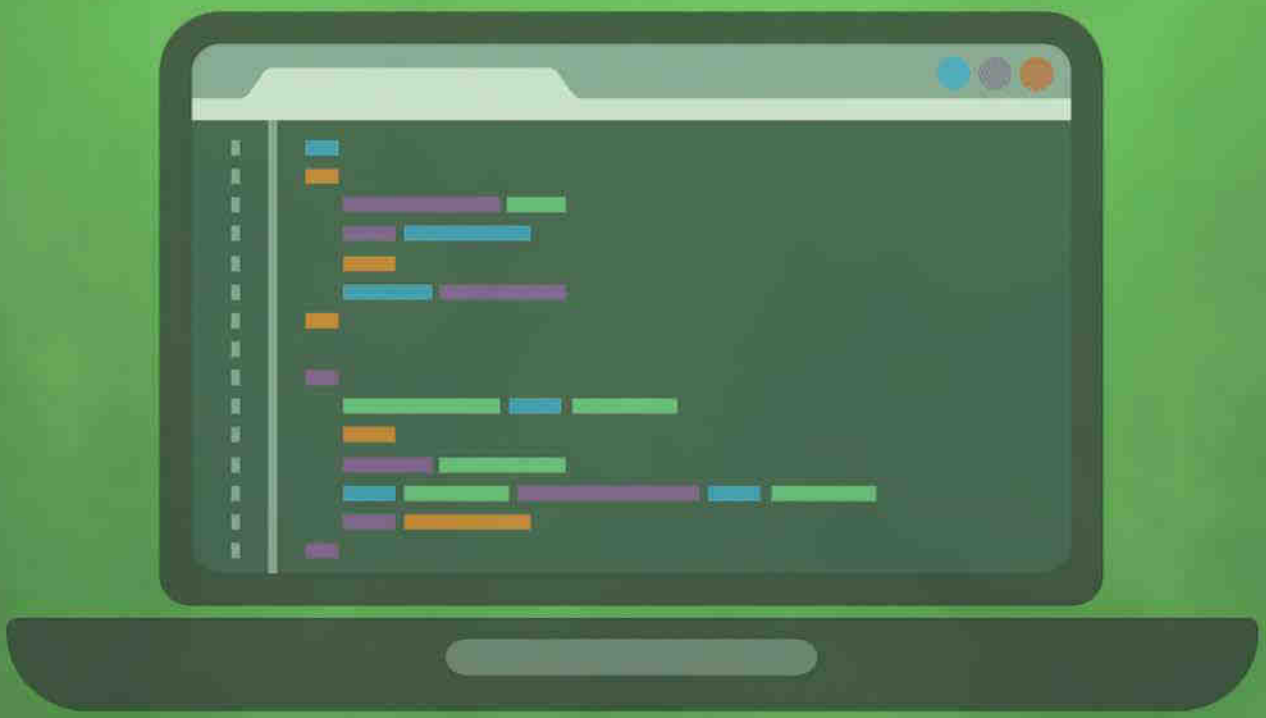
DATA STRUCTURE



INDEX

DATA STRUCTURE

Data structure is a systematic way to organize data in order to use it efficiently.

There are many ways of organizing the data in memory like array stores the elements in a continuous manner.

The data structure is not any programming language like c, c++, java etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algo. are known as Abstract data type. These abstract data types are the set of rules.

Following terms are the foundation terms of data str.

- Interface
- Implementation

INTERFACE

Each data structure has an interface.

Interface represents set of operations that a data structure supports.

An interface only provides the list of supported operations, types of parameters they can accept & return type of these operations.

IMPLEMENTATION

Implementation provides the internal representation of a data structure.

Implementation also provides the definition of the algo. used in operations of the data structure.

TYPES OF DATA STRUCTURE

There are two types of data structures:-

- Primitive Data Structure
- Non-Primitive Data Structures

PRIMITIVE Data structure

The primitive data structures are primitive data types. The int, char, float, double & pointer are the primitive data str. that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-Linear data structure

LINEAR Data Structure

Data structures where data elements are arranged sequentially or linearly where each & every element is attached to its previous & next adjacent is called a Linear data structure.

In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only.

Linear data structures are easy to implement because computer memory is arranged in a linear way.

Its examples are array, stack, queue, linked list etc.

Non-LINEAR Data structure

Data structures where data elements are not arranged sequentially or linearly are called as Non-Linear data structure.

In a non-linear data structure, single level is not involved, therefore, we can't traverse all the elements in single run only.

Non-linear data structures are not easy to implement in comparison to linear data structure.

It utilizes computer memory efficiently in comparison to a linear data structure.

Its examples are trees and graphs.

Data structures can also be classified as–

STATIC Data Structure

In static data structures the size of the structure is fixed.

The content of data structure can be modified but without changing the memory space allocated to it.

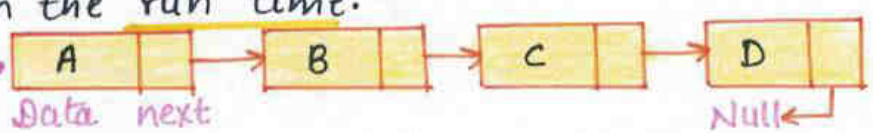
Example → array

DYNAMIC Data Structure

In dynamic data structure the size of the structure is not fixed and can be modified during the operations performed on it.

Dynamic data structures are designed to facilitate change of data structures in the run time.

Example- Linked list



Characteristics of Data Structure

Characteristics of data structure are as follows-

- Correctness – Data structure implementation should implement its interface correctly.
- Time Complexity - Running time or the execution time of operations of data structure must be as small as possible.
- Space Complexity – Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- Data Search – Consider an inventory of 1 million items of a store. If the application is to search an item in 1 million items every time slowing down the search.
- Processor speed - Processor speed although being very high, falls limited if the data grows to billion records.
- Multiple requests - As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue.

Data organised in a data structure in such a way that all items may not be required to be searched, & the required data can be searched almost instantly.

Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

Worst Case

This is the scenario where a particular data structure operation takes maximum time it can take.

If an operation's worst case time is f(n) then this operation will not take more than f(n) time where f(n) represents function of n.

Average Case

This is the scenario depicting the average execution time of an operation of a data structure.

If an operation takes f(n) time in execution, then m operations will take mf(n) time.

Best Case This is the scenario depicting the least possible execution time of an operation of a data str.

If an operation takes f(n) time in execution, then the actual operation may take time as the random number which would be maximum as f(n).

Basic Terminology in DS Data Structures are the building blocks of any program or the software.

Following terminology is used in data structure –

Data Data can be defined as an elementary value or the collection of values.

Eg: student's name & its id are the data about the student.

Group Item Data items which have subordinate data items are called group item.

Eg: Name of a student can have first name & last name.

Record Record can be defined as the collection of various data items.

Eg: If we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File A File is a collection of various records of one type of entity.

Eg: If there are 60 employees in the class, then there willbe 20 records in the related file where each record contains the data about each employee.

Attribute & Entity An entity represents the class of certain objects.

It contains various attributes. Each attribute represents the particular property of that entity.

Field Field is a single elementary unit of information representing the attribute of an entity.

Operations Performed in DS Following are some important operations are performed in data structure.

Traversing Every data structure contains set of data elements.

Traversing the data structure means visiting each ele-ment of the data structure in order to perform some specific operation like searching or sorting.

Insertion

Insertion can be defined as the process of adding the elements to the data structure at any loc^{n}.

If the size of data structure is 'n' then we can only insert 'n-1' data elements into it.

Deletion

The process of removing an element from the data structure is called deletion.

We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then underflow occurs.

Searching

The process of finding the location of element within the data structure is called searching.

There are two algorithms to perform searching, Linear search and Binary search.

Sorting

The process of arranging the data structure in a specific order is known as sorting.

There are many algorithms that can be used to per-form sorting, for eg. insertion sort, selection sort & bubble sort etc.

Merging

When two lists List A and List B of size M & N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

Advantages of DS

Following are some adva-ntages of data structure.

Efficiency

Efficiency of a program depends upon the choice of data structures.

Data structure helps in efficient storage of data in the storage device.

Data structure provides effective and efficient pro-cessing of small as well as large amount of data.

Manipulation of large amount of data can be carr-ied out easily with the use of good data str. approach.

Reusability Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place.

Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction Data structure is specified by the ADT which provides a level of abstraction.

The client program uses the data structure through interface only, without getting into the implementation details.

DATA-STRUCTURE ALGORITHM

An algorithm is a process or set of rules required to perform calculations or some other problem solving operations especially by a computer.

It is a step by step procedure, which defines a finite set of instructions to be executed in a certain order to get output.

It is not the complete program or code, it is just a solution (logic) of a problem, which can be represented either as an informal description using a flowchart or Pseudocode.

Characteristics of Algorithms The following are the characteristics of an algorithm →

Input An algorithm has some input values. We can pass '0' or some input value to an algorithm.

Output We will get '1' or more output at the end of an algorithm.

Unambiguity An algorithm should be unambiguous which means that the instructions in an algorithm should be clear & simple.

Finiteness An algorithm should have finiteness means that the algorithm should contain a limited number of instructions should be countable.

Effectiveness An algorithm should be effective as each instruction in an algorithm affects the overall process.

Language Independent An algorithm must be language-independent so that instruction can be implemented with the same output.

Dataflow of an Algorithm

Following parts are included in dataflow of an algorithm–

Problem A problem can be a real-world problem or any instance from the real-world for which we need to create a program or the set of instructions.

The set of instructions is known as an algorithm.

Algorithm An algorithm will be designed for a problem which is a finite step-by-step procedure.

Input After designing an algorithm, the required & the desired inputs are provided to the algorithm.

Processing Unit The input will be given to the processing unit, and the processing unit will produce the desired output.

Output The output is the outcome or result of the program.

Eg Algorithm to add two given numbers–

The following are the steps required to add two numbers.

STEP1: Start STEP2: Declare three variables a, b & sum.

STEP3: Enter the values of a&b STEP4: Add the values of a & b and store the result in the sum variable, sum=a+b.

STEP5: Print sum STEP 6: stop.

Factors of an Algorithm

The following are the factors that we need to consider the designing an algorithm:–

Modularity If any problem is given and we can break that problem into small-small modules or small-small steps, which is a basic definition of an algo.

Correctness The correctness of an algorithm is defined as when the given inputs produce desired output.

Maintainability Here maintainability means that the algo. should be designed in a very simple structured way.

When we redefine the algorithm, no major change will be done in the algorithm.

Functionality It considers various logical steps to solve the real world problem.

User-Friendly If the algorithm is not user friendly, then the designer will not be able to explain it to the programmer.

Robustness Robustness means that how an algorithm can clearly define our problem.

Simplicity If the algorithm is simple then it is easy to understand.

Extensibility If any other algorithm designer or pro-grammer wants to use our algorithm then is should be extensible.

Approaches of Algorithm The following are the app-roaches used after considering both the theoretical and practical importance of designing an algorithm-

Brute force Algorithm The general logic structure is applied to design an algorithm.

It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required sol.

These algorithm are of two types -

Optimizing Finding all the solutions of a problem and then take out the best solution or if the value of the best sol.^{n} is known it will terminate if the best sol.^{n} is known.

Sacrificing As soon as the best solution is found, then it will stop.

Divide & Conquer It allows us to design an algorithm in a step-by-step variation.

It breaks down the algorithm to solve the problems in different methods.

Valid output is produced for the valid input & this valid o/p is passed to some other function.

Greedy Algorithm It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution.

It is easy to implement and has a faster execution time, but there are very rare cases in which it provides the optimal solution.

Dynamic Programming It makes the algorithm more efficient by storing the intermediate results.

It follows five different steps to find optimal sol.^{n} -

1. It breaks down the problem into a subproblem to find the optimal solution.
2. After breaking down the problem, it finds the optimal solution out of these subproblems.
3. Stores the result of the sub problems is known as memorization.
4. Reuse the result so that it cannot be recomputed for the same subproblems.
5. Finally, it computes the result of the complex program.

Branch and Bond Algorithm ☛ The branch and bound algorithm can be applied to only integer programming problems.

☛ This approach divides all the sets of feasible solutions into smaller subsets, & these subsets are evaluated to find the best solution.

Randomised Algorithm ☛ The algorithms that have some defined set of inputs & required output, & follow some described steps are known as deterministic algorithms.

☛ In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output which is random in nature.

☛ Randomized algorithms are simpler and efficient than the deterministic algorithm.

Back-tracking ☛ Backtracking is an algorithmic technique that solves the problem recursively and remove the solution if it does not satisfy the constraints of a problem.

☛ Sort, Search, delete, insert & update are five major categories of an algorithm.

Algorithm Analysis ☛ The algorithm can be analyzed in two levels; i.e. first is before creating the algorithm and second is after creating the algorithm.

☛ The following are the two analysis of an algorithm-

Priori Analysis ☛ Priori analysis is the theoritical analysis of an algorithm which is done before implementing.

☛ Various factors can be considered before implementing the algo. like processor speed, which has no effect on the implent. part.

Posterior Analysis

Here, posterior analysis is a pra-ctical analysis of an algorithm.

The practical analysis is achieved by implementing the algorithm using any programming language.

This analysis basically evaluate that how much running time and space taken by the algorithm.

Algorithm Complexity

The performance of the algorithm can be measured in two factors:

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 no-tation to represent the time complexity.

The time complexity is mainly calculated by counting the number of steps to finish the execution.

Eg
```
sum = 0; /* we have to calculate sum of n numbers */
for i=1 to n
    sum = sum+i; /* When the loop ends then sum holds the sum of n numbers */
return sum;
```

In this code time complexity of the loop statement will be atleast n, & if the value of 'n' increases then the time complexity also increases.

The complexity of the code is not dependent on the value of n & will provide the result in one step.

We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

Space Complexity

An algorithm's space complexity is the amount of space required to solve a problem.

Space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

(1) To store program instructions
(2) To store constant values
(3) To store variable values
(4) To track the function calls, jumping statements etc.

Auxiliary Space

The extra space required by the algorithm, excluding the input size, is known as auxiliary space. So,

Space complexity = Auxiliary space + Input size

Types of Algorithm

Following are the types of algo.

- Search Algorithm
- Sort algorithm

Search Algorithm

In case of computers, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in memory & provides that data to the user.

There are mainly two techniques available to search the data in an array -

- Linear Search
- Binary Search

Linear Search

Linear Search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until element not found.

It compares the elements to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns '-1'.

This algorithm can be implemented on the unsorted list.

Binary Search

A Binary algorithm is the simplest algorithm that searches the element very quickly.

It is used to element from the sorted list.

The elements must be stored in sequential order or the sorted manner to implement the binary algorithm.

It can not be implemented if the elements are stored in a random manner & it is used to find middle element of the list.

Sorting Algorithm

Sorting algorithms are used to rearrange the elements in an array either in an ascending order or descending order.

The comparison operator decides the new order of the elements.

It produces information in a sorted order, which is a human-readable format.

ASYMPTOTIC ANALYSIS

Asymptotic analysis of an algorithm refers to defining the mathematical boundation / framing of its run time performance.

Using asymptotic analysis, we can conclude best case, average case and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e. if there is no i/p to the algorithm, it is concluded to work in constant time.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.

Eg: Let the running time of an operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means running time of first operation will increase linearly with the increase in n & the running time of second operation will increase exponentially with 'n'.

Usually, the time required by an algorithm falls under three types –

- Best case: Minimum time required for program execution.
- Average case: Average time required for execution.
- Worst Case: Maximum time required for execution.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- 'O' Notation (oh)
- 'Ω' Notation (omega)
- 'θ' Notation (Theta)

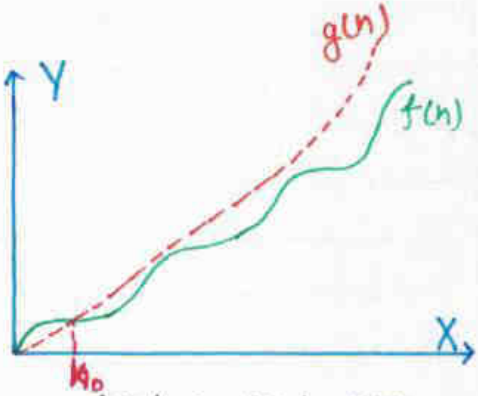
Big-oh Notation (O)

'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 'O(n)' provides an upper bound on a function which ensures that the function never grows faster than the upper bound.

It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation.

It is represented as shown in the graph. →



f(n) & g(n) are two functions.

In graph $f(n)$ & $g(n)$ 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 constant c & n_0 such that :

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

This implies that $f(n)$ does not grow faster than $g(n)$ or $g(n)$ is an upper bound on the function $f(n)$.

Eg. $f(n) = 2n+3$ & $g(n) = n$

we have to find Is $f(n) = O(g(n))$?

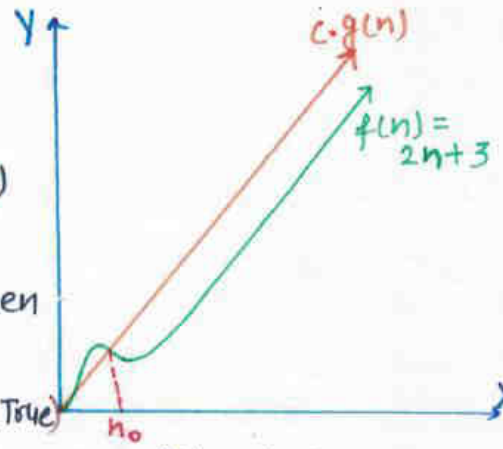
Solution → To check $f(n) = O(g(n))$, it must satisfy the cond. → $f(n) <= c.(g(n))$

⇒ replace $f(n)$ by $2n+3$ & $g(n)$ by n.

⇒ $2n+3 <= c.n$; assume $c=5$ & $n=1$ then

⇒ $2*1+3 <= 5*1$ ⇒ $5 <= 5$ (True.)

⇒ If $n=2$; $2*2+3 <= 5*2$ ⇒ $7 <= 10$ (True)



⇒ we know that, for any value of n, it will satisfy the above condition, i.e. $2n+3 <= c.n$.

⇒ Therefore we can say it is satisfying the condition & $f(n)$ is big oh of $g(n)$, it concludes that $c.g(n)$ is the upper bound of the $f(n)$.

⇒ It behaves in a linear manner in a worst-case.

Omega Notation (Ω)

It basically describes the best-case scenario which is opposite to the big O Notation.

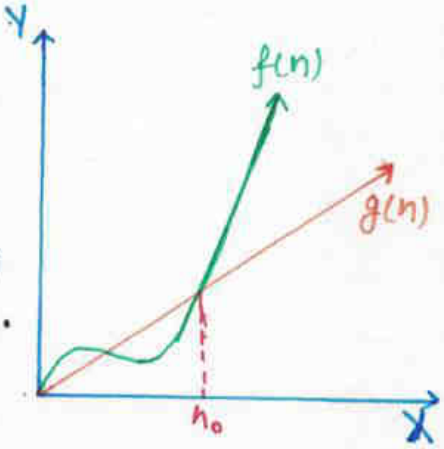
It is the formal way to represent the lower bound of an algorithms running time.

It determines what is the fastest time that an algo. can run.

We use big-Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large i/p size.

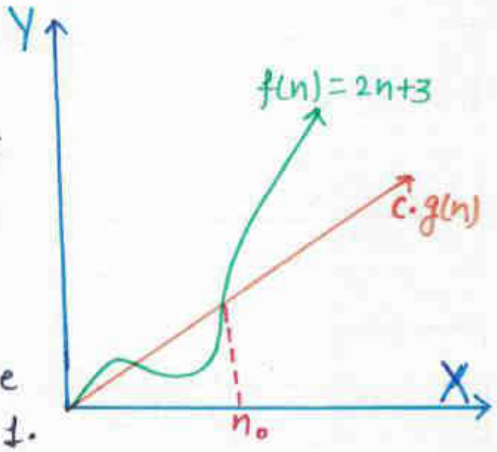
$f(n)$ & $g(n)$ are two functions then $f(n) = \Omega(g(n))$ as $f(n)$ is omega of $g(n)$.

$$f(n) >= c.g(n) \text{ for all } n \geq n_0 \text{ & } c > 0$$



Eg: $f(n) = 2n+3$ $g(n) = n$

we have to find Is $f(n) = \Omega(g(n))$?



solution→ To check $f(n) = \Omega g(n)$, it must satisfy the condition $f(n) >= C.\ g(n)$

⇒ replace $f(n)$ by $2n+3$ & $g(n)$ by n.

⇒ $2n+3 >= c*n$ assume $c = 1$

⇒ $2n+3 >= n$, this equation will be true for any value of n starting from 1.

⇒ we can see in the above figure $g(n)$ function is the lower bound of the $f(n)$ function when the value of c is equal to 1.

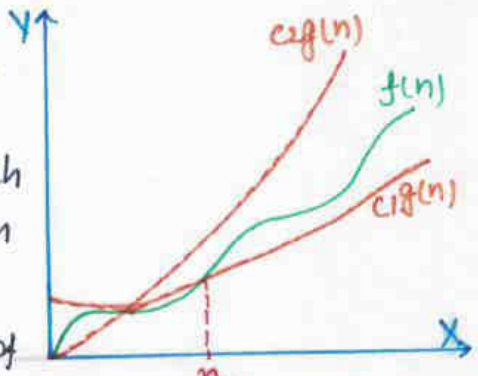
⇒ This notation gives the fastest running time.

Theta Notation (θ)

☞ The theta notation mainly describes the average case scenarios.

☞ It represents the realistic time complexity of an algorithm.

☞ Big theta (θ) is mainly used when the value of worst-case and the best-case is same.



☞ It is the formal way to express both the upper bound and lower bound of an algorithm running time.

☞ let $f(n)$ & $g(n)$ be the functions of n where n is the steps required to execute the program.

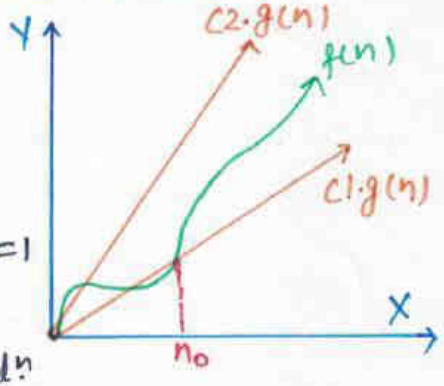
☞ $f(n) = \theta g(n)$, this condition is satisfied only if when:

☞ The condition $f(n) = \theta g(n)$ will be true if and only if $C1.g(n) <= f(n) <= C2.g(n)$ & $C1 * C2 > 0$

$C1.g(n)$ is less than or equal to $f(n)$ & $C2.g(n)$ is greater than or equal to $f(n)$.

