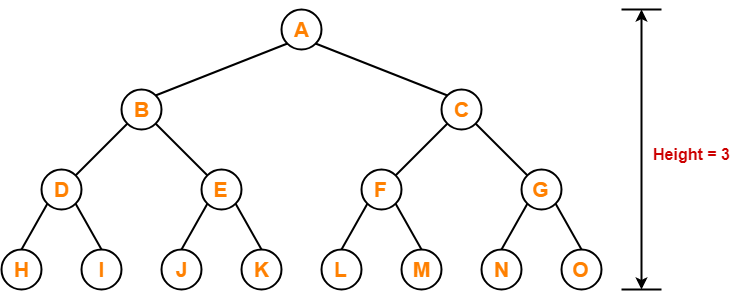
Here h is height of a tree. Height is considered

as is maximum number of nodes on root to leaf path

Minimum possible height =  ceil(Log2(n+1))

In Binary tree, number of leaf nodes is always one

more than nodes with two children.



No of Leaf nodes = 1 +(A+B ….+G)

ie 8 = 1 + 7

Time Complexity of Tree Traversal: O(n)

**Examples :** One reason to use binary tree or tree in general is for the things that form a hierarchy. They are useful in File structures where each file is located in a particular directory and there is a specific hierarchy associated with files and directories. Another example where Trees are useful is storing heirarchical objects like JavaScript Document Object Model considers HTML page as a tree with nesting of tags as parent child relations.

**Binary Search Tree**  
In Binary Search Tree is a Binary Tree with following additional properties:  
1. The left subtree of a node contains only nodes with keys less than the node’s key.  
2. The right subtree of a node contains only nodes with keys greater than the node’s key.  
3. The left and right subtree each must also be a binary search tree.

200px-Binary_search_tree.svg

Search : O(h)

Insertion : O(h)

Deletion : O(h)

Extra Space : O(n) for pointers

**h:** Height of BST

**n:** Number of nodes in BST

If Binary Search Tree is Height Balanced,

then h = O(Log n)

Self-Balancing BSTs such as AVL Tree, Red-Black

Tree and Splay Tree make sure that height of BST

remains O(Log n)

BST provide moderate access/search (quicker than Linked List and slower than arrays).  
BST provide moderate insertion/deletion (quicker than Arrays and slower than Linked Lists)

Example :  Its main use is in search application where data is constantly entering/leaving and data needs to printed in sorted order. For example in implementation in E- commerce websites where a new product is added or product goes out of stock and all products are lised in sorted order.

Binary Heap

A Binary Heap is a Binary Tree with following properties.

1) It’s a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap. It is mainly implemented using array.

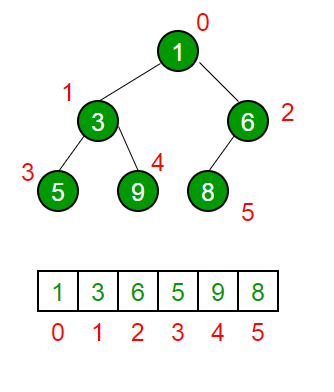
Get Minimum in Min Heap: O(1) [Or Get Max in Max Heap]

Extract Minimum Min Heap: O(Log n) [Or Extract Max in Max Heap]

Decrease Key in Min Heap: O(Log n) [Or Extract Max in Max Heap]

Insert: O(Log n)

Delete: O(Log n)



**Example :** Used in implementing efficient priority-queues, which in turn are used for scheduling processes in operating systems. Priority Queues are also used in Dijstra’s and Prim’s graph algorithms.  
The Heap data structure can be used to efficiently find the k smallest (or largest) elements in an array, merging k sorted arrays, median of a stream, etc.  
Heap is a special data structure and it cannot be used for searching of a particular element.