Eg: $f(n) = 2n+3$ $g(n) = n$

Is $f(n) = \theta(g(n))$?



sdution→ $C1.g(n) <= f(n) <= C2.g(n)$

⇒ replace $g(n)$ by n & $f(n) = 2n+3$

⇒ $C1.n \leq 2n+3 \leq C2.n$, if $C1 = 1$ & $C2 = 5$, $n = 1$

⇒ $1*1 <= 2*1+3 <= 5*1$ ⇒ $1 <= 5 <= 5$

⇒ If $n = 2$; $2 <= 7 <= 10$ satisfies the condn

⇒ Hence we can say $f(n)$ is big theta of $g(n)$. So, this is average case scenario which provides the realistic time complexity.

Following is the list of some common asymptotic notations.

constant	$O(1)$	logarithmic	$O(\log n)$	linear	$O(n)$
$n \log n$	$O(n \log n)$	quadratic	$O(n^2)$	Cubic	$O(n^3)$
polynomial	$n^{O(1)}$	exponential	$2^{O(n)}$		

ARRAYS

Array is a container which can hold a fix number of items and these items should be the same type.

Most of the data structures make use of array to implement their algorithm.

Following are the important terms to understand the concept of array:

- Element – Each item stored in an array called element.
- Index – Each location of an element in an array has a numerical index, which is used to identify the element.

Basic Operations

Following are the basic operations supported by an array–

- Traverse
- Insertion
- Deletion
- Search
- Update

Traverse Operation

This operation is performed to traverse through the array elements.

It prints all array elements one after another.

Eg: Traverse Operation

```
#include <stdio.h>
void main () {
   int LA[] = {1,3,5,7,8};
   int item = 10, k=3, n=5;
   int i=0 , j=n;
   printf ("The original array elements are:");
   for (i=0; i<n; i++) {
      printf ("LA[%d] = %d\n", i, LA[i]);
   }}
```

o/p:
The original array elements are:
LA[0] = 1 LA[3] = 7
LA[1] = 3 LA[4] = 8
LA[2] = 5

Insertion Operation

Insert operation is to insert one or more data elements into an array.

Based on the requirement, a new element can be added at beginning, end or any given index of array.

Eg: Inserting elements to array

```
#include<stdio.h>
void main() {
    int LA[] = {1,3,5,7,8};
    int item = 10, k=3, n=5;
    int i=0 , j=n;
    printf(" The original array
    elements are: \n");
    for (i=0; i<n; i++){
      printf(" LA[%d] = %d\n", i,
                              LA[i]);
    }
    n = n+1;
    while (j>=k) {
      LA[j+1] = LA[j];
       j = j-1; }
    LA[k] = item;
    printf(" The array elements
    after insertion : \n");
    for (i=0; i<n; i++){
      printf(" LA[%d] = %d\n", i,
                              LA[i]);
    }}
```

OIP: The original array elements are:

LA[0] = 1 LA[3] = 7
LA[1] = 3 LA[4] = 8
LA[2] = 5

The array elements after insertion:

LA[0] = 1 LA[3] = 10
LA[1] = 3 LA[4] = 7
LA[2] = 5 LA[5] = 8

Deletion Operation

As the name implies, this operation removes an element from the array & then reorganizes all of the array elements.

Eg: deleting elements from array

```
#include <stdio.h>
void main() {
    int LA[] = {1,3,5,7,8};
    int k=3, n=5;
    int i, j;
    printf(" The original array
    elements are: \n");
    for (i=0; i<n; i++) {
    printf(" LA[%d] = %d\n", i,
                              LA[i]);
    }
    j=k;
    while (j<n) {
      LA[j-1] = LA[j];
       j = j+1;
    }
      n = n-1;
    printf(" The array elements aft-
    -er deletion : \n");
    for (i=0; i<n; i++) {
      printf(" LA[%d] = %d\n", i,
                              LA[i]);
    }}
```

OIP: The original elements are:

LA[0] = 1 LA[3] = 7
LA[1] = 3 LA[4] = 8
LA[2] = 5

The array elements after deletion:

LA[0] = 1 LA[2] = 7
LA[1] = 3 LA[3] = 8

Search Operation

This operation is performed to search an element in the array based on the value or index.

Eg: searching element from array

```
#include <stdio.h>
void main() {
    int LA[] = {1,3,5,7,8};
    int item = 5, n=5;
    int i= 0, j= 0;
    printf("The original array elements are: \n");
    for(i=0; i<n; i++){
     printf("LA[%d] = %d\n", i,
     }                        LA[i]);
    while (j<n) {
      if (LA[j] == item){
        break;
      } j = j+1; }
    printf("Found element %d at position %d", item, j+1); }
```

O/P:
The original array elements are:
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3.

Update Operation

This operation is performed to update an existing array element located at the given index.

Eg: updating elements to array

```
#include <stdio.h>
void main () {
    int LA[] = {1,3,5,7,8};
    int k = 3, n=5, item=10;
    int i, j;
    printf("The original array elements are: \n");
    for (i=0; i<n; i++){
      printf("LA[%d] = %d\n",
      i, LA[i]);
    }
    LA[k-1] = item;
    printf("The array elements after updation : \n");
    for (i=0; i<n; i++) {
      printf("LA[%d] = %d\n", i,
      LA[i]);
    }}
```

O/P:
The original elements are:
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation:
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8

LINKED LIST

Linked list is a linear data structure that includes a series of connected nodes.

Linked list can be defined as the nodes that are randomly stored in the memory.

Linked list is also a sequence of data structures, which are connected together via links.

Node in the linked list contains two parts, i.e. first is the data part and second is the address part.

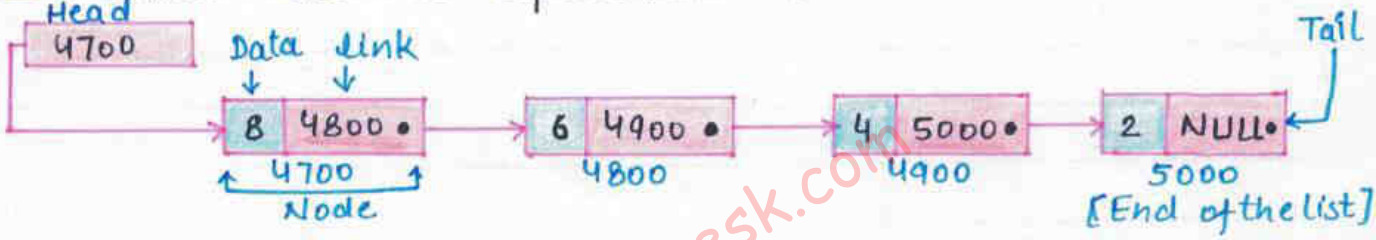
The last node of the list contains a pointer to the null.

Linked list is the second most used data structure.

Following are the important terms to understand the concept of linked list -

- **Link :** Each link of a linked list can store a data called an element.
- **Next :** Each link of a linked list contains a link to the next link called link.
- **Linked List :** A linked list contains the connection link to the first link called first.

Linked list are represented as -



Here, linked list is represented as the connection of nodes in which each node points to the next node of the list.

Linked list is classified into the following types -

- Singly - linked list
- Doubly linked list
- Circular singly linked list
- Circular doubly linked list

It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array.

Linked list contains two parts, and both are of different types, i.e. one is the simple variable, while another is the pointer variable.

The declaration of linked list is given as follows-

syntax
```
struct node { int data;
              struct node *next; }
```

SINGLY LINKED LIST

A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node.

Each element in a linked list is called a node.

A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.

The first node is called the head; it points to the first node of the list and helps us to access every element.

The last node, also sometimes called the tail, points to NULL which tells us when the list ends.

There are various operations performed on singly linked list.

Node Creation

Node is created as follows.

```
struct node {
    int data;
    struct node *next; };
```

```
struct node *head, *ptr;
ptr = (struct node*) malloc (sizeof
      (struct node*));
```

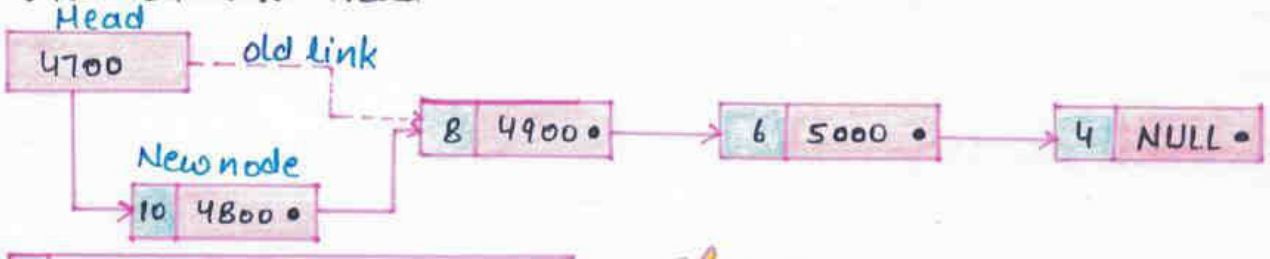
Insertion

The insertion into a singly linked list can be performed at different positions.

We can insert element at the beginning, at the end & can also insert in between.

Inserting at the beginning

In order to insert the new node at the beginning, we would need to have the head pointer pointing to this new node & the new node's pointer to the current head.

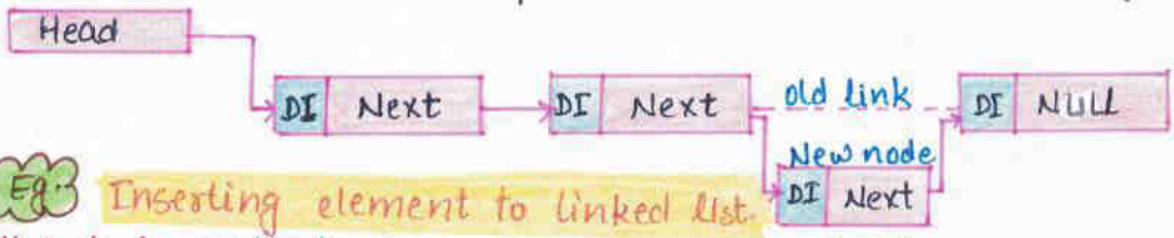


Inserting at the end

In order to insert the new node at the end, the second last node of the list should point to new node & new node will point to NULL.

DI → data item

Inserting in between In order to insert a new node in between next of first item (before than new added location) will point to the data item of new node.



Eg: **Inserting element to linked list.**

```
# include <stdio.h>
# include <stdlib.h>
  struct node {
      int data; struct node*next;
  }; struct node *head;
  void beginsert ();
  void lastinsert ();
  void randominsert ();
  void main () {
     int choice = 0;
     while (choice != 4) {
    printf (" Main Menu ");
    printf ("\n choose one option");
    printf ("\n 1. Insert in beginning
 \n2. Insert at last \n 3. Insert in bet:
  \n 4. exit\n" );
    printf (" Enter your choice ? ");
    scanf ("\n%d " &choice );
    switch (choise) {
      case 1: beginsert ();
      break;
      case 2: lastinsert ();
     break;
     case 3: randominsert ();
     break;
     case 4:
      exit (0); break;
     default: print (" Enter valid choice");
     }}}
     void beginsert () {
      struct node *ptr;
       int item ;
      ptr = (struct node* )malloc (
          sizeof (struct node*));
      if (ptr == NULL) {
       printf ("\n overflow"); }
      else {
       printf (" Enter value \n");
       scanf (" %d ", & item);
        ptr->data = item;
        ptr->next = head;
         head = ptr;
        printf (" Node inserted");
      }}
     void lastinsert () {
      struct node *ptr,* temp;
      int item;
      ptr = (structnode* )malloc (
      sizeof (struct node));
```

```
if(ptr==NULL) {
 printf("\n Overflow"); }
else {
 printf(" Enter value?");
 scanf("%d", &item);
 ptr->data = item;
 if (head==NULL) {
   ptr next = NULL;
    head= ptr;
   printf("\n Node inserted");}
  else {
    temp= head;
    while (temp->next!=NULL)
    { temp = temp->next; }
    temp -> next = ptr;
     ptr -> next = NULL;
     printf ("\n Node inserted");
  }}}
void randominsert () {
   int i, loc, item;
   struct node *ptr, *temp;
   ptr = (struct node*) malloc
   (sizeof (struct node));
   if (ptr==NULL) {
    printf("\n overflow"); }
else {
 printf(" Enter element value");
 scanf("%d", &item);
  ptr->data = item;
 printf("\n Enter Location");
 scanf("\n%d", &loc);
 temp = head;
 for (i=0; i<loc; i++) {
 temp = temp->next;
  if (temp==NULL) {
   printf(" can't insert");
    return; } }
 ptr->next = temp->next;
 temp->next = ptr;
  printf("Node inserted");
 }}
```

```
o/p: Main Menu
Choose option:
1. Insert at beginning
2. Insert at last
3. Insert at any random loc.n
4. Exit
Enter your choice?
1
Enter value
1
Node inserted.
Again repeat these steps
```

Deletion Operation

The deletion of a node from a singly linked list can be performed at different positions.

Based on the position of the node being deleted, the operation is categorized into the following categories:

Deleting at the beginning

Deleting a node from the beginning of the list is the simplest operation.

The first node of the list is to be deleted, therefore we just need to make the head, point to the next of the head.

Deleting at the end

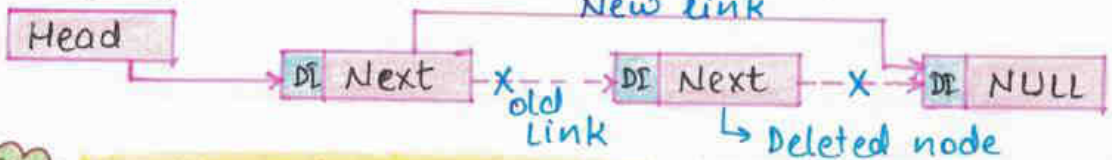
There are two scenarios in which, a node is deleted from the end of the linked list-

(1) There is only one node is the list & that needs to be deleted. Here, only one node head of the list will be assigned to null.

(2) There are more than one node in the list and last node of the list will be deleted.

Deleting in between

We can delete the node after a specified node in the linked list.



Eg: Deleting elements from LL

```
# include <stdio.h>
# include <stdlib.h>
 void create (int);
 void begdelete ();
 struct node {
      int data; struct node* next;
  }; struct node *head;
 void main() {
    int choice, item;
   do {
  printf ("\n 1. Append List \n 2.
  Delete node \n 3. Exit \n 4. Enter
  your choice");
  scanf ("%d", &choice);
  switch(choice) {
  case 1: printf (" Enter item");
   scanf ("%d", &item);
   create (item);
    break;
  case 2: begdelete ();
   break;
  case 3: exit (0);
   break;
   default: printf (" Valid choice");
  }}
  while (choice != 3); }
```


```
void create (int item) {
 struct node *ptr = (struct node*)
   malloc (size of (struct node*));
   if (ptr == NULL) {
      printf ("overflow"); }
    else {
      ptr → data = item;
      ptr → next = head;
      head = ptr;
     printf ("\Node inserted ");
  }}
  void begdelete () {
  struct node * ptr ;
    if (head = NULL) {
     printf ("\nList is empty");
      }
    else {
     ptr = head;
     head = ptr → next ;
    free (ptr);
     printf ("\n Node deleted
       from begining");
  }}
```

```
O|P:
1. Append List
2. Delde node
3. Exit
4. Enter your choice? 1
Enter the item:
23
Enter your choice? 2
Node deleted from the
begining.
```

If there are no duplicate elements worst case time complexity will be 'O(n)' because we are traversing list at least half.

Traversing Operation

Traversing means visiting each node of the list once in order to perform some operations.

Eg: Traversing elements of LL

```
#include <stdio.h>
# include <stdlib.h>
 void display (struct node *head)
 { struct node *p;
    if (head == NULL)
    printf ("List is empty ");
    return; }
    p = head;
  printf ("List is: \n");
  while (p != NULL)
  { printf ("%d", p → info);
    p = p → link; }
    printf ("\n\n"); }
void Count (struct node * head)
 { struct node * p;
    int cnt = 0;
    p = head;
 while (p != NULL ) {
  p = p → link ;
  cnt ++; }
  printf (No. of elements
    are %d\n", cnt);
 }
```

Searching Operation

Searching is performed in order to find the location of a particular element in the list.

Searching any element in the list needs traversing through the list & make comparison of every element of the list with the specified element.

If the element is matched with any of the list element then the location of the element is returned from the function.

Eg: searching elements

```
#include <stdio.h>
#include <stdlib.h>
void search (struct node *head,
     int item ) {
     struct node *p = head;
      int pos = 1;
   while (p! = NULL)
   { if (p->info == item ) {
printf (" Item %d found at
position %d ", item , pos);
   return; }
    p=p->link;
     pos++;
}
printf (" Item %d not found
in list \n ", item);
 }
```

Operations	Time complexity		space complexity	
	Average	Worst	Average	Worst
Access	$\theta(n)$	$O(n)$	$O(n)$	$O(n)$
Search	$\theta(n)$	$O(n)$	$O(n)$	$O(n)$
Insertion	$\theta(1)$	$O(1)$	$O(n)$	$O(n)$
Deletion	$\theta(1)$	$O(1)$	$O(n)$	$O(n)$

Drawbacks

Linked list use more memory than arrays because of the storage used by the pointer for address of next node.

Nodes are stored incontiguously, greately increasing the time periods required to access individual elements with in the list, especially with CPU cache.

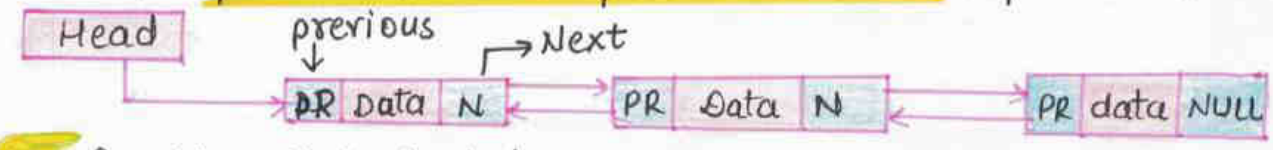
Nodes in a linked list must be read in order from beginning as linked lists are inherently sequential access.

Difficulties arises in linked list when it comes to reverse traversing.

DOUBLY LINKED LIST

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in sequence.

A doubly linked list's node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer).



Doubly linked list consumes more space for every node.

We can easily manipulate the elements of the list since the list maintains pointers in both the directions.

Node Creation

Node is created as follows —

Syntax→

```
struct node { struct node *prev;
    int data;   struct node* next; };
    node *head;
```

Insertion Operation

The insertion into a doubly linked list can be performed at different positions.

We can insert element at the beginning, at the end & can also insert in between.

Inserting at beginning

There are two scenarios of inserting any element into doubly linked list, either the list is empty or it contains at least one element.

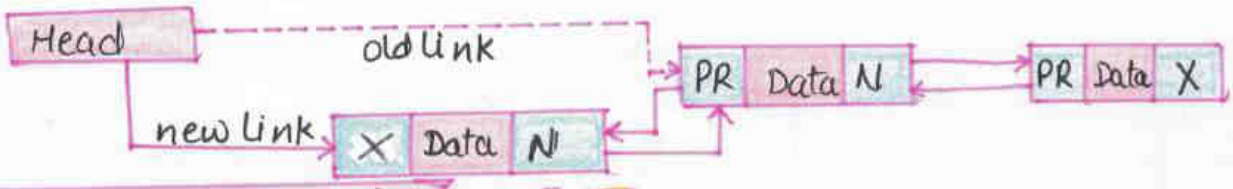
(1) Allocate the space for the new node in the memory.

(2) In this case, the node will be inserted as the only node of the list and therefore the prev & next pointers of the node will point to NULL & the head pointer will point to this node.

(3) In second scenario, the condition head==NULL becomes false and node will be inserted in beginning.

(4) The next pointer of the node will point to the existing head pointer of the node.

(5) The previous pointer of the existing head will point to the new node being inserted.

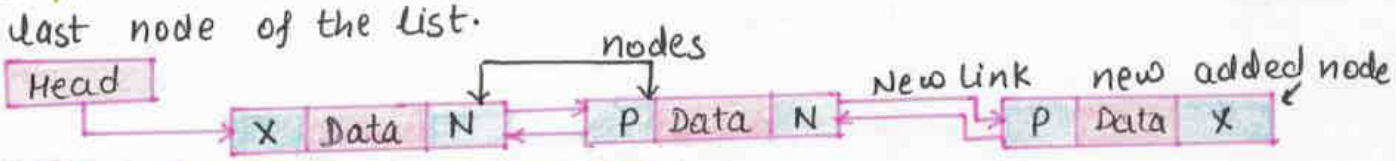


Inserting at the end

We can insert a node in doubly linked list in two cases either list is empty or one or more than one elements are in linked list.

(1) When list is empty, the node will be inserted as the only node of the list and therefore the prev and next pointer of the node will point to NULL & the head pointer will point to this node.

(2) In second scenario, the node will be inserted as the last node of the list.



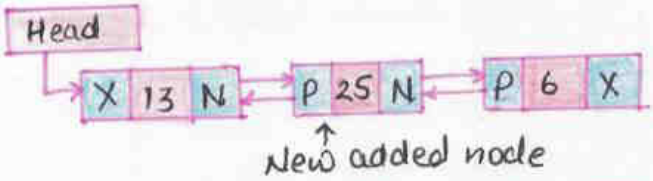
Eg: Inserting node to DLL

```
#include <stdio.h>
#include <stdlib.h>
void insertlast(int);
 structnode {
    int info; struct node* next;
     structnode* prev;
     };
 void display (structnode * head)
  { struct node *p;
     if (head == NULL)
  {printf ("List is empty\n");
     return ; }
    p= head;
   printf ("List is: \t ");
   while (p!= NULL)
   { printf ("%d", p→ info);
     p = p → next;
     printf ("\n\n");
   }
```

```
struct node * insbeg (structnode*
        head int data)
{ struct node *temp;
 temp = (struct node*) malloc
   (sizeof (structnode));
 temp → info = data;
 temp → prev = NULL;
 temp → next = head;
   if (head)
   { head→ prev = temp;
     head = temp;
  returnhead; } }
struct node* insestlast(struct
 node * head int data)
{ struct node *temp, *p;
  temp = (struct node*)
   malloc (sizeof (struct node));
  temp → info = data;
   p= head;
```

```
if (p)
{ while (p->next!=NULL)
    p = p -> next;
  p->next = temp;
  temp -> prev = p;
 } /* assigning temp to p */
 else
    head = temp;
 return head; }
```


```
/* insert after a given pos.n */
struct node * insmid (struct node *
        head, int data, int item)
{ struct node *temp, *p;
  temp = (struct node *) malloc
   (sizeof ( struct node));
  temp->info = data;
   p = head;
  while ( p != NULL )
{ if (p -> info == item)
  {
   temp -> prev = p;
   temp -> next = p -> next;
  if (p next != NULL)
   p -> next -> prev = temp;
   p->next = temp;
  return head; }
   p = p next ; }
  printf (" %d not present in
  the list\n");
  return head; }
```

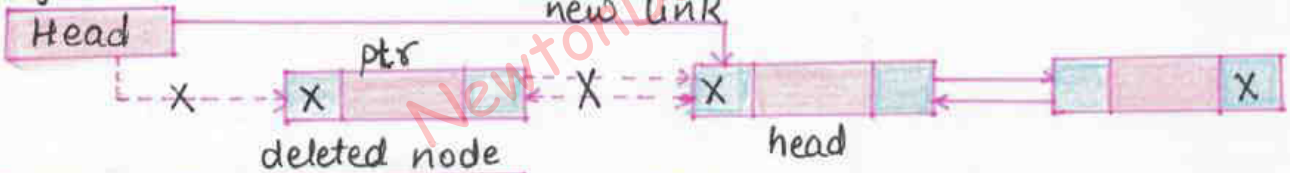


Deletion Operation

We can delete the element from beginning, end & in between the linked list.

Deleting at beginning

It is simplest operation, we just need to copy the head pointer to the pointer ptr & shift the head pointer to its next.

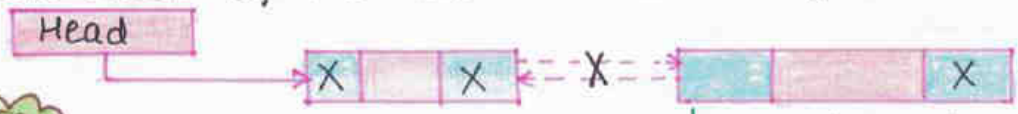


Deleting at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

(1) If the list is already empty then the cond.n head == NULL will true and the operation can not be carried on.

(2) If there is only one node then we just need to assign the head of the list to NULL & free head.



Eg.

```
if (temp → info == data)
{
  temp → prev → next = NULL;
    free temp;
    return head;
}
printf ("Element %d not found", data);
return head; }
```

Traversing means visiting each node of the list once to perform some specific operation.

For this purpose, copy the head pointer in any of the temporary pointer, then traverse through the list using while loop. Keep shifting until we find the last node.

We need to traverse the list in order to searching a specific element in the list.

Compare each element of the list with the item which is to be searched.

If the item matched with any node value then the location of that value is returned otherwise NULL is returned.

Drawbacks / Cons

Doubly linked list has two major drawbacks:

(1) It occupies more space or more memory. That's because here we have two pointers.

DLL → 2ptr, SLL → 1ptr

(2) We have to adjust, modify or edit the list then more number of pointers are updated.

(3) When we insert/delete a node then more number of pointer operations are performed.

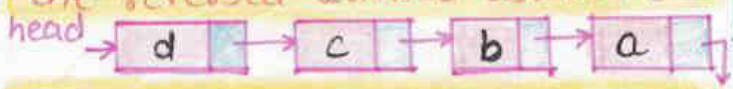
Time complexity for inserting & deleting a node in worst case & average case will be 'O(1)'

For searching and accessing a node time complexity of worst case & average case will be 'O(n)'.

Que

Consider the problem of reversing a singly linked list.

the reversed linked list looks -



Which one of the following statement is TRUE about time complexity for space O(1)?

(A) The best algorithm for the problem takes $\Theta(n)$ in the worst case.

(B) $\Theta(n \log n)$ (C) $\Theta(n^2)$

(D) not possible in case of O(1)

option [A].

Que

N items are stored in a sorted doubly linked list. For a delete operation, a pointer is provided to the record to be deleted. For a decrease-key operation, a pointer is provided to the record on which the operation is to be performed.

An algorithm performs the following operations on the list in order. $\Theta(N)$, delete, $O(\log N)$ insert $O(\log N)$ find, & $\Theta(N)$ decrease-key. What is the time complexity of all these operations put together.

(A) $O(\log^2 N)$ (C) $O(N^2)$

(B) $O(N)$ (D) $O(N^2 \log N)$

option B. O(N)

Solution: ⇒ The time complexity of decrease key is $\Theta(1)$ since we have the pointer to record.

⇒ We must keep the DLL sorted and after the decrease key operation we need to find the new location of the key.

⇒ This step will take $\Theta(N)$ time and since there are $\Theta(N)$ decrease key operations, the time comp. becomes $O(N^2)$.

Que

What is the worst case complexity of inserting n elements into an empty linked list, if the linked list needs to be maintained in sorted order?

(A) $\Theta(n)$ (C) $\Theta(n^2)$

(B) $\Theta(n \log n)$ (D) $\Theta(1)$

Solution: option B & C

There are two possible cases:

1. When we are inserting an element to empty linked list and to perform sorted order list of every element take $O(n^2)$.

⇒ Each insertion into a sorted LL will take $\Theta(n)$ & hence the total cost for n operations $\Theta(n^2)$.

2. using merge sort all elements will take $O(n \log n)$ time.

Que

In worst case, the number of comparisons needed to search a SLL of length n for a given element is -

(A) $\log_2 n$ (C) $\log_2 n - 1$

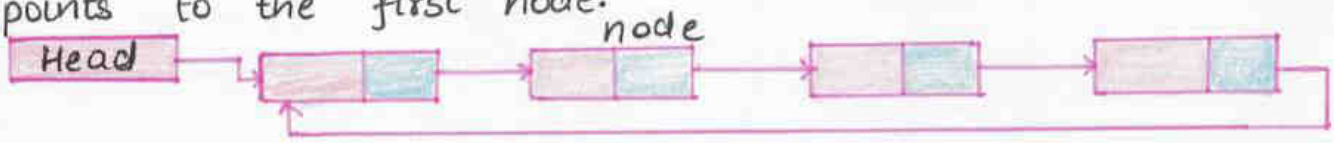
(B) $n/2$ (D) n

(D) option

CIRCULAR SINGLY LINKED LIST

Circular Linked List is a variation of Linked list in which the first element points to the last element and last element points to the first ele

In singly linked list, the next pointer of the last node points to the first node.



The circular singly linked list has no begnning and no ending & there is no null value present in the next part of any of the nodes.

These are mostly used in task maintenance in OS.

Insertion, deletion, search and traversing are done like singly linked list.

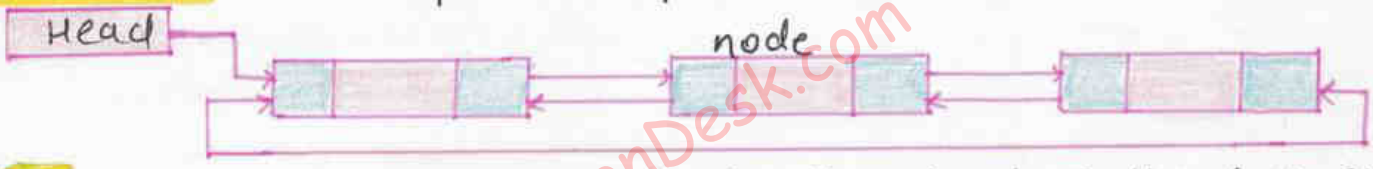
CIRCULAR DOUBLY LL.

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node.

Circular doubly linked list does not contain NULL in node.

The last node of the list contains the address of the first node of the list.

The first node of the list also contain address of the last node in its previous pointer.



It contains three parts in its structure therefore, it demands more space per node and more expensive basic op.

It provides easy manipulation of the pointers and the searching becomes twice as efficient.

Insertion, deletion, search and traversing are done like doubly linked list.

Operation	Time complexity	Space complexity
Insert	O(1) or O(n)	O(1)
Delete	O(1)	O(1)

The insertion operations that do not require traversal have the time complexity of O(1) & an insertion that requires traversal has a time complexity of O(n).

Que The following C function takes a simply-Linked list as i/p argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank.

```
typedef struct node {
    int value; struct node* next;
    } node;
Node *move_front (Node *head){
    Node *p, *q;
if ((head == NULL: || (head->next
  == NULL)) return head;
   q = NULL;  p = head;
  while (p  next != NULL) {
     q = p;
     p =  p->next;
  }
  ____________
  return head; }
```

Choose the correct alternative to replace blank line.

(A) q = NULL; p->next = head; head = p;
(B) q->next = NULL; head = p; p->next = head;
(C) head = p; p->next = q; q->next = NULL;
(D) q->next = NULL; p->next = head; head = p;

option [D].

Que The following C function takes a single linked list of integers as a parameter and rearranges the elements of the list. The function is called with the list containing the integers 1,2,3,4,5,6,7 in the given order. What will be the contents of the list after the function completes execution?

```
struct node {
    int value; struct node * next;};
void rearrange (struct node *list)
 { struct node *p, *q;
    int temp;
 if (!list || !list next)
  p= list; q= list next;
  while (q) {
    temp = p -> value;
   p -> value = q -> value;
   q -> value = temp;
   p = q -> next;
   q = p ? p -> next : 0;
  } }
```

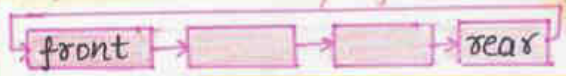
(A) 1,2,3,4,5,6,7 (C) 1,3,2,5,4,7,6
(B) 2,1,4,3,6,5,7 (D) 2,3,4,5,6,7,1

(B) 2,1,4,3,6,5,7

Solution → Given linked list is 1→2→3→4→5→6→7. If we carefully observe the given function, it just swaps the adjacent values for every pair till it reaches the end.

The modified linked list is 2→1→4→3→6→5→7

Que A circularly linked list is used to represent a Queue. A single variable p is used to access the queue. To which node should p point such that both the operations enQueue & dequeue can be performed in const time.



p → ? (A) rear node
(B) front node (C) not possible
(D) node next to front

[A] rear node

STACK

A stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle.

Stack has one end & It is ADT (Abstract-data type).

It contains only one pointer top pointer pointing to the topmost element of the stack.

A stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

It is called stack because it behaves like a real-world stack, piles of books etc.

A stack is an abstract data type with a predefined capacity, which means that it can store the elements of it.

It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO & FILO.

Standard STACK operations The following are some common operations implemented on the stack:

push() When we insert an element in a stack then the operation is known as a push.

If the stack is full then the overflow conditions occurs.

pop() When we delete an element from the stack, the operation is known as a pop.

If the stack is empty means that no element exists in the stack, this state is known as underflow state.

isEmpty() It determines whether the stack is empty or not.

isFull() It determines whether the stack is full or not.

Peek() It returns the element at the given position.

Count() It counts the total number of elements available in a stack.

change() It changes the element at the given position.

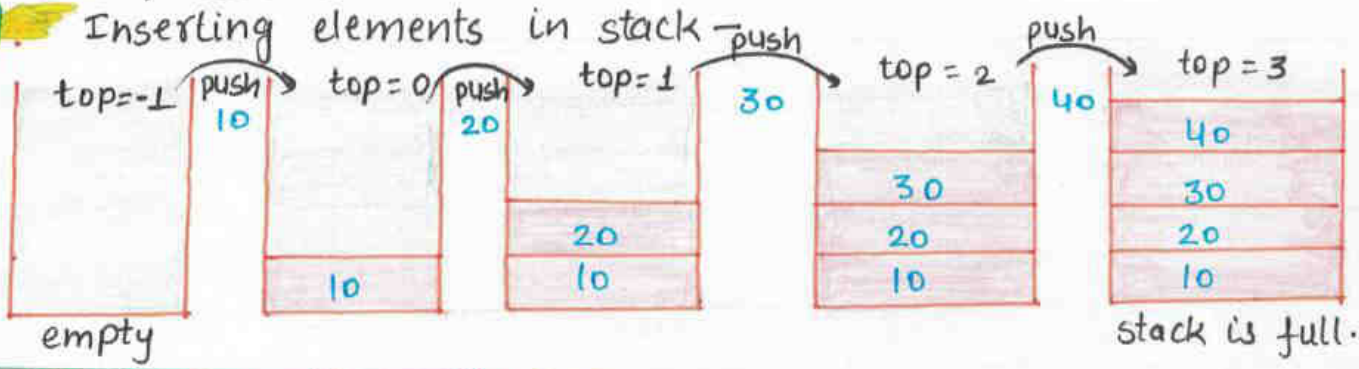
display() It prints all the elements available in the stack.

A stack can be implemented by means of Array, structure, pointer & linked list.

PUSH() Operation

The steps involved in the push operation is given below:

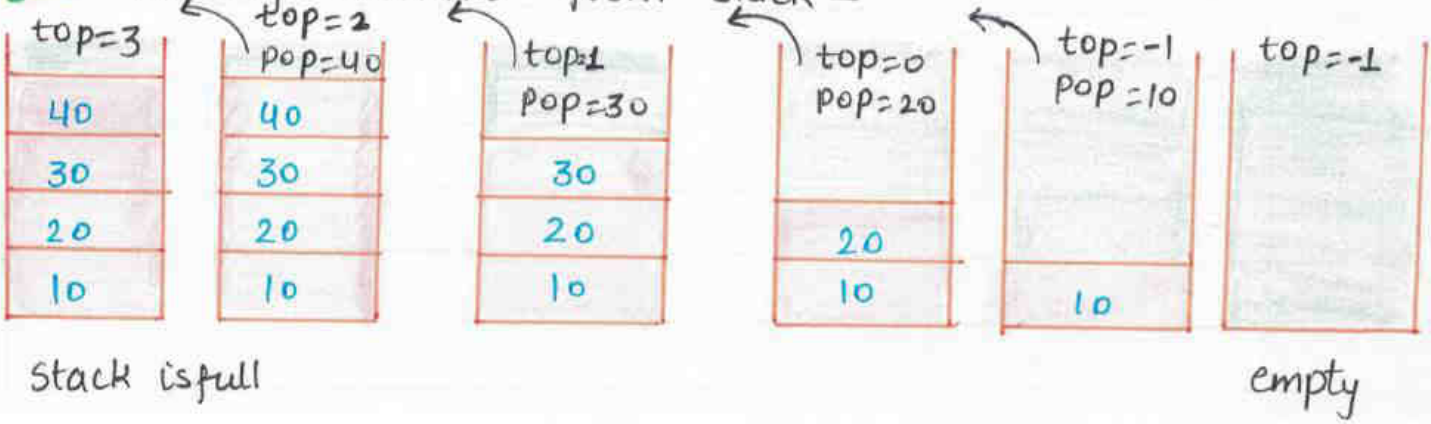
- Before inserting an element in a stack, we check whether the stack is full or not. If we try to insert the element in a full stack then the overflow condition occurs.
- When we initialize a stack, we set the value of top as '-1' to check that stack is empty.
- When the new element is pushed in a stack, first, the value of top gets incremented i.e. 'top = top + 1', and the element will be placed at the new position of the top.
- The elements will be inserted until we reach the max size of the stack.
- Inserting elements in stack –



POP() Operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether stack is empty or not. If we try to delete the element from the empty stack, then the underflow occurs.
- If the stack is not empty, we first access the element which is pointed by the top.
- Once the pop operation is performed, the top is decremented by '1' i.e. 'top = top-1'.
- Deleting elements from stack –



Applications of Stack

The following are the applications of the stack:

Balancing of Symbols Stack is used for balancing of symbols. For example: int main () { cout << "Hello"; }

As we know, each program has an opening and closing braces; when the opening braces come, we push the brace in a stack & when closing braces appears, we pop the opening braces from the stack.

The net value comes out to be zero.

String Reversal Stack is also used for reversing a string.

For reversing a string first we push all the characters of the string in a stack until we reach the null character.

After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

UNDO/REDO It can also be used for performing UNDO/REDO operations.

Eg. we have an editor in which we write a then b & then c; therefore, the text written in an editor is abc. So, there are three states a, ab & abc. which are stored in a stack. There would be two stacks in which one stack shows UNDO state, & the other shows REDO state.

Recursion The recursion means that the function is calling itself again.

To maintain previous states, the compiler creates a system stack in which all the previous records of function are maintained.

DFS (Depth First Search) This search is implemented on a Graph and Graph uses the stack data structure.

Backtracking Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, & we realize that we come on the wrong way.

In order to create a new path & coming at beginning we have to use stack data structure.

Expression Conversion

Stack can also be used for Expression conversion, & this is most important application.

The list of the expression conversion is given below-

- Infix to prefix
- Infix to postfix
- Prefix to Infix
- Prefix to postfix
- Postfix to Infix

Memory Management

The stack manages memory.

The memory is known as stack memory as all the variables are assigned in a function call stack memory.

The memory size assigned to the program is known to the compiler.

When the function is created, all its variables are assigned in the stack memory, when the function completed its execution all the variables assigned in the stack are released.

Implementation of Stack

There are two ways to implement a stack:

- Using array
- Using Linked list

Stack using array

In array implementation, the stack is formed by using the array.

All the operations regarding the stack are performed using arrays.

1. For adding the element we use push operation & element is inserted at the position of incremented top.
2. For deleting the element we use pop operation & element deleted & top is decremented by 1.
3. Peek operation involves returning the element which is present at the top of the stack without deleting it.

Time complexity of inserting & deleting element is O(1) & returning element is O(n).

Eg: Stack using array

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
struct stack {
        int top; unsigned capacity;
    int *array; };
```

```
/* function to create a stack
struct stack* crStack (unsigned
    capacity ) {
struct Stack* stack = (struct
stack*) malloc (sizeof (struct
  stack ));
```

```
    return stack; }
/* Stack is full when top is
equal to the last index */
int isFull (struct Stack* stack){
 returnstack->top == stack
          ->capacity -1; }
/* Stack is empty when top is
   equal to -1 */
int isEmpty ( struct Stack* stack){
   return stack top == -1; }
/* Function to add an item */
void push (struct Stack* stack,
          int item) {
   if (isFull (stack))
       return ;
   stack->array [++stack->top] = item;
   printf ("%d pushed to stack \n",
                item); }
/* Function to remove an item*/
int pop ( struct stack * stack) {
    if (is Empty (stack) )
      return INT-MIN;
return stack -> array[stack->top--];}
```

```
/* Function to return top */
int peek (struct stack* stack) {
   if (isEmpty (stack))
      return INT_MIN;
returnstack array [stack top]; }
/* Driver program to test
  above functions */
int main() {
 struct Stack* stack = crstack( );
   push (stack, 10);
   push (stack, 20);
   push ( stack , 30);
   printf ("%d popped from
     stack \n", pop (stack));
   return 0; }
```

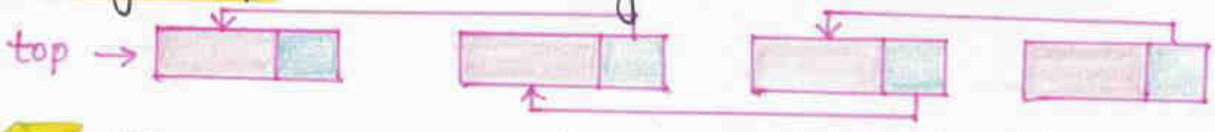
```
O/P:
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 popped from stack
Top element is : 20
Elements present: 20 10
```

- Stack using array is easy to implement & memory is saved as pointers are not involved.
- It is not dynamic & it does not grow and shrink depending on needs at runtime.

Stack using Linked List

- We can implement stack using linked list.
- Linked list allocates memory dynamically, then time complexity in array & linked list will be same for all operations i.e. push, pop & peek.
- In this nodes are maintained non contiguously in the memory.
- Each node contains a pointer to its immediate successor node in the stack.

Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



The top most node in the stack always contains null in its address field.

For adding element to stack push operation is reffered and following steps are involved:

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list.
3. If there are some nodes in the list already, then we have to add the new element in the begnning of the list.

Deleting a node from the top of stack is referred to as pop operation & following steps are involved:

1. First we have to check underflow condition, if the stack is empty the head pointer of the list points to null.
2. In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted & node must be freed.

Eg:- stack using linked list.

```
# include <limits.h>
# include <stdio.h>
# include <stdlib.h>
Struct stacknode {
    int data; struct StackNode*
    next; };
Struct StackNode* newNode
    ( int data ) {
 Struct StackNode* stackNode =
    (struct stackNode*) malloc
    (sizeof (struct StackNode))
 stackNode -> data = data;
 stackNode -> next = NULL;
    return stackNode; }
int isEmpty ( struct StackNode*
                root )
 { return !root; }
void push ( struct StackNode**
      root, int data ) {
 struct StackNode* stackNode
    = newNode(data);
 stackNode -> next = *root;
 *root = stackNode;
 printf (" %d pushed to
    stack\n", data);
 }
```