**HashingHash Function:** A function that converts a given big input key to a small practical integer value. The mapped integer value is used as an index in hash table. A good hash function should have following properties  
1) Efficiently computable.  
2) Should uniformly distribute the keys (Each table position equally likely for each key)

<https://www.youtube.com/watch?v=MfhjkfocRR0>

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

Chaining:The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

Space : O(n)

Search : O(1) [Average] O(n) [Worst case]

Insertion : O(1) [Average] O(n) [Worst Case]

Deletion : O(1) [Average] O(n) [Worst Case]

Hashing seems better than BST for all the operations. But in hashing, elements are unordered and in BST elements are stored in an ordered manner. Also BST is easy to implement but hash functions can sometimes be very complex to generate. In BST, we can also efficiently find floor and ceil of values.

<https://www.youtube.com/watch?v=Dk57JonwKNk>

**Example :** Hashing can be used to remove duplicates from a set of elements. Can also be used find frequency of all items. For example, in web browsers, we can check visited urls using hashing. In firewalls, we can use hashing to detect spam. We need to hash IP addresses. Hashing can be used in any situation where want search() insert() and delete() in O(1) time.

**Graph**

Graph is a data structure that consists of following two components:

A finite set of vertices also called as nodes.

A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

V -> Number of Vertices.

E -> Number of Edges.

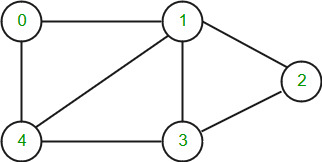
Graph can be classified on the basis of many things, below are the two most common classifications :

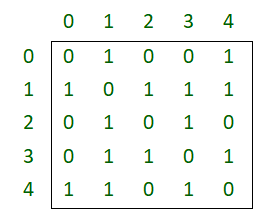
Direction :

Undirected Graph : The graph in which all the edges are bidirectional.Directed Graph : The graph in which all the edges are unidirectional.

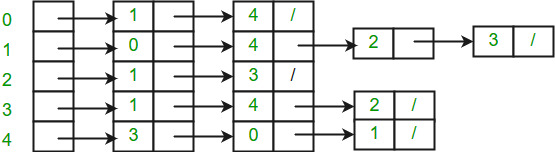
Weight :

Weighted Graph : The Graph in which weight is associated with the edges.Unweighted Graph : The Graph in which their is no weight associated to the edges.

Let us take below example graph two see two representations of graph.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Tree_overview_of_data_structures_1.jpg)

1. [](https://www.geeksforgeeks.org/wp-content/uploads/Tree_overview_of_data_structures_2.png)

*Adjacency Matrix Representation of the above graph*

1. [](https://www.geeksforgeeks.org/wp-content/uploads/Tree_overview_of_data_structures_3.jpg)

Time Complexities in case of Adjacency Matrix :

Traversal :(By BFS or DFS) O(V^2)

Space : O(V^2)

Time Complexities in case of Adjacency List :

Traversal :(By BFS or DFS) O(ElogV)

Space : O(V+E)

**Trie**

Trie is an efficient data structure for searching words in dictionaries, search complexity with Trie is linear in terms of word (or key) length to be searched. If we store keys in binary search tree, a well balanced BST will need time proportional to M \* log N, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in O(M) time. So it is much faster than BST.

Hashing also provides word search in O(n) time on average. But the advantages of Trie are there are no collisions (like hashing) so worst case time complexity is O(n). Also, the most important thing is Prefix Search. With Trie, we can find all words beginning with a prefix (This is not possible with Hashing). The only problem with Tries is they require a lot of extra space. Tries are also known as radix tree or prefix tree.

The Trie structure can be defined as follows :

struct trie\_node

{

int value; /\* Used to mark leaf nodes \*/

trie\_node\_t \*children[ALPHABET\_SIZE];

};

root

/ \ \

t a b

| | |

h n y

| | \ |

e s y e

/ | |

i r w

| | |

r e e

|

r

The leaf nodes are in blue.

Insert time : O(M) where M is the length of the string.

Search time : O(M) where M is the length of the string.