```
int pop(struct StackNode** root){
    if (isEmpty (*root))
        return INT_MIN;
    struct StackNode* temp = *root;
    *root = (*root) next;
    int popped = temp data;
    free (temp);
    return popped; }
int peek ( struct StackNode *root)
{    if (isEmpty (root))
    return INT_MIN;
    return root data; }
int main ()
{ struct StackNode *root = NULL;
    push (&root, 10);
    push (&root, 20);
    push (&root, 30);
    printf ("%d popped from
    stack \n", pop (&root));
    printf (" Top element is
    %d\n", peek (root));
    return 0;
}
```

O/P:
10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popeed from stack
Top element is : 20
Elements present in stack : 20,10

- The linked list implementation of a stack can grow and shrink according to the needs at runtime.
- Requires extra memory due to involvement of pointers.

Expression Conversion

- The way to write arithm-etic expression is known as a Notation.
- An arithmetic expression can be written in three dif-ferent but equivalent notations, i.e. without changing the essence or output of an expression. These notations are-
  - Infix Notation
  - Prefix (Polish) Notation
  - Postfix (Reverse-Polish) Notation

Infix Notation

- We write expression in infix notation, e.g. a-b+c; where operators are used in-between operands.
- It is easy for us humans to read, write and speak in infix notation but the same does not go well with computing devices.
- An algo. to process infix notation could be difficult & costly in terms of time & space consumption.

Prefix Notation Operator is prefixed to operands, i.e. operator is written ahead of operands.

Eg: +ab. This is equivalent to its infix notation a+b.

Prefix notation is also known as Polish Notation.

Postfix Notation This notation style is known as Reversed Polish Notation.

In this notation style, the operator is postfixed to the operands i.e. the operator is written after the op-erands, 'ab+' is equivalent to its infix notation a+b.

Conversion of Infix to Postfix We use the stack data structure for the conversion of infix to postfix expression.

Whenever an operator will encounter, we push ope-rator into the stack.

If we encounter an operand, then we append the operand to the expression.

Rules for conversion from infix to postfix expression:-

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthe-sis on top, push the incoming operator on to the stack.
3. If the symbol is '(', push it on the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precendance than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedance than the top of the stack, pop & print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedance with the top of the stack then use the associativity rules.
8. At the end of the expression, pop and print all the operators of the stack.

Eg: Infix expression: $(A/(B-C)^K D+E) \rightarrow ABC-/D^*E+$

sym.	stack	Output	sym.	stack	Output
(	(	-	)	(/	ABC-
A	(	A	*	(*	ABC-/
/	(/	A	D	(*	ABC-/D
B	(/(	AB	+	(+	ABC-/D*
-	(/(-	AB	E	(+	ABC-/D*E
C	(/(-	ABC	)	Empty	ABC-/D*E+

Conversion of Infix to Prefix

We use stack data structure for the conversion of infix to prefix expression:-

In order to parsing expression, we need to take care of two things i.e; Operator precedence and Associativity.

Operator precedence means the precedence of any operator over another operator.

Eg. $A + B^*C \longrightarrow A + (B^*C)$

Operators	Symbols	Operators	Symbols
Parenthesis	{}, (), []	Multi. & Division	*, /
Exponential notation	^	Addition & Subtraction	+, -

Associativity means when the operators are having same precedence, so they are evaluated with the help of associativity.

Operators	Associativity	Operators	Associativity
^	Right to left	+, -	Left to right
*, /	Left to right	++, --	Right to Left

Rules for the conversion of infix to prefix expression:

1. First, reverse the infix expresion.
2. Scan the expression from left to right.
3. If the operator arrives & the stack is found to be empty then simple push the operator into stack.
4. If the incoming operator has higher precedence than the Top of the stack, & having same precedence then push the incoming operator into the stack.
5. If the incoming operator has lower precedence than Top of the stack, pop & print the top of the stack. Test again & pop until the lower or same precedence opr found.

6. When we reach the end of expression, pop & print all the operators from the top of the stack.

7. If the operator is ')', then push it into the stack.

8. If the operator is '(', then pop all the operators from stack till it finds opening bracket in the stack.

9. If the top of the stack is ')', push the operator on stack.

10. At the end, reverse the output.

Eg: Infix exp. → (P+(Q*R)/(S-T)) ⇒ Prefix Exp → +P/*QR-ST

Sym.	Stack	Output	Sym.	Stack	Output
)	))	-	R	)/)	R-ST
T	))	T	*	)/)*	R-ST
-	))-	T	Q	)/)*	QR-ST
S	))-	ST	(	)/	*QR-ST
(	)	-ST	+	)+	/*QR-ST
/	)/	-ST	P	)+	P/*QR-ST
)	)/)	-ST	(	Empty	+P/*QR-ST

Conversion of Prefix to Postfix

prefix expression is defined as an expression in which all the operators precede operands.

Evaluation of prefix expression →

Eg: Expression - 5, 8, 16, /, 2, 2, *, -, + ⇒ final result - 7

Symbol	Stack	Symbol	Stack	Symbol	Stack
5	5	/	5,2	*	5,2,4
8	5,8	2	5,2,2	-	5,2
16	5,8,16	2	5,2,2,2	+	7

postfix expression is defined as an expression in which all the operators are present after the operands.

Evaluation of postfix expresion →

Eg: Expression - 5, 6, 2, +, *, 12, 4, /, - ⇒ final result - 37

Symbol	Stack	Symbol	Stack	Symbol	Stack
5	5	+	5,8	4	40,12,4
6	5,6	*	40	/	40,3
2	5,6,2	12	40,12	-	37

Rules for prefix to postfix conversion:-

1. Reverse, the expression from left to right.
2. If the incoming symbol is an operand then push into the stack & if it is an operator then pop two operan-ds from the stack.
3. Once the operands are popped out from the stack, we add the incoming symbol after the operands.
4. When the operator is added after the operands, then the expression is pushed back into the stack.
5. Once the whole expression is scanned, pop & print the po-stfix expression from the stack.

Eg. Prefix exp. → *+AB-CD ⇒ postfix exp : AB+CD*-

prefix to Infix : (A+B)*(C-D)

Infix to postfix : AB+CD-*

Implement Two Stacks using One Array

There are two approaches to implement two stacks using one array.

In First approach we will divide the array into two sub-arrays.

The array will be divided in two equal parts, first the sub-array would be considered stack 1 & another sub-arr-ay would be considered stack 2.

Eg. An array of 8 elements would be divided into two parts.

push1() Stack 1 Stack 2 push2()

pop1() pop2()

If the size of array is odd like size of an array is 9 then the left subarray would be of 4 size & the right sub-array would be of 5 size.

Main disadvantage of this approach is stack overflow co-ndition occurs even if there is a space in the array.

In Second approach, we are having a single array named as 'a'. In this case, stack 1 starts from 0 while stack 2 starts from n-1.

Both the stacks strat from the extreme corners.

Stack 1 starts from leftmost corner & stack 2 starts from right most corner & extends in left direction & stack 1 extends in right direction.

In this case stack overflow conditions occurs only when top+1 = top 2.

This approach provides a space-efficient implementation means that when the array is full, then only it will show the overflow error.

Eg: 2 stacks using one array

```
# include<stdio.h>
# define SIZE 10
  int ar[SIZE], top= -1 , top2 = SIZE;
/* functions to push data */
void push_stack1 (int data)
 {   if (top1 < top2 - 1)
       ar [++ top1] = data;
     else
     printf ("Stack full!"); }
void push_stack 2 (int data)
  {    if (top 1< top2-1)
       ar[- - top2 ] = data ;
       else
    printf(" Stack Full!"); }
/* Functions to pop data */
void pop-stack1 ( )
  {    if (top1 >=0)
    { int popped_value = ar[top1--];
    printf (" %d popped from stack
        1\n", popped_value); }
    else
    printf (" stack Empty ! ");
    }
void pop_stack 2 ( )
   {  if (top 2< SIZE)
     {int popped_value = ar[top2 ++];
    printf(" %d popped from stack2
         \n", popped_value); }
      else
    printf("stack Empty ! "); }
/* Functions to print stacks */
void print_stack 1 ( )
   {   int i;
     for(i= top 1 ; i>=0; --i )
     { printf [" %d ", ar[i] ); }
      printf ("\n"); }
void print_stack 2 ( )
   {    int i;
     for (i= top2 ; i< SIZE; ++i )
     { printf ("%d", ar[i]); }
    printf ("\n") ;   }
int main ( )
 {   int ar[SIZE], i , num ;
  printf ["We can push a total
       of 5 values \n");
/* No. of elements in stack1: 3 */
/* No. of elements in stack 2: 2*/
```

```
for (i=1; i<=3; ++i)
 { push_stack1 (i);
   printf (" value pushed in stack1
      %d\n", i); }
for (i = 1; i<= 2; ++i)
{  push_stack2 ( );
   printf ("value pushed in stack2
      %d\n", i); }
 /* Print both stacks */
 print_stack1 ( );
 print_stack2 ( );
 /* pushing on stack full */
 printf (" Pushing value in stack 1
      is %d\n", 5);
 push_stack 1 (5);
 /* Popping all elements fronst1
    num = top1 + 1;
```

```
while (num)
  { pop_stack1 ( );
    -- num;  }
 /* Trying to pop from empty
    stack */
  pop_stack1 ( );
        0; }
```

o/p: We can push total 5 values.
value pushed in stack1 1
value pushed in stack1 2
value pushed in stack1 3
value pushed in stack 2 1
value pushed in stack 2 2
3 2 1
2 1
pushing value in stack1 is 5
stack Full!
3 is popped from stack 1
2 is popped from stack 1
1 is popped from stack 1
stack Empty!

Reverse a Stack Using Recursion

- We can reverse a stack using recursion method.
- The most common way of reversing a stack is to use an auxiliary stack, in which following steps are followed.

1. First, we will pop all the elements from the stack & push them into auxiliary stack.
2. Once all elements are pushed back, then it contains the elements in the reverse order & we simply print them.
3. When we print stack, reverse stack will find.

- Recursion means calling the function itself again & again is used to reverse the stack.
- In recursion method following steps are followed:

1. First, we pop all the elements from the input stack & push all the popped items into the function call stack until the stack becomes empty.

2. When the stack becomes empty, all the items will be pushed at the stack.

Eg. Program to reverse a stack using Recursion

```
#include <stdio.h>
#define MAXSIZE 10
# define TRUE 1
# define FALSE 0
/* Defining the structure of stack type */
Struct Stack { int top; int array[MAXSIZE]; }st;
/* Initialization of top variable */
Void initialize () {
   st.top = -1; }
/* Checking whether the stack is full or not */
int isFull () {
   if (st.top >= MAXSIZE-1)
      return TRUE;
   else
      return FALSE ; }
/* checking stack is empty or not
int isEmpty ( ) {
   if (st.top = = -1)
      return TRUE;
   else
      return FALSE; }
/* Function to push the element into the stack */
void push ( int num) {
   if (isFull ( ))
      printf ("stack is Full!");
   else {
      st.array[st.top+1]=num;
      st.top++;
   /* Function to pop element */
int pop() {
   if (isEmpty ())
   printf ("Stack is Empty!");
   else {
      st.top = st.top-1;
      return st.array [st.top +1]; }}
void printStack () {
/* Cond. to check Stack is Empty */
   if (! isEmpty ())
      int temp = pop();
      printStack ();
      printf ("%d\n", temp);
      push (temp); }}
/* Function to insert the element at the bottom of stack */
void insert At Bottom (int item){
   if (is Empty ()) {
      push (item); }
   else {
      int top = pop();
      insertAt Bottom (item);
      push (top); }}
```

```
/* function to reverse the order
  of stack */
void reverse() {
   if (isEmpty()) {
     int top = pop();
      reverse();
      insertAtBottom(top); }}
 int getsize() {
 return st.top+1;   }
 int main() {
    initialize(st);
    push(1); push(2); push(3);
      push(4);
      push(5);
 printf("original stack:");
 printstack();
 reverse();
 printf("\n Reversed stack:");
  printstack();
   return 0;   }
```

```
O/P: original stack:
     1 2 3 4 5
     Reversed stack:
     5 4 3 2 1
```

CALL STACK

A Call Stack is a data structure that stores information about the active subroutines of a computer program.

This kind of stack is also known as execution stack, program stack, control stack, run time stack or machine stack.

A call stack is used for having one is to keep track of the point to which each active subroutine should return co-ntrol when it finishes executing.

An active subroutine is one that has been called, but is yet to complete execution, after which control should be hand-ed back to the point of call.

Call stack is used everywhere, when we design compilers.

Eg:
```
factorial (n)
   if n=1
   return 1
   else
   return n* factorial (n-1)
```

factorial (5)
⇓ 120
5 x factorial (4) ⇒ 5x24=120
⇓24
4x factorial (3) ⇒ 4x6 = 24
⇓6
3x factorial (2) ⇒ 3x2 = 6
⇓2
2x factorial (1) ⇒ 2x1 = 2
⇓
1

we use 'n' frame in fac-torial of n (n!) Then space complexity will be ⇒
S(n) = O(n)

The best data structure to check whether an arithmetic has balanced parenthesis is

(A) Queue (C) Tree
(B) Stack (D) List

[B]. stack

Stack is best data structure that uses push & pop operations

Assume that the operators +, -, * are left associative and ^ is right associative. The order of precedence (from highest to lower) is ^, *, +, -. The postfix expression corresponding to the infix expression - a+b*c-d^e^f is -

(A) abc*+def^^-
(B) abc*+de^f^-
(C) ab+c*d-e^f^
(D) -+a*bc^^def

Solution: Infix - a+b*c-d^e^f

⇒ (a+(b*c))-(d^(e^f))

⇒ a bc* + def^^ -

⇒ abc*+def^^-

[A]. abc*+def^^-

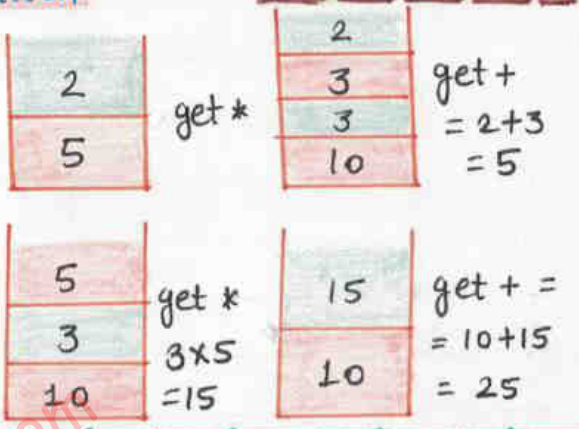
```
# include <stdio.h>
# define EOF -1
void push (int );
 int pop (void );
void flagError ( );
 int main ( ) {
int c, m, n, r;
while ((c = getchar( )) != EOF )
  { if ( isdigit( ))
      push (c);
elseif ((c == '+')||(c == '*' )
 {    m = pop( );
      n = pop( );
   r = (c == '+' )? n+m : n*m;
    push(r); }
elseif (c != ' ')
   flagError ( );
 printf ("%d", pop( )); }
```

What is output of the program for the following input -
5 2 * 3 3 2 + * +

(A) 15 (C) 30
(B) 25 (D) 150

Solution

op: **[B]. 25**



Which of the following is essential for converting an infix expression to the postfix form efficiently -

(A) An operator stack
(B) An operand stack
(C) An operand stack and an operator stack
(D) A parse tree

[A]. An operator stack

Infix to postfix exp. → Stack

A function f is defined on stacks of integers satisfies the following property- $f(\Phi) = 0$ & $f(push(s,i)) = max(f(s), 0) + i$ for all stacks s & integers i.

If a stack s contains the integers 2, -3, 2, -1, 2 in order from bottom to top. What is f(s) ?

(A) 6 (B) 4 (C) 3 (D) 2

[C] option - 3

f(s)	i	-1	2
0	2	2	-1
2	-3	1	2

$f(push(s,i)) = max(f(s), 0) + i$

$= max(0,0) + 0 = 0$

Following sequence of operations is performed on stack push(10). push(20). pop. push(10). push(20) pop pop pop push(20). pop. sequence of popped elements.

(A) 20,10,20,10,20 (B) 10,20,20,10,20

(C) 20,20,10,10,20 (D) 20,20,10,20,10

2/0
2/0
1/0
2/0
1/0

→ popped elements 20,20,10,10,20

[C] - 20,20,10,10,20

QUEUE

Queue is an ADT (Abstract data type) and a linear structure which is mostly similar to stacks.

The difference between queue & stack is, a queue is open at both its ends.

It is defined as an ordered list enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.

Queue follows FIFO (First-In-First-Out) methodology, i.e. the data item stored first will be accessed first.

Eg: A real-world example of queue is a single-lane one-way road, where the vehicle enters first, exit first.

Queue representation :-

Applications Of Queues

There are various applications of queues discussed as below:-

1. Queue are widely used as waiting lists for a single shared resource like printer disk, CPU.

2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg.. pipes, file IO, sockets.

3. Queues are used as buffers in most of the applications like MP3 media player, CD player etc.

4. Queues are used to maintain the play list in media pl--ayers in order to add and remove the songs from the playlist.

5. Queues are used in Operating System for handling interrupts.

Time complexity of accessing & searching elements in average case will be 'θ(n)' and time complexity of insertion & deletion will be 'θ(1)'.

Time complexity of searching & accessing elements in worst case will be 'O(n)' and of insertion & deletion will be 'O(1)'.

Space complexity in worst case will be 'O(n)'.

Operations Performed on Queue

Following operations are performed on queues.

Enqueue

The enqueue operation is used to insert the element at the rear end of the queue & returns void.

Below are the steps to enqueue data in a queue -

1. Check whether the queue is full or not.
2. If the queue is full then print the overflow error & exit the program. & If queue is not full then increment the rear pointer to point to the next empty space.
3. Add the element in the position pointed by the Rear.
4. After adding the elements return success.

Dequeue

It performs the deletion from the front-end of the queue.

It returns the element which has been removed from the front-end & also return an integer value.

Below are the steps to perform dequeue operation -

1. Check whether the queue is empty or not.
2. If queue is empty then print the underflow & exit the program. & if queue is not empty then access the data where the front is pointing.

3. Increment front pointer to point to the next available data element & returns success.

Peek This returns the element, which is pointed by the front pointer in the queue but does not delete it.

Queue Overflow (isFull) It shows the overflow condition when the queue is completely full.

Queue Underflow (isEmpty) It shows the underflow condition when the queue is empty i.e. no elements are in the queue.

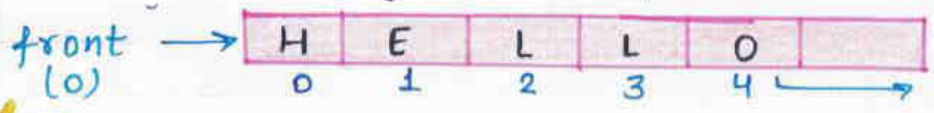
Implementation of Queue There are two ways of Implementing the Queue:

Implementation Using array We can easily represent queue by using linear arrays.

There are two variables i.e. front and rear, that are implemented in case of every queue.

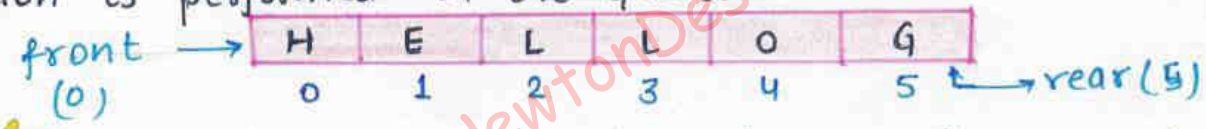
Front & Rear variables point to the position from where insertions and deletions are performed in a queue.

The value of front and queue is -1 which shows empty queue.



The above figure shows the queue of characters forming the word "HELLO". No deletion is performed in the queue till now, therefore the value of front remains -1.

The value of rear increases by one every time an insertion is performed in the queue.



After inserting one element rear increase by one.

After deleting an element, the value of front increase from -1 to 0. then queue will look like this -

Eg. Function to insert element

```
void insert (int queue[ ], int
max, int front, int rear,
int item ) {
if (rear + 1 = = max)
printf ("overflow");
else {
if (front = = -1 && rear = = -1){
```

```
    front = 0;
    rear = 0; }
else
    rear = rear+1;
queue [rear ]= item;
}}
```

For inserting an element check queue is full then overflow returns.

If the item is to be inserted as the first element in the list, then set the value of front and rear to 0 & insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

For deleting the element, check the value of front is -1 or value of front is greater than rear then write an underflow message and exit.

Otherwise keep increasing the value of front and return the item stored at the front of the queue.

Eg. function to delete element

```
int delete(int queue[ ], int max,
    int front , int rear){
      int y;
  if (front == -1 || front > rear)
      printf("underflow");
  else {
      y = queue [front];
  if (front = = rear ){
      rear = front = -1;
  else
      front = front +1; }
  return y;
}}
```

Drawbacks

The technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end & the value of front might be so high so, that more memory wasted.

0-4 all elements deleted.

The queue can be extended at run time but it is time taking process & impossible at runtime then a lot reallocations takes place.

Implementation Using Linked List

The array implementation cannot be used for the large scale applications so, we use linked list.

The storage requirement of linked representation of a queue with n elements is O(n) while the time complexity of operations is O(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and link part and each element of the queue points to its immediate next element in the memory.

There are two pointers maintained in the memory ie. front pointer and rear pointer.

The front pointer contains the address of the starting element and the rear pointer contains the address of last element of the queue.

If front and rear both are NULL, it indicates empty queue.

The insertion operation append the queue by adding an element to the end of the queue & the new element will be last element of the queue.

Deletion operation removes the element that is first inserted among all the queue elements.

Eg: Function to insert element

```
void insert(struct node *ptr, int item)
{ ptr = (struct node*) malloc (sizeof (struct node));
  if (ptr == NULL)
    printf ("overflow");
  else {
    ptr -> data = item;
    if (front == NULL) {
    front = ptr; rear = ptr;
    front -> next = NULL;
    rear -> next = NULL; }
  else {
    rear -> next = ptr;
    rear = ptr;
    rear -> next; } }}
```

Eg: Function to delete element

```
void delete (struct node * ptr) {
    if (front == NULL)
     printf ("underflow");
    else {
       ptr = front;
      front = front    next
      free (ptr);
      }}
```

Types of Queues

There are four different types of queue that are mentioned as follows-

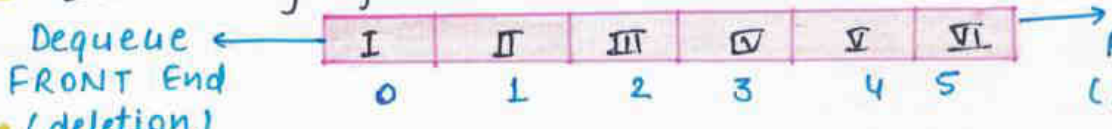
- Linear Queue
- Circular Queue
- Priority Queue
- Double ended Queue (Deque)

Linear Queue

The linear queue is also known as simple queue.

In linear queue, an insertion takes place from rear end & deletion takes place from front end.

It strictly follows FIFO rule.



The major drawback of using a linear queue is that insertion is done only from the rear end.

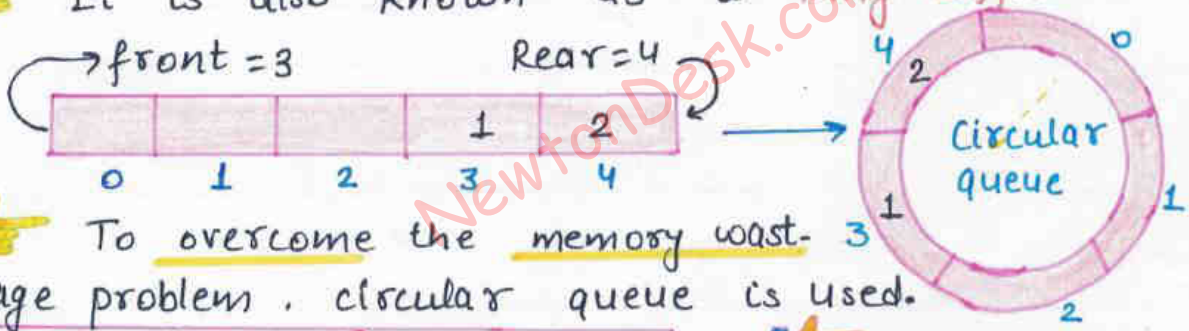
If the first elements are deleted from the queue then we cannot insert more elements then it shows overflow condition as the rear is pointing to the last element.

Circular Queue

A circular queue is similar to a linear queue as it is also based on FIFO rule.

The difference between linear queue and circular queue is that in circular queue the last position is connected to the first position that form a circle.

It is also known as a Ring buffer.



To overcome the memory wastage problem, circular queue is used.

Operations Performed

Following are the operations that can be performed on a circular queue.

- **Front :** It is used to get the front element from the Queue.
- **Rear :** It is used to get the rear element from the queue.

- **enQueue(value)**: This function is used to insert the new value in the queue, new value is always inserted from rear end.
- **deQueue()**: This function deletes an element from the queue. The deletion in a Queue always takes place from front end.

The complexity of the enqueue and dequeue operations of a circular queue is O(1) for array implementations.

Applications

The circular queue provides memory management. The memory is managed efficiently by placing the elements in a location which is unused.

The operating system also uses the circular queue to insert the processes and then execute them.

In a Computer-control traffic system, traffic light is one of the best examples of circular queue.

Enqueue Operation

The steps of enqueue operation are given below:

1. First, we will check whether queue is full or not.
2. Initially the front and rear are set to -1. When we insert the first element, front & rear both are set to 0
3. When we insert a new element, the rear gets incremented, i.e. rear = rear+1.

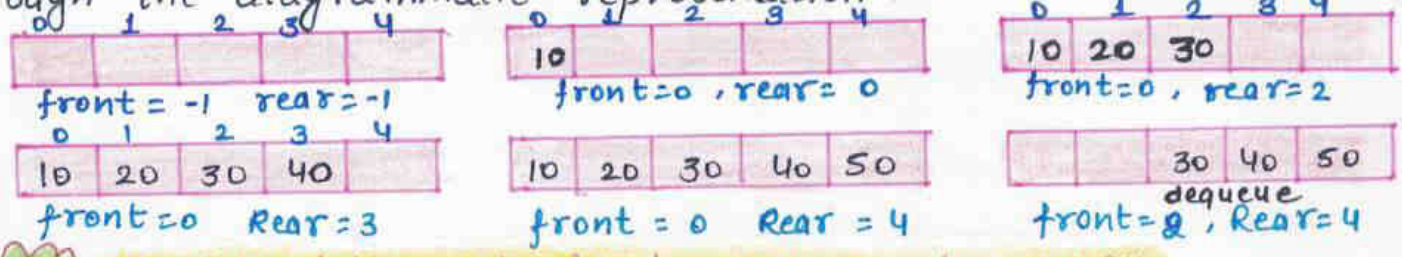
When front == 0 && rear = max-1, which means front is at the first position & rear is at last position of the queue & front == rear+1; in both cases we can't insert the element in the queue.

Dequeue Operation

The steps of dequeue operation are given below:

1. First check queue is empty or not. If queue is empty we can't perform the dequeue operation.
2. When the element is deleted, the value of front gets decremented by 1.
3. If there is only one element left which is to be deleted, then the front and rear are reset to -1.
4. Check if (front == rear) if it is true then set front = rear = -1 else check if (front == size-1), if it is true then set front = 0 & return the element.

Lets understand the enqueue & dequeue operation through the diagrammatic representation.



Eg: Implementation of circular queue using array

```
#include <stdio.h>
#define max 6
 int queue[max];
  int front = -1 , rear = -1;
/* function to insert element */
void enqueue (int element)
 if (front == -1 && rear == -1){
      front = 0;        /* check queue
      rear = 0;            is empty */
    queue [rear] = element; }
elseif ((rear+1)%max == front)
     printf ("overflow");
 else {
   rear = (rear+1)%max;
   queue [rear] = element; }}
/* function to delete element */
 int dequeue () {
 if ((front == -1) && (rear == -1))
   printf ("underflow");
elseif (front == rear) {
  printf ("\n Dequeued element is
    %d ", queue [front]);
    front = -1;
    rear = -1; }
 else {
  printf ("\n Deque ed element is
        ", queue [front]);
    front = (front+1)%max ;}}
/* function to display elements */
void display () {
  int i = front;
  if (front == -1 && rear == -1)
   printf (" Queue is empty ");
  else {
   printf ("\n Elements in Queue: ");
  while (i <= rear) {
   printf ("%d", queue[i] );
    i = (i+1) % max; }}}
 int main () {
 int choice = 1 , x;
  while(choice < 4 && choice != 0){
   printf (" Press 1: Insert ");
   printf (" Press 2: Delete ");
   printf (" Press 3: Display ");
   printf (" Enter your choice :");
   scanf (" %d ", &choice);
 Switch(choice) {
  case 1: printf (" Enter the ele-
  -ment is to be inserted ");
  scanf ("%d ", &x);
   enqueue(x);
   break;
```

```
case 2:
 dequeue ();
 break;
 case 3:
 display ();
 }}
 return 0;
 }
```

```
O/P: Press 1 : Insert
Press 2 : Delete
Press 3 : Display
Enter your choice: 1
Enter the element which is to be inserted: 10
Press 1 : Insert
Press 2 : Delete
Press 3 : Display
Enter your choice: 1
Enter the element which is to be inserted: 20
Press 1 : Insert
Press 2 : Delete
Press 3 : Display
Enter your choice: 2
The dequed element is:
10
```

We know that linked list is a linear data structure that stores two parts; we can implement circular queue using LL.

Time complexity of both enqueue & dequeue operations is O(1).

Eg: Implementation of circular queue using linked list

```
#include <stdio.h>
struct node { int data ; struct node* next ; };
struct node *front = -1;
struct node* rear = -1;
/* function to insert element */
void enqueue (int x) {
 struct node* newnode ;
 newnode = (struct node*)
  malloc ( sizeof ( struct node));
 newnode -> data = x;
 newnode -> next = 0;
 if (rear = -1) {
  front = rear = newnode ;
  rear -> next = front; }
 else {
  rear -> next = newnode;
  rear = newnode ;
  rear -> next = front ; }}
/* function to delete element*/
void dequeue () {
 struct node* temp;
 temp = front;
 if ((front = =-1) && (rear = -1))
  printf ("Queue is Empty" );
 else if(front = = rear) {
  front = rear = -1
  free (temp); }
 else {
  front = front -> next;
  rear -> next = front;
  free(temp); }}
/* function to get the front */
int peek () {
 if ((front = = -1) && (rear == -1))
  printf ("Queue is empty");
 else
  printf ("The front element is:
  %d", front -> data); }}
/* function to display all the ele-
 -ments of queue */
void display () {
 struct node * temp;
 temp = front;
```

```
printf ("\n Elements of Queue:");
  if ((front==-1 && rear==-1))
   printf("Queue is Empty");
  else   {
 while (temp->next != front){
     printf ("%d", temp->data );
        temp = temp->next; }
    printf ("%d", temp data );}}
 void  main ( ) {
```

```
enqueue ( );
enqueue ( );
enqueue ( );
display ( );
dequeue ( );
 peek ( );  }
```

O/P: The elements of Queue:
34, 10, 23
The front element is: 10

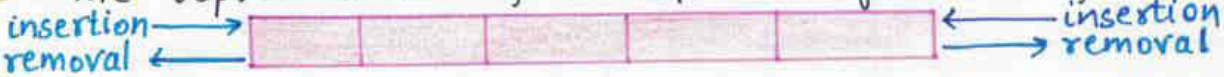
Double-Ended Queue (Deque)

The deque stands for Double Ended Queue.

Deque is a linear data structure where the insertion and deletion operations are performed from both ends.

It does not follow the FIFO rule.

The representation of a deque is given as follows -



There are two types of deque -

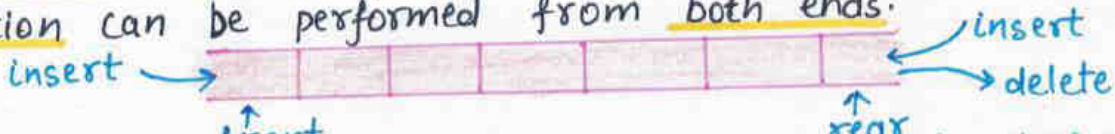
- Input restricted Queue
- Output restricted Queue

Input restricted Queue

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



There are the following operations performed on deque -

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operation through which we can get the deque's front & rear elements.

Insertion at the front end

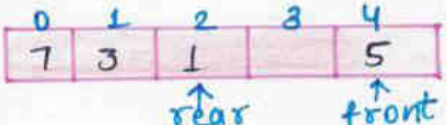
In this operation, the element is inserted from the front end of the queue.

Before implementing the operation, first we have to check whether the queue is full or not.

If the queue is not full, then element can be inserted from the front end by using below conditions-

1. If the queue is empty, both rear & front are initialized with 0. Now, both will point to the first element.
2. Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by front = n-1, i.e. the last index of array.

Reinitialized it, front = n-1 (last index)



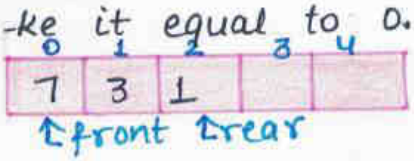
Now add the new key to a[front]

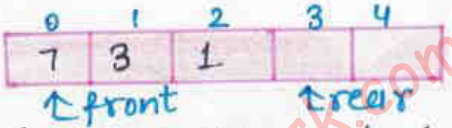
Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue.

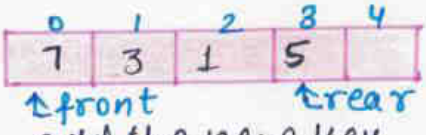
If the queue is not full then follow these steps-

1. If the queue is empty, both rear and front are initialized with 0. Now both will point to first element.
2. Otherwise, increment the rear by 1. If the rear is the last index (size-1), then instead of increasing it by 1, we have to make it equal to 0.





Increase the rear by 1.



add the new key

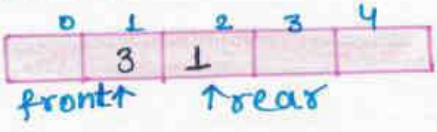
Deletion at the front end

In this operation, the element is deleted from the frond end of the queue.

If the queue is empty, front = -1, shows underflow cond.

If the queue is not full, then the element can be inserted by using below conditions -

1. If the deque has only one element, set rear = -1 & front = -1.
2. Else if front is at end (front = size-1) set front = 0.
3. Else front = front + 1.



Deletion at the rear end

In this operation, the element is deleted from rear end of the queue.

- If the queue is empty, front = -1, shows underflow.
- If the deque has only one element, set rear = front = -1
- If rear = 0 (rear is at front), then set rear = n-1 else decrement the rear by 1 (rear = rear-1).

- Check empty is performed to check whether the deque is empty or not. If front = -1, it means deque is empty.
- Check full is performed to check whether the deque is full or not. If front = rear+1 or front = 0 & rear = n-1 it means that deque is full.
- Time Complexity of all the above operations of the deque is O(1), i.e. constant.

Applications

Deque can be used as both stack & queue, as it supports both operations.

- Deque can be used as palindrome checker means that if we read string from both ends, the string would be same.

Implementation of deque

```
#include <stdio.h>
#define size 5
int deque[size];
int f = -1, r = -1;
/* Function to insert value at front */
void insert_front(int x) {
  if ((f==0 && r==size-1) || (f==r+1))
    printf("overflow");
  else if ((f==-1) && (r==-1)) {
      f = r = 0;
      deque[f] = x; }
  else if (f==0) {
      f = size-1;
      deque[f] = x; }
  else {
      f = f-1;
      deque[f] = x; } }
/* Function to insert value at rear */
void insert_rear(int x) {
  if ((f==0 && r==size-1) || (f==r+1))
    printf("overflow");
  else if ((f==-1) && (r==-1)) {
      r = 0;
      deque[r] = x; }
  else if (r==size-1) {
      r = 0;
      deque[r] = x; }
  else {
      r++;
      deque[r] = x;
  } }
```

```
/* function to print all values */
void display() {
   int i=f;
   printf("Elements in deque: ");
   while(i!=r) {
   printf("%d", deque[i]);
        i=(i+1)%size; }
    printf("%d", deque[r]); }
/* function to retrieves first value*/
void getfront() {
   if((f==-1) && (r==-1))
    printf("Deque is Empty");
   else
   printf(" The value of the element
      at front %d", deque[f]); }
/* function to retrieves the last
   value of deque */
void getrear() {
   if((f==-1) && (r==-1))
    printf("Deque is Empty");
   else
    printf(" The value of element
    at rear is %d", deque[r]);}
/* function delete front element */
void delete_front() {
    if((f==-1)&&(r==-1))
    printf("Deque is Empty");
    else if(f==r) {
    printf(" The deleted element is
      %d", deque[f]);
      f=-1;
      r=-1; }
   elseif(f==(size-1)) {
   printf(" Deleted element is:%d",
      deque[f]);
     f=0; }
    else {
   printf(" The deleted element is
      %d", deque[f]);
      f=f+1; }}
/*funtion deletes the element at rear*/
void delete_rear() {
    if((f==-1) && (r==-1))
    printf("Deque is Empty");
   else if(f==r) {
    printf(" The deleted element
      is %d", deque[r]);
      f=-1;
      r=-1; }
   elseif(r==0) {
   printf("The deleted element
     is %d", deque[r]);
     r=size-1; }
    else {
    printf(" The deleted element
     is %d", deque[r]);
      r=r-1; }}
int main() {
   insert_front(20);
   insert_front(10);
   insert_rear(30);
   insert_rear(50);
   display(); getfront();
   getrear(); delete_front();
   deleterear(); display();
   return 0; }
```

```
o/p: Elements in deque : 10,20, 30,50
value of front element: 10
value of rear element: 50
Deleted element: 10
Deleted element: 50
Elements in deque: 20, 30
```

Priority Queue

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e. the element with the highest priority comes first in priority que.

The priority of the elements in a priority queue will determine the order in which elements are removed.

The priority queue supports only comparable elements, which means that the elements are either arranged in ascending order or descending order.

Eg. Suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a queue in ascending order than 1 numer would be having highest priority and 22 having lowest priority.

An element with the higher priority will be deleted before the deletion of the lesser priority & poll() function used to delete the element.

If two elements having the same priority, they will be arranged using FIFO principe.

There are two types of priority queue-

- Ascending order priority Queue
- Descending Order priority Queue

Ascending Order Priority Queue

In ascending order priority queue, a lower priority number is given as a higher priority.

Descending Order Priority Queue

In descending order priority queue, a higher priority number is given as a higher priority.

The priority queue can be implemented in four ways- array, linked list, heap data structure and binary search tree.

The heap data structure is the most efficient way of implementing the priority queue, so we will discuss here, implementation using heap.

Analysis of complexities using different implementations:

Operation	Unordered array	Ordered array	Linked List	Binary Heap	Binary Search Tree
Insert	O(1)	O(N)	O(1)	O(nlogn)	O(logn)
Delete	O(N)	O(1)	O(n)	O(nlogn)	O(logn)
Peek	O(N)	O(1)	O(n)	O(1)	O(1)

HEAP

A heap is a tree-based DS that forms a complete binary tree, and satisfy the heap property.

If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap.

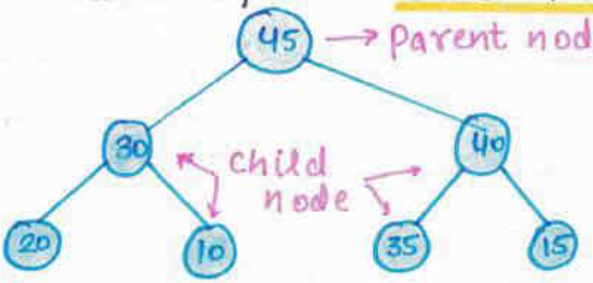
It means that the value of the parent node could be more than or equal to child node or the value of parent node could be less than or equal to the value of child node.

There fore we can say there are two types of heaps:

- Max heap
- Min heap

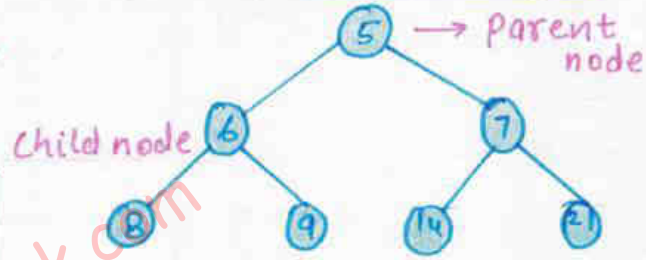
Max Heap

The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.



Min Heap

The min heap is a heap in which the value of the parent node is less than the value of child nodes.



Both the heaps are the binary heap. as each has exactly two child nodes.

Operations performed

The common operations that we can perform on a priority queue are insertion, deletion & peek.

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom & left to right.

If the element is not in correct place then it is compared with the parent node; if it is found out of order elements are swapped & continue till correct position.

As we know that in a max heap, the maximum element is the root node, when we remove, it creates empty slot.

The last inserted element will be added in this empty slot, then this element is compared with the child nodes, i.e. left & right child & swap with the smaller of the two & keep moving down the tree until the heap property is restored.

Applications

It is used in the Dijkstra's shortest path algorithm.

It is used in prim's algorithm.

It is used in data compression techniques like Huffmancod.

It is used in heap sort.

It is also used in operating system like priority scheduling, load balancing and interrupt handling.

Eg: Priority queue using binary max Heap.

```
#include <stdio.h>
#include <stdlib.h>
  int heap[40];
  int size = -1;
/* retriving the parent node of
   the child node */
 int parent (int i) {
      return (i-1)/2; }
/* retriving the left child of the
 parent node */
 int left_child ( int i) {
  return i+1; }
/* retriving the right child of
the parent node */
 int right_child ( int i){
      return i+2; }
/* returning the element having
the highest priority */
 int get_Max ( ) {
  return heap[0]; }
/* returning the element having
the minimum priority */
 int get_Min ( ) {
   return heap[size]; }
/* function to move the node up
the tree in order to restore the
 heap property. */
void moveUp (int i) {
  while (i>0){
/* swapping parent node with a
  child node */
  if (heap[parent (i)]< heap[i]){
     int temp;
     temp = heap[parent(i)];
     heap[parent (i)] = heap[i];
     heap[i] = temp; }
/* updating the value i → i/2 */
       i = i/2; }}
/* function to move the node down
the tree to restore heap property*/
void moveDown (int k) {
    int index = k;
/* getting location of leftchild*/
    int left = left_child (k);
    if (left <= size && heap[left] >
        heap[index] ) {
```

```
    index = left; }
/* getting the location of right child */
    right = right_child(k);
  if (right <= size && heap[right] >
    heap[index]) {
    index = right; }
/* If k is not equal to index */
  if (k != index) {
    int temp;
    temp = heap[index];
    heap[index] = heap[k];
    heap[k] = temp;
    moveDown(index); }}
/* removing the element of maxi-
-mum priority */
 void removeMax() {
   int r = heap[0];
     heap[0] = heap[size];
    size = size-1;
     moveDown(0); }
/* inserting the element in a prio-
-rity queue */
void insert(int p) {
   size = size+1;
   heap[size] = p;
/* moveUp to maintain heap property
   moveUp(size); }
/* removing the element from the
  priority queue at given index i */
void delete(int i) {
   heap[i] = heap[0]+1;
/* move the node stored at ith
 location is shifted to the root
 node */    moveUp();
/* removing the node having max.
 priority */
   removeMax(); }
int main() {
 /* inserting elements */
    insert(20);
    insert(19);
    insert(21);
    insert(18);
    insert(12);
    insert(17);
    insert(15);
    insert(16);
  int i=0;
 printf("Elements in priority
  queue: ");
 for(int i=0; i<=size; i++){
   printf("%d", heap[i]); }
  delete(2);
  printf("\n After deleting ele-
 -ments in priority queue:");
  for(int i=0; i<=size; i++){
   printf("%d", heap[i]); }
int max = get_Max();
  printf("\n Element having h-
 -ighest priority : %d", max);
  int min = get_Min();
  printf("\n Element having
 mininum priority: %d", min);
        0; }
```

O/P: Elements in priority queue : 21, 19, 20, 18, 12, 17, 15, 16
After deleting elements in priority queue: 21, 19, 18, 17 12 16 15

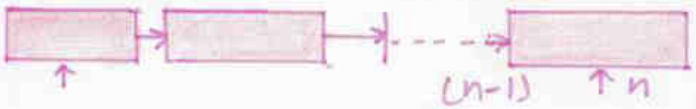
Que

A queue is implemented using a non-circular singly linked list. The queue has a head pointer and a tail pointer, as shown in the figure. Let n denote the no. of nodes in the queue. Let enqueue be implemented by inserting a new node at the head, and dequeue be implemented by deletion of a node from the tail.

Which of the following is the time complexity of the most efficient implementation of 'enqueue' and 'dequeue' respectively for this data structure.

(A) $O(1), O(1)$ (C) $O(1), O(n)$
(B) $O(n), O(1)$ (D) $O(n), O(n)$

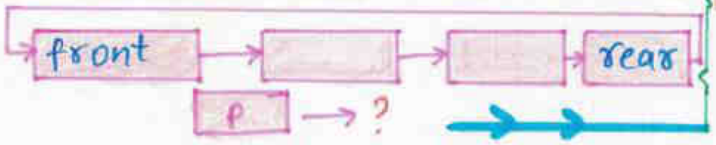
Solution: [C]. $\theta(1), \theta(n)$



for enqueue → $\theta(1)$
for dequeue → $\theta(n)$
we want next of second last so, we can not get previous of it from tail point in SLL.

Que

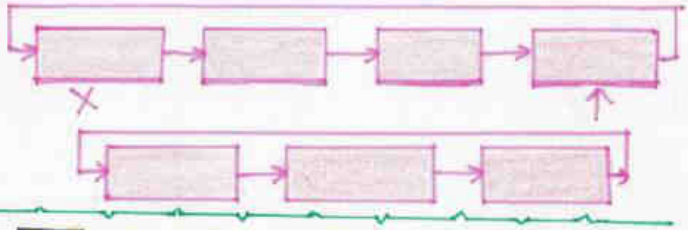
A circularly linked list is used to represent a queue. A single variable P is used to access the queue. To which node should P point such that both the operations enqueue and dequeue can be performed in constant time.



(A) rear node (B) front node
(C) not possible with a single pointer (D) next node to front

Solution: [A]. rear node

⇒ When pointer at FRONT add some at REAR: if n-element list, it will take $O(n)$ time.
⇒ When pointer points at rear - p→next = FRONT of queue
enqueue = $O(1)$; dequeue = $O(1)$



Que

Let Q denote a queue containing 16 numbers and s be an empty stack. Head(Q) returns the element at the end of the queue Q without removing it from Q. Similarly Top(s) returns the element at the top of s without removing from s. Consider the algorithm -

```
While Q is not empty do -
if S is empty OR Top(S) ≤ Head(Q)
then, x = Dequeue(Q)
      push(S, x);
else  x = pop(S);
      enqueue(Q, x);
end
end
```

The maximum possible no of iteration of the while loop in the algorithm is -

(A) 16 (B) 32 (C) 256 (D) 64

Solution: [B] 32.

Dequeue c push into stack.

16,15 - - - - - - - - - - - - 1

15,14 - - - - - - - - - - - 1

15,14 - - - - - - - - - - 1,16

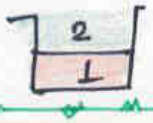
14,13 - - - - - - - - - - 1,16,15

1 , 16, 15 - - - - - - - - - 2

top(S) ≤ head(Q) always

30+1 push(16) } 2 Iteration
pop(16) }

15,14 - - - - - - - 2



31, 29, 27 - - - - -

Consider the following operation along with enque-ue & dequeue operations on queues, where k is a glo-bal parameter.