Space : O(ALPHABET\_SIZE \* M \* N) where N is number of

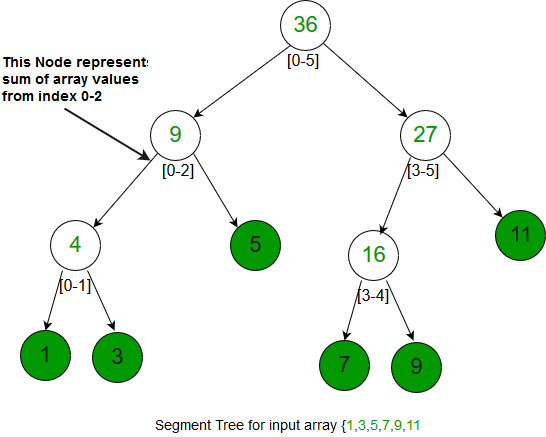
keys in trie, ALPHABET\_SIZE is 26 if we are

only considering upper case Latin characters.

Deletion time : O(M)

**Example :** The most common use of Tries is to implement dictionaries due to prefix search capability. Tries are also well suited for implementing approximate matching algorithms, including those used in spell checking. It is also used for searching Contact from Mobile Contact list OR Phone Directory.

**Segment Tree**

This data structure is usually implemented when there are a lot of queries on a set of values. These queries involve minimum, maximum, sum, .. etc on a input range of given set. Queries also involve updation of values in given set. Segment Trees are implemented using array.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Tree_overview_of_data_structures_4.jpg)

Construction of segment tree : O(N)

Query : O(log N)

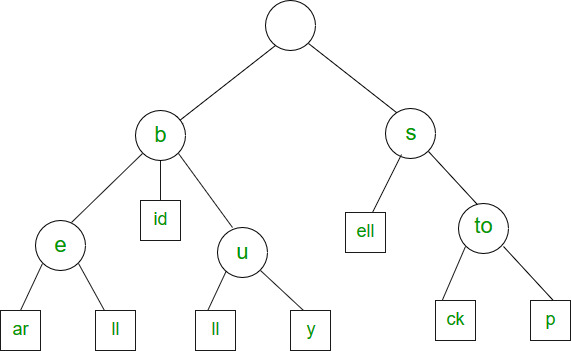
Update : O(log N)

Space : O(N) [Exact space = 2\*N-1]

**Suffix Tree**

Suffix Tree is mainly used to search a pattern in a text. The idea is to preprocess the text so that search operation can be done in time linear in terms of pattern length. The pattern searching algorithms like KMP, Z, etc take time proportional to text length. This is really a great improvement because length of pattern is generally much smaller than text.  
Imagine we have stored complete work of William Shakespeare and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. But using Suffix Tree may not be a good idea when text changes frequently like text editor, etc.

Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.  
1) Generate all suffixes of given text.  
2) Consider all suffixes as individual words and build a compressed trie.



**Example :** Used to find find all occurrences of the pattern in string. It is also used to find the longest repeated substring (when text doesn’t change often), the longest common substring and the longest palindrome in a string.

Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value 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. The process of providing only the essentials and hiding the details is known as abstraction.

Eg are

List ADT

Queue ADT

Stack ADT

**Evaluation of Postfix Expression)**

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have discussed infix to postfix conversion. In this post, evaluation of postfix expressions is discussed.

Following is algorithm for evaluation postfix expressions.

1) Create a stack to store operands (or values).

2) Scan the given expression and do following for every scanned element.

…..a) If the element is a number, push it into the stack

…..b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack

3) When the expression is ended, the number in the stack is the final answer

public int CalculatePostfix(string expr)

{

Stack<int> stack = new Stack<int>();

foreach (var item in expr.ToCharArray())

{

int value = 0;

bool isNumber = int.TryParse(item.ToString(), out value);

if (isNumber)

{

stack.Push(value);

}

else

{

item1 = Convert.ToInt32(stack.Pop());

item2 = Convert.ToInt32(stack.Pop());

switch (item.ToString())

{

case "\*":

stack.Push(item1 \* item2);

break;

case "+":

stack.Push(item1 + item2);

break;

case "-":

stack.Push(item1 - item2);

break;

case "/":

stack.Push(item1 / item2);

break;

case "%":

stack.Push(item1 % item2);

break;

}

}

}

if (stack.Count > 1)

{

throw new InvalidOperationException("String is wrong");

}

return stack.Pop();

}