```
MultiDequeue ( ) {
    m=K;
while((Q is not empty) & (m>0))
  { Dequeue (Q);
    m=m-1; }}
```

What is the worst case time complexity of a sequence of n queue operations on an initia-empty queue.

(A) O(n) (C) O(n*k)
(B) O(n+k) (D) O(n²)

Solution: [A]. O(n)

Multiqueue is ap-plying deque till the queue is not empty. O(n)

less than equal to n elem-ents ≤ n opertions.

A queue is implemented using an array such that ENQUEUE & DEQUEUE operations are performed effic-iently. Which one of the follow-ing is correct (n refers to the number of items in the queue.

(A) Both operations can be per-formed in O(1) time.

(B) At most one operation can be performed in O(1) time but the worst time for the other operation will be Ω(n)

(C) The worst case time comple-xity for both operation will be Ω(n)

(D) Worst case time complexity for both operations will be Ω(log n)

[A].option Circular array.

Suppose a circular que-ue of capcity (n-1) elem-ents. Assume that the insertion and deletion opera-tions are carried out using REAR & FRONT as array index variables reppectively. Initially Rear = front = 0. The conditions to detect queue full and queue empty are:

(A) FULL: (rear+1) mod n == front, empty: rear == front

(B) Full: (rear+1) mod n = front
empty: (front+1) mod n == rear

(C) Rear = front
empty: (rear+1) mod n == front

(D) Full: (front+1) mod n = Rear
empty: Rear == front

[A].

TREE

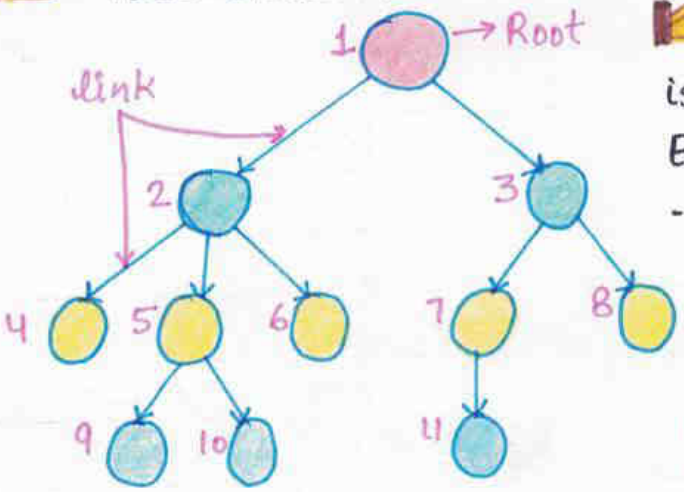
A tree is non-linear and hierarchial data structure consisting of a collection of nodes such that each node of the tree stores a value, a list of references to nodes.

It is defined as a non-linear data structure because it does not store in a sequential manner. In this elements are arranged in multiple levels.

In the tree data structure, the top most node is known as a root node. Each node contains some data & data can be of any type.

Each node contains some data & the link or reference of other nodes that can be called children nodes.

Tree structure is shown below:



In this structure, each node is labeled with some number, & Each arrow in the figure is known as a link between the two nodes.

Some terms that are used in trees are as follows:-

Root

The root node is the topmost node in the tree hierarchy, or the root node is one that doesnot have any parent.

In the above structure, node numbered 1, is the root node of the tree.

If a node is directly linked to some other node, it would be called a parent-child relationship.

Child node

If the node is a descendant of any node, then the node is known as a child node.

Parent

If the node contains any sub-node, then that node is said to be the parent of that sub-node.

Sibling ☛ The node that have same parent are kn-own as siblings.

Leaf Node ☛ The node of the tree, which does not have any child node, is called a leaf node.

☛ A leaf node is the bottom-most node of tree.

☛ There can be any number of leaf nodes present in gen-eral tree. Leaf nodes also known as external nodes.

Internal Nodes ☛ A node has atleast one child node kn-own as an internal node.

Ancestor Node ☛ An ancestor of a node is any predecess-or node on a path from the root to that node.

☛ The root node doesn't have any ancestors.

☛ In the tree shown, nodes 1, 2 & 5 are the ancestors of 10.

Descendant ☛ The immediate successor of give node is kno-wn as descendant of a node.

☛ In the figure shown 10 is the descendant of node 5.

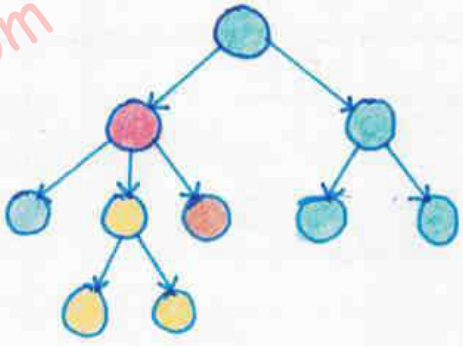
Properties ☛ The properties of tree data structure are following:-

Recursive Data Structure ☛ The tree is also known as a recursive data structure.

☛ A tree can be defined as recursively because the disting-uished node in a tree DS is known as root node.

☛ The root node of tree contains a link to the roots of its subtrees.

☛ The left subtree is shown in yellow colour and right subtree is shown in green colour.



☛ The subtrees can furthur split into other subtrees.

☛ Recursion means reducing something in a self-similar manner, so this recursive property of tree DS is imple-mented in various applications.

Number of edges ☛ If there are 'n' nodes, then there would 'n-1' edges.

☛ Each arrow in the structure represents link or path.

Each node, except the root node, will have atleast one in-coming link known as an edge.

There would be one link for the parent-child relation.

Depth of Node X

The depth of node x can be defined as the length of the path from the root to the node x.

One edge contributes one-unit length in the path.

The depth of node x can also be defined as the number of edges between the root node and the node x.

The root node has '0' depth.

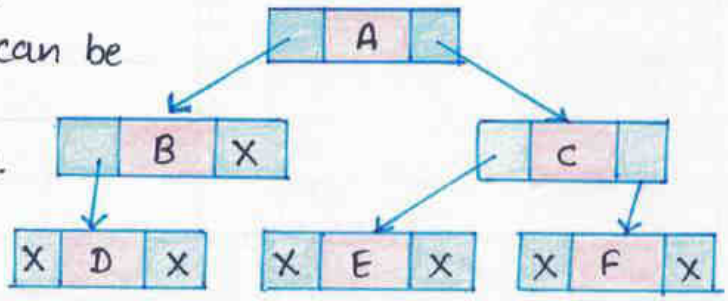
Height of Node x

The height of node x can be defined as the longest path from the node x to the leaf node

Implementation

The tree data structure can be created by creating the nodes dynamically with the help of the pointers.

The tree in the memory can be represented as shown:



This figure shows the representation of the tree Ds in the memory in which node contains three fields.

The second field stores the data, the first field stores the address of the left child, and the third field stores the address of the right child.

The structure of a node can be defined as:

```
Struct node { int data; structnode *left; struct node* right; }
```

The above structure can only be defined for the binary trees because binary tree can have utmost two children, & generic trees can have more than two children.

The structure of node for generic trees would be different as compared to the binary tree.

Applications

The following are the applications of trees:

Storing Naturally Hierarchical Data

Trees are used to store the data in the hierarchical structure.

For eg.→ file system. The file system stored on disc.

the file and folder are in the form of the naturally hier-archical data and stored in the form of trees.

Organize Data ☞ It is used to organize data for efficient insertion, deletion and searching.

☞ For eg. a binary tree has a logN time for searching.

Trie ☞ It is a special kind of tree that is used to store dictionary.

☞ It is a fast and efficient way for dynamic spell checking.

Heap ☞ It is also a tree DS implemented using arrays.

☞ It is used to implement priority queues.

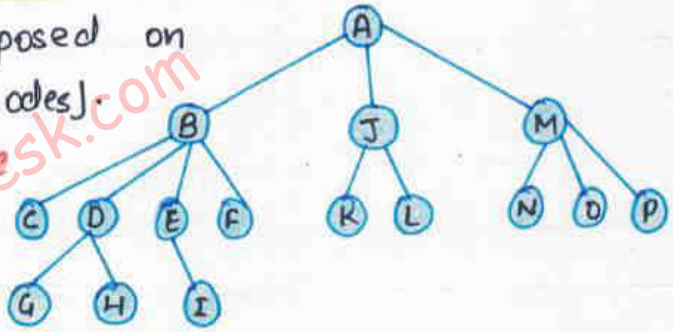
B-Tree and B+ Tree ☞ B-Tree and B+ Tree are the tree DS used to implement indexing in databases.

Routing Table ☞ The tree data structure is also used to store in routing tables in the routers.

☞ Types of tree data structure are following :-

- General Tree
- Binary Tree
- Binary Search Tree
- AVL Tree
- Red-Black Tree
- Splay Tree
- Treap
- B-Tree

GENERAL TREE ☞ In general tree, a node can have either 0 or maximum n numbers of nodes.



☞ There is no restriction imposed on the degree of the node (no. of nodes).

☞ Topmost node is root node & children of the parent node are known as subtrees.

☞ There can be n number of subtrees & these are unordered as the node in the subtree cannot be ordered.

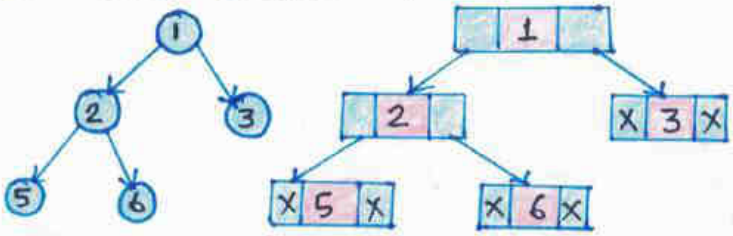
☞ Every non empty tree has a downward edge, & these edges are connected to the nodes known as child nodes.

☞ The root node is labeled with 0 & the nodes that have the same parent are known as siblings.

BINARY TREE

Binary Tree means that the node can have maximum two children.

Binary name itself suggests that 'two', therefore each node can have either 0, 1 or 2 children.



This tree is binary tree because each node contains the utmost two children.

In this tree node 1 contains two pointers, i.e. left and a right pointer pointing to the left and right node respectively.

The node 2 contains both the nodes : therefore it has two pointers (left & right)

The nodes 3, 5 &6 are the leaf nodes, so all these nodes contain NULL pointer on both left & right ports.

Properties

At each level of i, the maximum no. of nodes is 2^i.

The height of tree is longest path from the root node to leaf node. The above tree has height equal to 3.

In general, the maximum number of nodes possible at height h is-

$$(2^0 + 2^1 + 2^2 + \dots 2^h) = 2^{h+1} - 1$$

The mininum number of nodes possible at height h is $h+1$.

If the number of nodes is minimum, then height of the tree would be maximum & vice-versa.

The minimum height can be computed as: 'n' = number nodes.

we know that. $n = 2^{h+1} - 1$

$$n+1 = 2^{h+1}$$

Taking log both sides

$$\log_2(n+1) = \log_2(2^{h+1})$$

$$\log_2(n+1) = h+1$$

$$h = \log_2(n+1) - 1$$

The maximum height can be computed as:

n = numer of node

we know that,

$$n = h+1$$

$$h = n-1$$

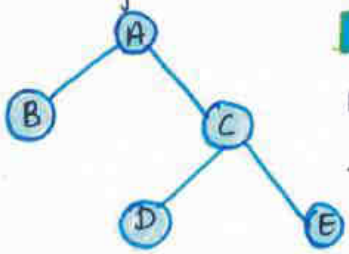
There are different types of binary tree:

- Full/Proper/strict Binary Tree
- Perfect Binary Tree
- Balanced Binary Tree
- Complete Binary Tree
- Degenerate Binary Tree

Full / Proper Binary Tree

The full binary tree is also known as a strict binary Tree.

The tree can only be considered as the full binary tree if each node must contain either 0 or two children.



In this tree we can observe that each node is either containing 0 or 2 children, therefore it is full binary tree.

Properties

The number of leaf nodes is equal to number of internal nodes plus 1;

In above example, the number of internal node is 5; therefore the number of leaf nodes is equal to 6.

The maximum number of nodes is the same as the no. of nodes in the binary tree, i.e. $(2^{h+1}-1)$.

The mininum number of nodes in the full binary tree is $(2*h-1)$.

The minimum height of the full binary tree is $\log_2(n+1)-1$.

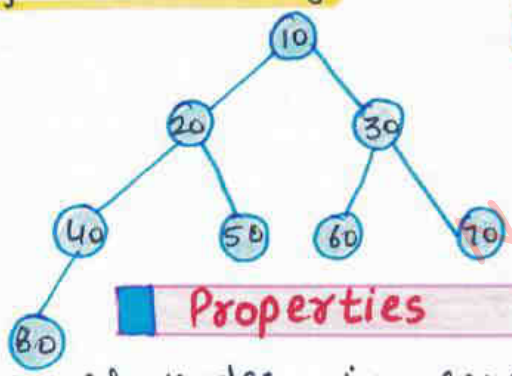
The maximum height of the full binary tree is $(n+1)/2$.

Complete Binary Tree

The complete binary Tree is a tree in which all nodes are completely filled except last level.

In last level, all the nodes must be as left as possible.

In a complete binary tree, the nodes should be added from the left.



This tree is complete tree is a complete binary tree because all the nodes are completely filled & all the nodes in the last level are added at the left first.

Properties

The maximum number of nodes in complete binary tree is $2^{h+1}-1$.

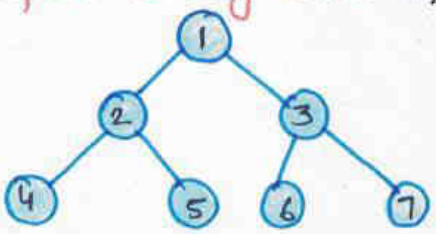
The minimum number of nodes in complete binary tree is 2^h.

The mininum height of a complete binary tree is $\log_2(n+1)-1$.

The maximum height of a complete binary tree is $(n+1)/2$.

Perfect Binary Tree

A tree is perfect binary tree if all the internal nodes have 2 children & all the nodes are at same level.

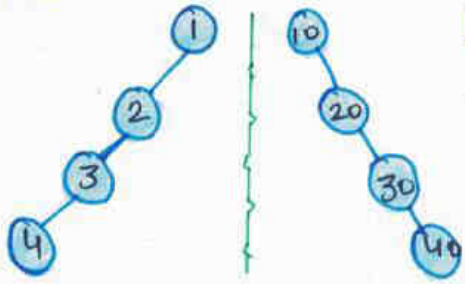


All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice-versa not true, i.e. all complete binary trees and full binary trees are perfect binary trees.

Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only 1 children.

Let's understand the Degenerate binary tree through example.



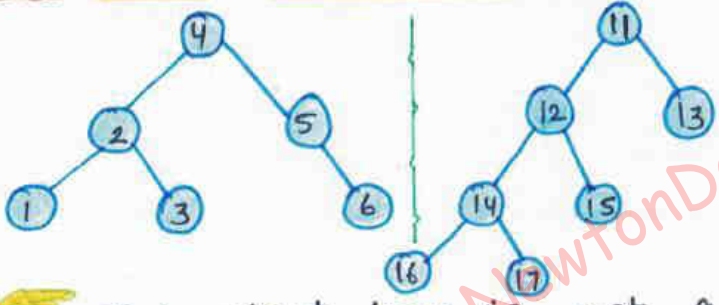
The both trees are degenerate binary tree because all the nodes have only one child.

One tree is a left-skewed tree as all the nodes have a left child only & the other tree is a right-skewed tree as all the nodes ha-ve a right child only.

Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1.

Eg: AVL & Red-Black trees are balanced binary tree.



The left tree is a balanced binary tree because the difference between the left subtree and right sub-tree is zero.

The right tree is not a balanced tree because the difference between the left subtree and the right subt-ree is greater than 1.

Implementation

A binary tree is implemented with the help of pointers.

The first node in the tree is represented by the root pointer, & each node in tree consists of three parts, i.e. data, leftpointer and right pointer.

To create a binary tree, we first need to create the node. We will create the node as shown below:-

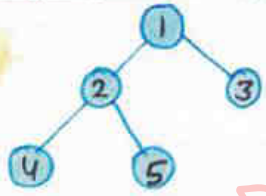
```
struct node { int data; struct node * left, * right;}
```

In the above structure, data is the value, left pointer contains the address of left node, & right pointer contains the address of the right node.

Eg: Binary Tree

```
# include < stdio.h>
  struct node { int data;
    struct node *left, *right;}
 void main (){
   struct node* root;
    root = create; }
  struct node* create (){
    structnode* temp;
    int data;
    temp = (struct node*) malloc
   (sizeof (struct node));
   printf ("Press 0 to exit");
   prinf ("\n Press 1 for new node);
   printf ("Enter your choice");
   scanf ("%d", &choice);
    if (choice ==0)
      return 0; }
    else {
   printf("Enter the data");
   scanf ("%d", &data);
    temp →data = data;
    printf("Enter left child of %d", data);
   temp →left = create();
   printf("Enter right child of %d", data);
   temp →right = create ();
    returntemp; } }
```

Eg: Count leaf nodes




```
# include <stdio.h>
# include <stdlib.h>
 struct node{ int data; struct
   node* left; struct node*
   right; };
/* function to count leaf nodes */
unsigned int getLCount ( struct
    node* node ) {
  if (node == NULL)
     return 0;
  if (node →left == NULL && node
   →right == NULL)
     return 1;
  else
    return getLCount (node →left) +
          getLCount (node →right); }
struct node* newNode (int data){
   Struct node* node = (structnode*
       malloc (sizeof (struct node));
    node →data = data;
    node →left = NULL;
    node →right = NULL;
    return (node); }
/* Driver program to test
   above function */
```

```
int main() {
 /* create a tree */
struct node *root = newNode(1);
  root->left =   newNode(2);
  root->right =  newNode(3);
 root->left->left = newNode(4);
 root->left->right = newNode(5);
 /* get leaf count of created tree */
 printf("Leaf count of the tree is
  %d", getLCount(root));
  getchar();
  return 0; }
```

O/P: The leaf count of the tree is 3.

Eg: Count Non leaf nodes


```
# include <stdio.h>
# include <stdlib.h>
 struct node { int info; struct
     node *left, *right ; };
 /* function to create new node */
 struct  node* createnode ( int key)
 { struct node* newnode = ( struct
   node*) malloc (sizeof (struct node));
    newnode -> info  =  key;
    new node ->left  =  NULL;
    newnode -> right =  NULL;
     return (newnode); }
 /* function to count no. of non leaf
   nodes */
    int  count = 0;
    int nonleafnodes (structnode* newnode)
 { if (newnode != NULL) {
     nonleafnodes (newnode->left);
  if ((newnode->left != NULL) ||
     (newnode->right != NULL))
   {  count ++; }
  nonleafnodes (newnode->right);}
    return count ; }
int main() {
 struct node *newnode =createnode
 (1);
 newnode-> left = createnode (2);
 newnode->right = createnode (3);
 newnode->left->left= createnode (4);
newnode->left->right= createnode(5);
 printf (" Non leaf count of
  the tree is %d", nonleafnodes
 (newnode));
   return 0; }
```

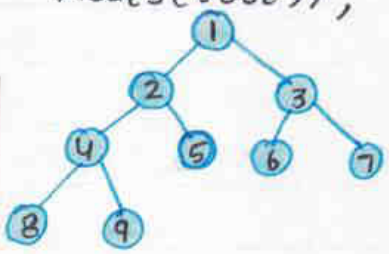
O/P: Non leaf count of the tree is 2

Eg: Count total no. of nodes

```
# include <stdio.h>
# include <stdlib.h>
 typedef  struct node { int data;
     struct node* left;  struct node*
     right; } node;
 node* newNode (int data) {
    node *Node = (node*) malloc
     (sizeof (node));
    Node ->data = data;
    Node -> left = NULL;
    Node -> right = NULL;
    return (node);
 /* function to get left height
    of tree */
int L-height(node* node) {
  int ht = 0;
```

```
while (node) {
    ht++;
    node = node->left; }
return ht; }
/* function to get right height */
int R_height (node* node) {
    int ht = 0;
    while(node){
        ht++;
        node = node->right; }
    return ht; }
/* Function to count nodes */
int TotalNodes (node* root) {
    if(root == NULL)
    return 0;
    int lh = L_height (root);
    int rh = R_height(root);
    if (lh == rh)
        return (1<< lh) - 1;
    return 1 + TotalNodes (root->left) + TotalNodes (root->right); }
int main() {
    node* root = newNode (1);
    root->left = newNode (2);
    root->right = newNode (3);
root->left->left = newNode (4);
root->left->right = newNode (5);
root->right->left = newNode (6);
root->right->right = newNode (7);
root->left->left->left = newNode (8);
root->left->left->right = newNode (9);
    printf (" Total number of nodes in created tree is:%d", TotalNodes(root));
    return 0; }
```

O/P: Total number of node in created tree is:
9



Binary Tree Traversal

A tree traversal is a method of visiting every node in the tree because all nodes are connected via edges, we start from the root node.

There are four common ways to traverse a binary tree:-

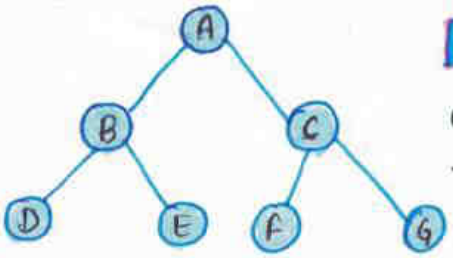
- Preorder
- Inorder
- Postorder
- Level order

Inorder Traversal

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins.

In other words, left subtree is visited first then the root and later the right sub-tree.

If a binary tree is traversed in order, the output will produce sorted key values in an ascending order.



We start from A, and following in-order traversal, we move to its left sub tree B & this process goes on until all the nodes are visited.

The output of inorder traversal of this tree will be-

D → B → E → A → F → C → G

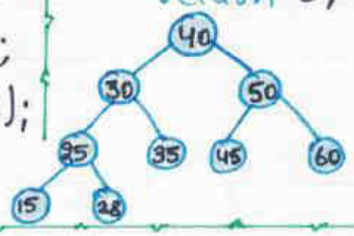
The time complexity of inorder traversal is O(n), where 'n' is the size of binary tree.

The space complexity of inorder traversal is O(1), if we do not consider the stack size for function calls.

Otherwise, the space complexity of inorder traversal is O(h), where 'h' is height of tree.

Eg: Inorder Traversal

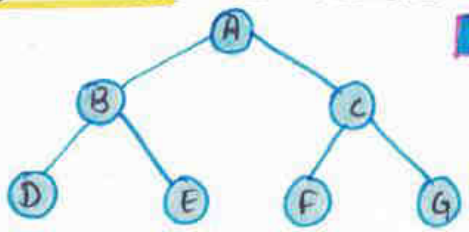
```
#include <stdio.h>
#include <stdlib.h>
struct node { int element; struct node* left, *right; };
/* To create new node */
struct node* createNode (int val){
  struct node* Node = (struct node*)
  malloc (sizeof (struct node));
  Node -> element = val;
  Node -> left = NULL;
  Node -> right = NULL;
  return (Node); }
/* function to traverse the nodes of binary tree in Inorder*/
void traverseInorder ( struct node *root ) {
  if (root == NULL)
    return;
  traverseInorder (root -> left);
  printf ("%d ", root -> element);
  traverseInorder (root -> right);
}
int main () {
  struct node* root = createNode (40);
  root -> left = createNode (30);
  root -> right = createNode (50);
  root -> left -> left = createNode (25);
  root -> left -> right = createNode (35);
  root -> left -> left -> left = createNode (15);
  root -> left -> left -> right = createNode (28);
  root -> right -> left = createNode (45);
  root -> right -> right = createNode (60);
  printf ("\n The inorder traversal of given BT is: \n");
  traverseInorder (root);
  return 0; }
```



O/P: The Inorder traversal of given BT is: 15 25 28 30 35 40 45 50 60

Preorder Traversal

In preorder traversal, first root node is visited, then left sub tree and after that right sub-tree is visited.



The preorder traversal technique follows the Root Left Right policy.

The output of pre order traversal of this tree will be - A → B → D → E → C → F → G

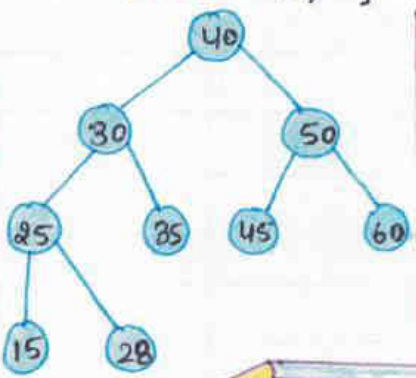
The time complexity of pre order traversal is O(n)

The space complexity of preorder traversal is O(1), if we do not consider the stack size for function calls.

Otherwise, the space complexity of preorder traversal is O(h), where 'h' is height of the tree.

Eg: Preorder Traversal

```
#include <stdio.h>
#include <stdlib.h>
Struct node { int element;
  structnode* left, *right;};
  To create new node
structnode* crNode (intval){
 struct node* Node = (structnode*)
  malloc (sizeof (structnode));
  Node → element = val;
  Node → left = NULL;
  Node → right = NULL;
  return (Node); }
  Function to traverse the
  nodes of BT in pre order
void traverse Preorder (struct
    node* root ) {
  if (root == NULL)
     return ;
  printf (" %d", root→element);
  traverse Preorder (root→left);
  traverse. Preorder (root→right);
  }
int main ( ) {
  struct node* root = crNode (40);
  root→left = crNode (30);
  root→right = crNode (50);
  root→left→left = crNode (25);
  root→left→right = crNode (35);
  root→left→left→left = crNode (15);
root→left→left→right = crNode (28);
root→right→left = crNode (45);
root→right→right = crNode (60);
  printf (" \nThe preorder trav-
  -ersal of given BT is:\n" );
   traverse Preorder (root);
    return 0; }
```

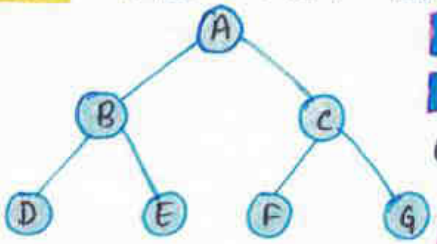


O/P: The preor-der traversal of given BT is: 40 30 25 15 28 35 50 45 60

Postorder Traversal

In post order traversal method the root node is visited last.

First we traverse the left subtree, then the right sub tree and then the right subtree and finally the root node.



It follows the LRN (Left-right-node)

The output of post-order traversal of this tree will be -

D → E → B → F → G → C → A

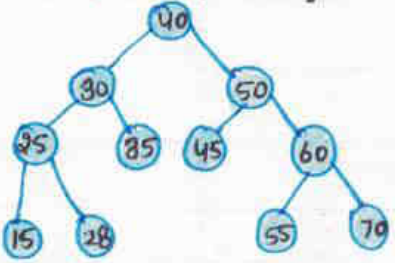
The time complexity of post order traversal is O(n).

The space complexity of post order traversal is O(1), if we do not consider the stack size for function calls.

Otherwise, the space complexity of post order traversal is O(h), where 'h' is height of tree.

Eg: Postorder Traversal

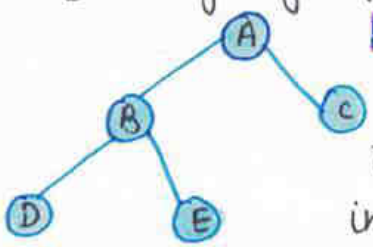
```
#include<stdio.h>
#include<stdlib.h>
struct node{ int element; struct node* left; struct node* right;};
To create a new node
struct node* crNode (int val) {
structnode* Node = (structnode*) malloc (sizeof( struct node));
Node -> element = val;
Node -> left = NULL;
Node -> right = NULL;
return (Node); }
/* function to traverse the nodes of binary tree in postorder */
void traversePostorder (structnode* root)
{ if (root == NULL)
    return;
  traversePostorder (root->left);
  traverse Postorder (root->right);
  printf ("%d ", root -> element); }
int main() {
structnode* root = createNode (40);
root ->left = createNode (30);
root ->right = createNode (50);
root ->left->left = createNode (25);
root->left->right = createNode (35);
root->left->left->left = createNode (15);
root->left->left->right = createNode (28);
root ->right->left = createNode (45);
root ->right->right= createNode (60);
root->right->right->left= createNode (55);
root->right->right->right = createNode (70);
printf ("\n The Postorder traver-sal of given BT is \n ");
  traverse Postorder (root);
  return 0; }
```



The post order traversal of given BT is:
15 28 25 35 30 45 55 70 60 50 40

Level Order Traversal

In a level order traversal, the nodes are visited level by level starting from the root and going from left to right.



This order requires a queue data structure, so it is not possible to develop a recursive procedure to traverse the binary tree in level order.

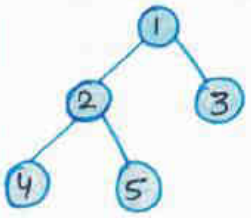
This is nothing but a breadth first search technique.

The output of above tree will be - A → B → C → D → E

Eg: Level order Traversal

```
#include<stdio.h>
#include<stdlib.h>
struct node { int data;
 struct node * left, * right;};
void printCurrentLevel (struct node*
    root, int level);
int height (structnode* node);
struct node* newNode (int data);
/* Function to print level order
  traversal a tree */
void printLevelOrder (structnode*
                root);
{    int h = height (root);
     int i;
   for (i= 1; i<= h; i++)
     printCurrent Level (root, i);
void printCurrenLevel (structnode*
   root, int level)
{    if (root= = NULL)
         return;
     if (level = = 1)
   printf ("%d", root data);
  elseif (level>1) {
     printCurrentLevel (root→ left,
     level-1);
   printCurrentLevel (root→right,
     level-1); }}
int height ( struct node* node)
  {  if (node ==NULL)
       return 0;
     else {
   int lheight = height(node→left);
   int rheight = height(node→right);
     if (lheight >rheight )
      return (lheight +1);
      else
       return (rheight +1); }}
struct node* newNode ( int data)
{  struct node* node = (struct
     node* )malloc (sizeof (structnode));
     node→ data = data ;
     node → left = NULL;
     node → right = NULL;
     return (node); }
int main ( ) {
struct node*  root = newNode(1);
     root → left = newNode (2);
     root → right = newNode (3 );
    root → left → left = newNode (4);
    root → left → right = newNode(5 );
```

```
printf("Level order traversal of binary tree is\n");
printLevelOrder(root);
return 0; }
```

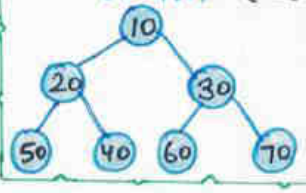


O/P: Level order traversal of binary tree is: 1 2 3 4 5

Time and space complexity of level order traversal is O(n) where n is the number of nodes in the tree.

Eg: Program to check whether binary tree is full or not.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
struct Node { int key; struct Node* left, *right; };
struct Node *newNode(char k)
{ struct Node *node = (struct Node*)malloc(sizeof(struct node));
  node->key = k;
  node->right = node->left = NULL;
  return node; }
/* Function to test full binary tree or not */
bool isFullTree(struct Node* root)
{ if (root == NULL)
    return true;
  if (root->left == NULL && root->right == NULL)
    return true;
  if ((root->left) && (root->right))
    return (isFullTree(root->left) && isFullTree(root->right));
  return false; }
int main()
{ struct Node* root = NULL;
  root = newNode(10);
  root->left = newNode(20);
  root->right = newNode(30);
  root->left->left = newNode(50);
  root->left->right = newNode(40);
  root->right->left = newNode(60);
  root->right->right = newNode(70);
  if (isFullTree(root))
    printf("The BT is Full");
  else
    printf("The BT is not full");
  return (0); }
```



O/P: The Binary Tree is Full.

Eg: mirror image of binary tree

```
#include <stdio.h>
#include <stdlib.h>
struct Node { int data; struct Node* left, *right; };
struct Node* newNode(int data)
{ struct Node* node = (struct Node*)malloc(sizeof(struct Node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return (node);
}
```

```
void mirror (struct Node* node)
{   if (node==NULL)
      return ;
    else {
   struct Node* temp;
    mirror ( node→left);
    mirror (node→ right);
/* swap the pointers */
   temp  = node→left ;
   node→left = node→right;
   node→right = temp; }}
void inOrder ( struct Node* node)
{    if (node==NULL)
       return ;
     inOrder (node→left);
   printf ("%d ", node→data);
   inOrder ( node→right); }
```

```
int main ( ) {
  struct Node* root = newNode (1 );
   root → left  =  newNode (2 );
   root →right =  newNode (3 );
  root→ left→left = newNode (4 );
  root →left→right = newNode (5 );
  printf ("Inorder traversal of
   the constructed tree is\n");
   inOrder (root);
    mirror (root);
 printf ("Inorder traversal of
 the mirror tree is\n");
   inOrder (root );
    return 0; }
```

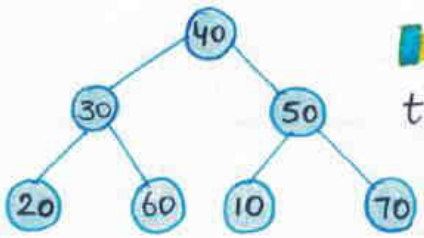
```
O/P: Inorder traversal of construc-
ted tree is: 4 2 5 1 3
Inorder traversal of the mirror
  tree is: 3 1 5 2 4
```

BINARY SEARCH TREE

A binary search tree follows some order to arrange the elements.

In a binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node.

This rule is applied recursively to the left & right subtrees of the root.



In this figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root and all the nodes of right sub-tree are greater than the root node, so it is Binary search tree.

BST is a collection of nodes arranged in a way where they maintain BST properties & each node has a key and an associated value.

- We can perform insert, delete and search operations on the binary search tree (BST).
- In BST, searching a node is very easy because elements in BST are stored in a specific order.
- To delete a node from BST, there are three positions-
  1. The node to be deleted is the leaf node
  2. The node to be deleted has only one child.
  3. The node to be deleted has two children.
- For deleting a node we have to replace the leaf node with NULL and replace with child node when only one child possibility occured.
- When it has two children then these steps followed-
  1. first find the inorder successor of the node to be deleted.
  2. After that replace that node with the inorder successor until the target node is placed at the leaf of tree.
  3. And at last, replace the node with NULL and free up the allocated space.
- The inorder successor is required when the right child of the node is not empty.
- A new key in BST is always inserted at the leaf node.
- The time complexity of insertion, deletion & search in worst case is O(n) & in best & average case is, O(logn), where 'n' is the number of nodes.
- The space complexity of insertion, deletion and search is O(n) where n is number of nodes.

Eg. Binary Search Tree

```
#include<stdio.h>
#include<stdlib.h>
 typedef struct BST
  { int data;
    struct BST * left;
    struct BST* right;}node;
 node* create ();
void  insert (node*, node*);
void  preorder (node*);
int main ( ) {
  char ch;
 node*root = NULL *temp;
 do {
  temp = create ( );
 if (root = = NULL)
   root = temp;
 else
  insert (root, temp);
 printf (" Do you want to enter ");
```

```
    getchar();
  scanf("%c",&ch);}
  while(ch=='y'|ch=='Y');
   printf("Preorder Traversal");
   preorder(root);
   return 0;}
   node* create(){
   node *temp;
   printf("Enter data:");
    temp=(node*)malloc(size
    of(node));
   scanf("%d",&temp->data);
   temp->left=temp->right=NULL;
    return temp;}
void insert(node*root,node*temp)
 {if(temp->data<root->data){
   if(root->left!=NULL)
     insert(root->left,temp);
   else
     root->left=temp;}
  if(temp->data>root->data){
    if(root->right!=NULL)
    insert(root->right,temp);
    else
     root->right=temp;}}
void preorder(node*root){
   if(root!=NULL){
   printf("%d",root->data);
    preorder(root->left);
    preorder(root->right);
    }}
```

```
O/P: Enter data: 5
Do you want to enter more: y
Enter data: 10
Do you want to enter more: y
Enter data: 13
Do you want to enter more: y
Enter data: 2
Do you want to enter more: n
Preorder Traversal: 5 2 10 13
```

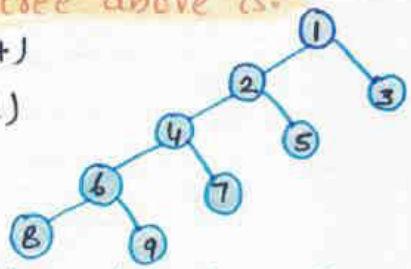
The post order traversal of a binary tree is 8,9,6,7,4,5,2,3,1. The inorder traversal of the same tree is 8,6,9,4,7,2,5,1,3. The height of a tree is the length of longest path from root to any node (leaf). The height of the binary tree above is.

solution post – 8, 9, 6, 7, 4, 5, 2, 3, 1 (L, R, R+)

in order – 8, 6, 9, 4, 7, 2, 5, 1, 3 (L, Rt, R)

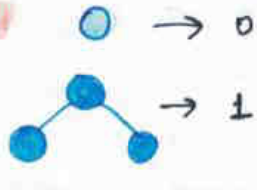
h = 4 . It is just binary tree not BST.

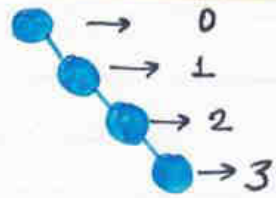
Height = 4



Let T be a binary search tree with 15 nodes. The min. & max. possible heights of T are: NOTE– height of tree with single node is 0 (A) 4&15 (B) 3&14 (C) 4&14 (D) 3&15

solution



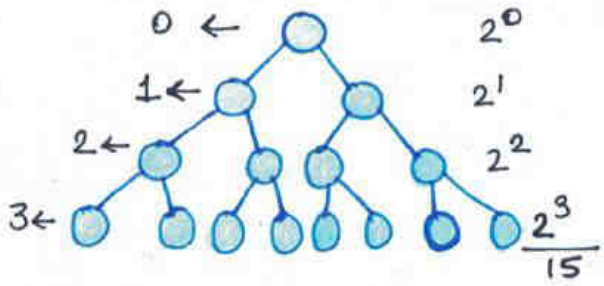


Min. height $= \log_2(15+1) - 1$

$= 4 - 1 = 3$

nodes → 2^0

$\lceil \log_2(n+1) \rceil - 1 = h$ → $+2^1 + 2^2 + 2^3$



$0 \rightarrow 1 = 2^0 = 2^1 - 1$

$1 \rightarrow 1+2 = 3 = 2^2 - 1$

$2 \rightarrow 1+2+4 = 7 = 2^3 - 1$

$3 \rightarrow 1+2+4+8 = 15 = 2^4 - 1$

$2^{k+1} - 1$

Max. $h = n-1$

$\Rightarrow 15-1 = 14$

[B]. 8 & 14

Que Let A be an array of 31 numbers consisting of a sequence of 0's followed by a sequence of 1's. The problem is to find the smallest index i such that A[i] is 1 by probing the min. no. of locations in A. The worst case number of probes per-formed by an optimal algo is- (A) 2 (B) 3 (C) 4 (D) 5

Solution: 1, 2, 3, 4 ----- 31

0 0 0 ..... 1 1 1 1 → sorted

[D] 5

Binary Search: 1 2 3 4 ---- 31

0 1 1 1

$m = \left[\frac{l+r}{2}\right] = 16$ A[16] = 1

$l = 1$, $r = 16$ ⇒ A[8] = 1, A[2] = 1

A[1] = 1

Worst case of this problem will be 1 at end & take log n time. n = 31,

$\log_2 31 = 5$

Que Let T(n) be the number of different binary search trees on n distinct elements: $T(n) = \sum_{k=1}^{n} T(k-1)\,T(x)$ where x is

(A) n-k+1 (B) n-k (C) n-k-1 (D) n-k-2

Solution

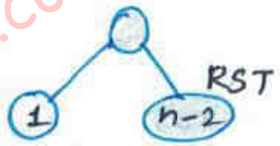
k = 1, 2, 3, ---- n

n = 1 + K - 1 + RST

RST = n-K

when K = 1

for K = 2



DIP: [B]. n-K

Que Supposte the numbers 7 5 1 8 3 6 0 9 4 2 are inserted in that order into an initially empty binary search tree. The BST uses the usual ordering on natural numbers. What is inorder traversal sequence of resultant tree.

(A) 7, 5, 1, 0, 3, 2, 4, 6, 8, 9 (C) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

(B) 0, 2, 4, 3, 1, 6, 5, 9, 8, 7 (D) 9, 8, 6, 4, 2, 3, 0, 1, 5, 7

Solution inorder traversal → automatically generates sorted array.

SO, → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 [C] option

Que What is the tightest upper bound that represents the time complexity of inserting an object into a BST of n-nodes. (A) O(1) (B) O(logn) (C) O(n) (D) O(nlogn)

Solution To insert an element, we need to search for its place first. The search operation may take O(n) for a skewed tree.

[C]. O(n)

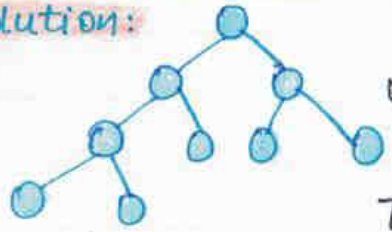
Que What are the worst case complexities of insertion and deletion of a key in BST.
(A) O(logn) for both insert & delete (B) O(n) (C) O(n) for insertion O(logn) for deletion (D) O(logn) for insertion & O(n) for deletion.

Solution The worst case BST behave like LL so O(n) for insertion & deletion.

[B] (O(n))

Que Why are given a set of n distinct elements and an unbalanced binary tree with n nodes. In how many ways can we populate the tree with the given set so that it becomes a BST. (A) 0 (B) 1 (C) n! (D) $\left(\frac{1}{n+1}\right)^{2n}C_n$.

Solution:



[B]. 1

unlabelled structure of the tree is fixed.
n = 9 1, 2, 3, 4, 5, 6, 7, 8, 9
There is only one single way.

Que Linked list are not suitable data structures for which one of the following problems –
(A) Insertion sort (B) Binary search (C) Radix sort (D) None

[B]. Binary Search

Que Consider the C function given below. Assume that the array list h contains n(>0), h(>0) elements sorted in ascending order –

```
int ProcessArray ( int *listA,
int x, int n){
  int i, j, k ;
  i = 0; j = n-1 ;
  do{
    k = (i+j)/2;
    if(x<= listA[k])
      j = k-1;
    if (listA[k] <=x)
      i = k+1 ; }
```

```
while (i<=j) {
  if (list A[k]==x)
    return (k);
  else
    return -1; }}
```

Which one of the following statements about function processing is correct;

(A) It will run into an infinite loop when x is not in last A.

(B) It is an implementation of Binary Search.

(C) It will always find the max element in list A.

(D) It will return -1 even where x is present in list A.

Solution The function is iterative implementation of Binary search & k keeps track of current middle element.

i & j keep track of left and right corners of current subarray.

[B] option

Que The unusal $\theta(n^2)$ implementation of insertion sort to sort an array uses linear search to identify the position where an element is to be inserted into the already sorted part of the array. If inserted, we use binary search to identify the position worst case running time will-

(A) remain $\theta(n^2)$

(B) become $\theta(n(\log n)^2)$

(C) become $\theta(n \log n)$

(D) become $\theta(n)$

Solution If we use binary search then there will be $\log_2(n!)$ comparisons in the worst case, which is $\theta(n \log n)$.

But the algorithm as a whole will still have a running time of $\theta(n^2)$ on average because of the series swaps required for each insertion.

[A] remain $\theta(n^2)$

Que Consider the label sequences obtained by the following pairs of traversals on a labelled binary tree. Which of these pairs identify a tree uniquely.

(i) Preorder & postorder

(ii) inOrder & Post-order

(iii) Preorder & in-order

(iv) level order & postorder

(A) (i) only

(B) (ii), (iii)

(C) (iii) only

(D) (iv) only

Solution Pre-order → Rt, L, R

Post-order → L, R, Rt

In-order → L, Rt, R

level order → level-by-level

In level order first print first level elements then second & third & so on——.

[B] (ii) & (iii)

Pre	Rt	L	R
Post	L	R	Rt

⇒ We can not get unique tree using Pre - post. We can get using In + Pre, In + Post, so, Option (B) is correct.

⇒ level order & post order

level-order: AB
Post-order: BA

level order = AB
Post order = BA

Both the trees are not same but their traversal are same, so this is also not correct.

Que In a binary tree with n-nodes, every node has an odd no. decendants. Every node is considered to be its own de-cendants. What is the no. of nodes in the tree that have exactly one child. (A) 0 (B) 1 (C) (n-1)/2 (D) n-1

Solution

→ No. of decendants = 1

→ No. of decendants = 2

Two children,
No. of des. → 3

Binary Tree - Every node has 0 or 2 children so answer is 0.

[A]. 0

Que The preorder traversal of a binary search tree is given by 12, 8, 6, 2, 7, 9, 10, 16, 15, 19, 17, 20 Then the post order traversal of this tree is -

(A) 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 19, 20
(B) 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12
(C) 7, 2, 6, 8, 9, 10, 20, 17, 19, 15, 16, 12
(D) 7, 6, 2, 10, 9, 8, 15, 16, 17, 20, 19, 20

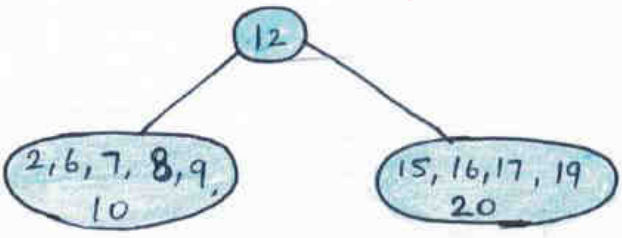
Solution

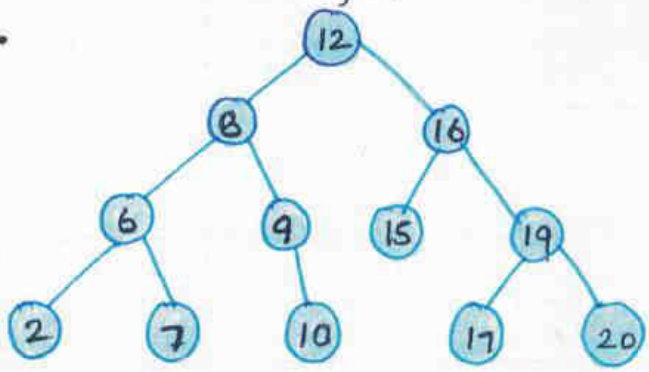
Pre: 12, 8, 6, 2, 7, 9, 10, 16, 15, 19, 17, 20

In: 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 19, 20

[B]. option

start from first in Pre-order traversal for insertion location check it in In-order.





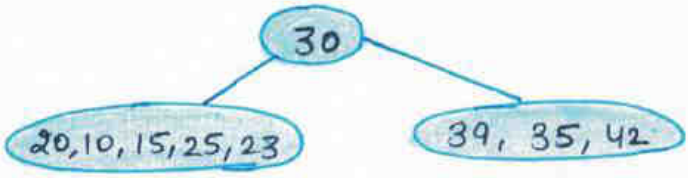
Post order: L, R, Rt

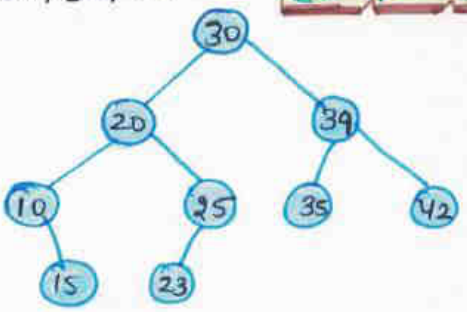
2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12.

Que The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the post order traversal sequence of the same tree.

(A) 10, 20, 15, 23, 25, 35, 42, 39, 30
(B) 15, 10, 25, 23, 20, 42, 35, 39, 30
(C) 15, 20, 10, 23, 25, 42, 35, 39, 30
(D) 15, 10, 23, 25, 20, 35, 42, 39, 30

Solution: Pre: 30, 20, 10, 15, 25, 35, 23, 39, 35, 42 — (D) option





Post: 15, 10, 23, 25, 20, 35, 42, 39, 30

Everything is working here, because, it is binary search tree.

Que In the binary tree the number of internal nodes of degree 1 is 5. And number of internal nodes of degree 2 is 10. The number of leaf nodes in the binary tree is-

(A) 10 (B) 11 (C) 12 (D) 15

Solution: Degree of node: Number of other nodes, which are connected to this node. — [B] 11

5 internal nodes → deg. 1
10 internal nodes → deg. 2

	node	edges
○ →	1	0
○—○ →	2	1
○—○—○ →	3	2

In tree for 'n' node (n-1) edges. So, there are 11 leaf nodes in the tree.

5 int. nodes of deg. 1 → 5 edges
10 int. nodes of deg. 2 → $\frac{10 \times 2 = 20}{25 \text{ edges}}$

Binary tree can have deg. 0, 1, 2.

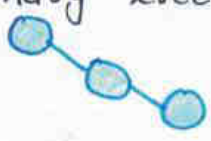
n nodes → (n-1) edges
26 nodes ← 25 edges

leaf nodes = 26 - 5 - 10 = 11

Que A scheme for storing binary trees in an array X is as follows. Indexing of X starts at 1, instead of 0. The root is stored at X[1]. for a node stored at X[i], the left child, if any is stored in X[2i] and the right child, if any in X[2i+1]. To be able to store any binary tree on n vertices, the minimum size of X should be-

(A) $\log_2 n$ (B) n (C) 2n+1 (D) $2^n - 1$

Solution: For a right skewed binary tree, number of nodes will be $2^n - 1$. — [D] $2^n - 1$



AVL TREE

AVL tree is invented by GM Adelson-velsky and EM Landis in 1962.

AVL tree can be defined as height balanced binary search tree in which each node is associated with a balance factor.

AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than one, this difference is called as Balance factor.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced & need to be balanced.

Balance Factor = height (left - subtree)

If the difference in height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations-

- Left rotation
- Right rotation
- Left-right rotation
- Right-Left rotation

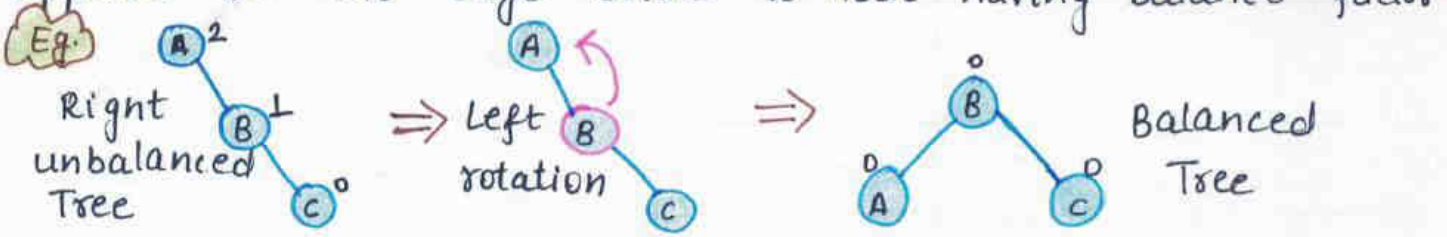
The first two rotations are single rotations and the next two rotations are double rotations.

RR - Rotation

When BST becomes unbalanced due to a node is inserted into right subtree of the right subtree of A, then we perform RR rotation.

RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2.

Eg.



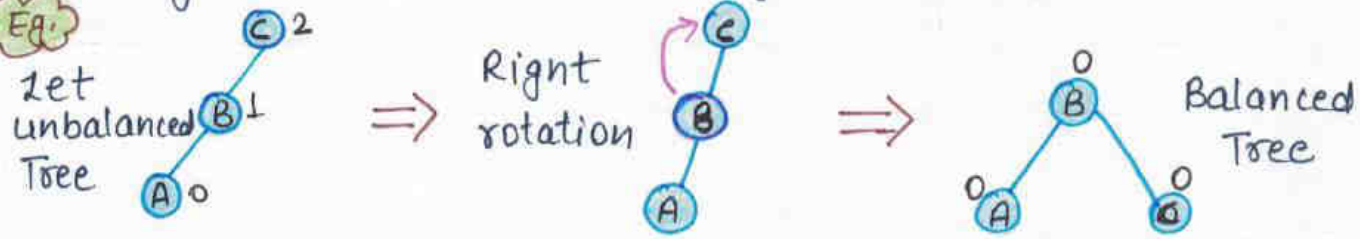
In above example, node A has balance factor -2 because a node C is inserted in the right subtree of a right subtree. We perform the RR rotation on the edge below A.

LL - Rotation

When BST becomes unbalanced, due to node is inserted into left subtree of the left subtree of C, then we perform LL rotation.

LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

Eg:



Let unbalanced Tree

In the above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left sub-tree. We perform LL rotation on the edge below A.

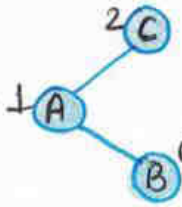
LR - Rotation

LR rotation is double rotation which is tougher than single rotation.

LR rotation = RR rotation + LL rotation, i.e. first RR rotation is performed on sub tree, & then LL rotation is performed on full tree.

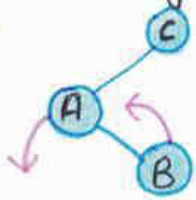
By full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0 or 1.

Lets understand each and every step clearly:-

1. 

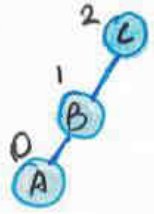
A node 'B' has been inserted into the right subtree of 'A' the left subtree of C, because of which C has become an unbalanced node having balance factor 2.

2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C.

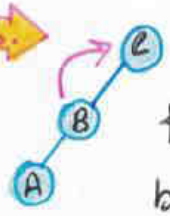
3. 

As LR rotation = RR+LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first.

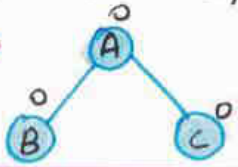
4. By doing RR rotation, node A has become the left subtree of B.

5. 

After performing RR rotation, node C is still unbalanced, i.e. having balance factor 2, as inserted node A is in the left of left C.

6. 

Now we perform LL clockwise rotation on full tree, i.e. on node C. Node C has now become the right subtree of node B, A is left sub tree of B.

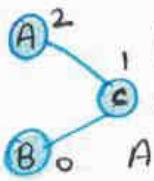
7. 

Balanced factor of each node is now either -1, 0, 1 i.e. BST is balanced now.

RL - Rotation

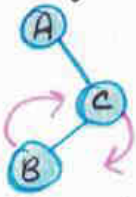
RL rotation = LL rotation + RR rotation, first LL rotationed is performed on full tree, full tree means balance factor other than -1, 0, 1.

Let us understand each & every step clearly.

1. 

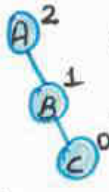
A node B has been inserted into the left subtree of C the right subtree of A. because of which A has become an unbalanced node having B.F = -2.

2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A.

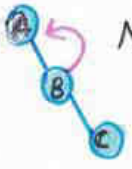
3. 

As RL rotation = LL rotation + RR rotation, hence, LL (clockwise rt.) on subtree rotation at C is performed first.

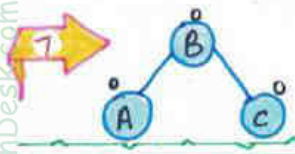
4. By doing RR rotation, node C has become the right subtree of B.

5. 

After performing LL rotation, Node A has still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right subtree node A.

6. 

Now we perform RR rotation anti clock wise rotation on full tree, i.e. a Node A, node C has now become the right subtree of node B and node A has become the left subtree of B.



Balance factor of each node is now either -1, 0 or 1. BST is balanced now.

Operations On AVL Tree

AVL tree is also a binary search tree, therefore, all the operations are performed in the same way as they are performed in binary search tree.

Insertion

In insertion operation the new node is added into AVL tree as the leaf node.

It may lead to violation in the AVL tree properties & therefore tree can be balanced by appling rotations.

Rotation is required only if the balance factor of any node is disturbed upon inserting the new node.

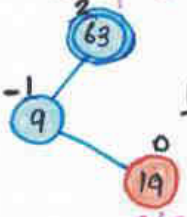
Eg. Construct an AVL Tree by inserting 63, 9, 19, 27, 18, 108, 99, 81.

Solution At each step, we must calculate the balance factor for every node, if it is found to be more than 2 or less than -2, then we need a rotation to rebalance the tree.

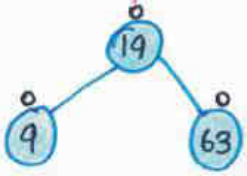
Step 1

Step 2

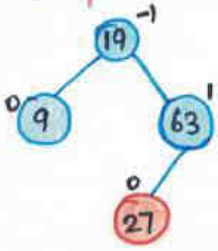
Step 3



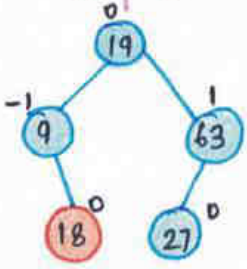
Step 4



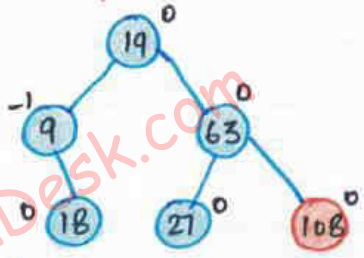
Step 5



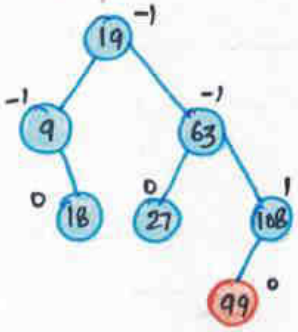
Step 6



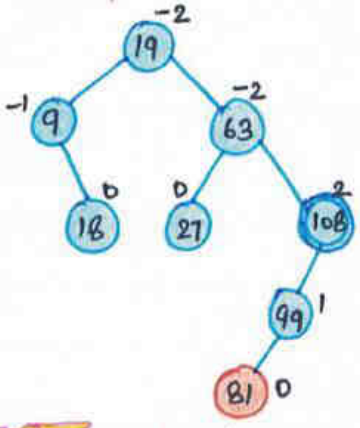
Step 7



Step 8

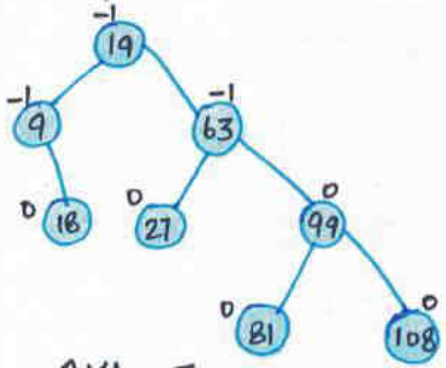


Step 9



LL Rotation

Step 10



AVL Tree

Height of a NULL tree and single node tree is -1 & 0 respectively.

Deletion

Deletion may disturb the balance factor of an AVL tree then we need to perform rotations to rebalance the tree.

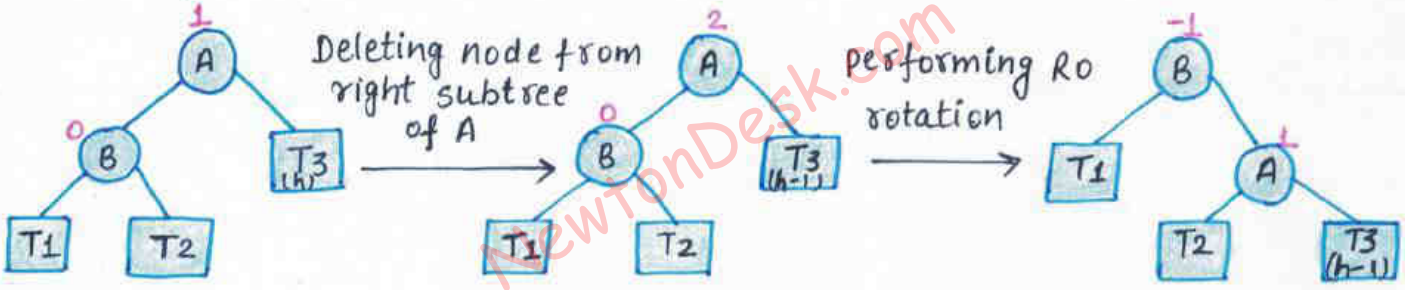
There are two types of rotations are performed → L-rotation and R-rotation, both are mirror images of them.

If the node which is to be deleted is present in the left sub tree of the critical node, then L rotation needs to be applied else if the node which is to be deleted is present in the right sub tree of the critical node then R-rotation needs to be applied.

Let's consider that, A is the critical node, B is the root node of its left subtree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three diff-erent situations: R0 rotation, R1 rotation, R-1 rotation.

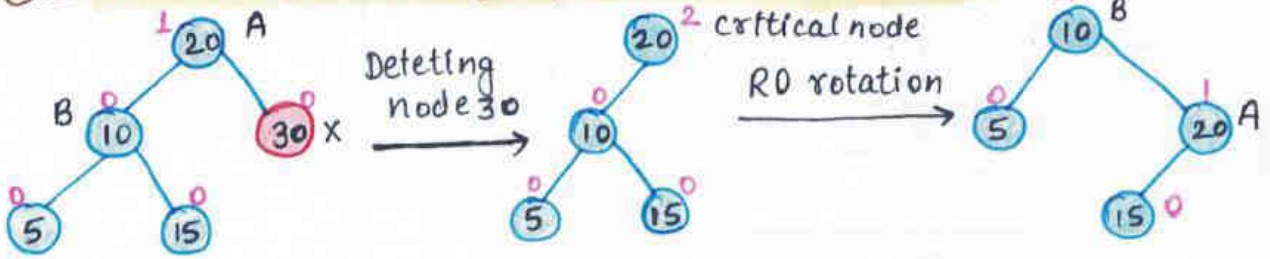
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

1. The critical node A is moved to its right and the node B becomes the root of the tree with T_1 as its left sub-tree.
2. The sub trees T2 & T3 becomes the left and right sub-trees of the node A.



AVL Tree | Non AVL Tree | R0 rotated Tree

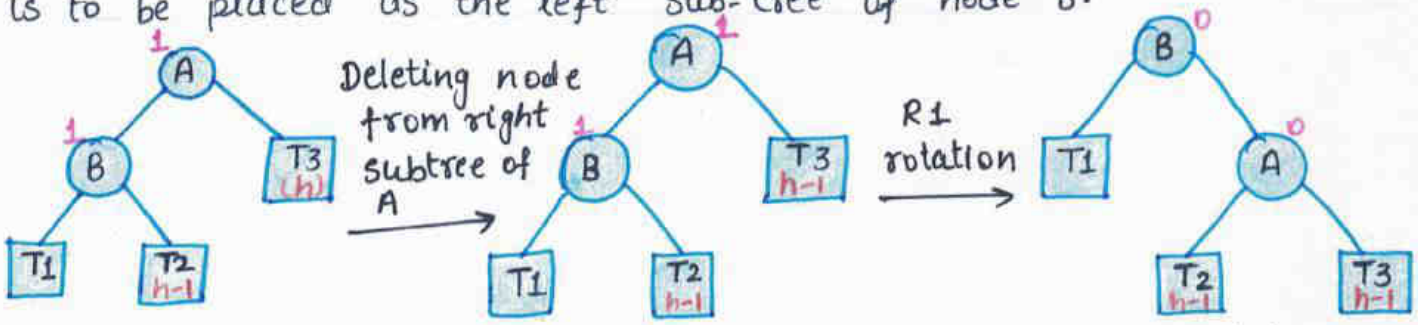
Eg: Delete the node 30 from AVL Tree.



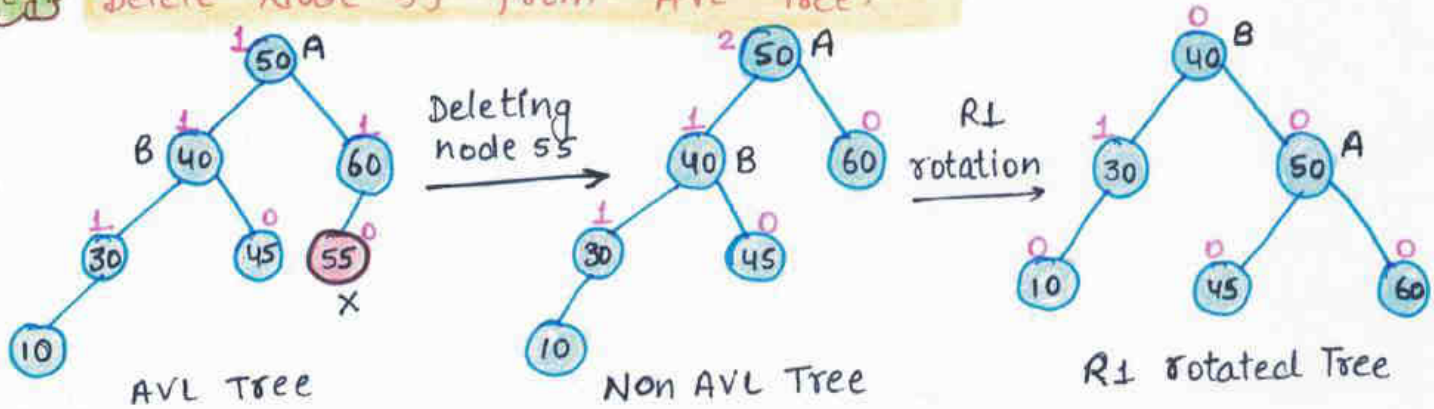
AVL Tree | Non-AVL Tree | R0 rotated Tree

R1 rotation is to be performed if the BF (Balance factor) of node B is 1.

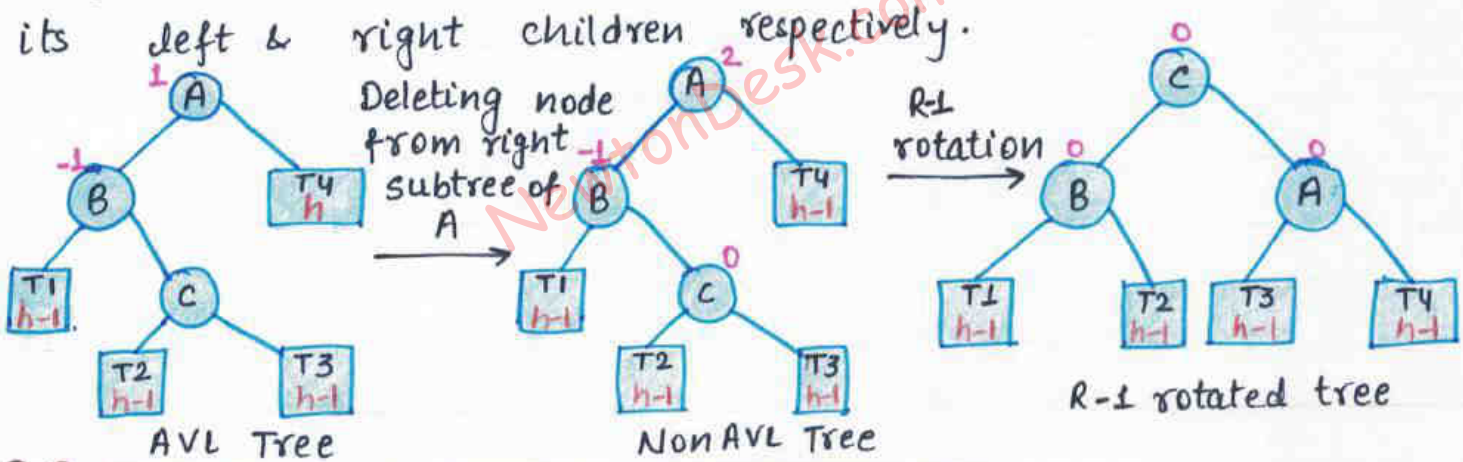
In R1 rotation, the critical node A is moved to its right having subtrees T2 & T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of node B.



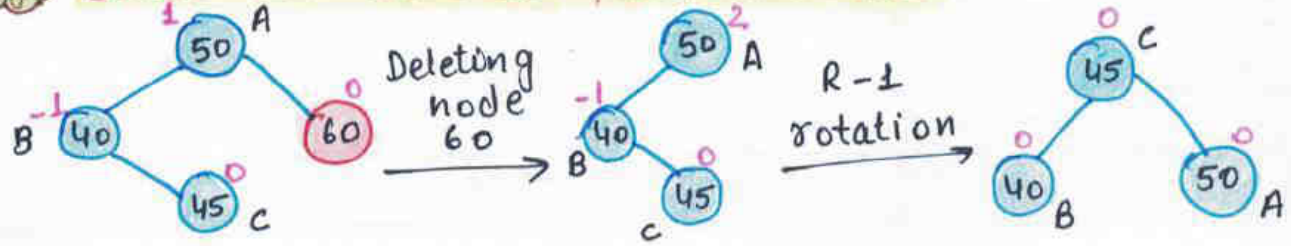
Eg: Delete Node 55 from AVL Tree.



R-1 Rotation is to be performed if the node B has BF -1. In this case, the node C, which is the right child of nodeB, becomes the root node of the tree with B & A as its left & right children respectively.



Eg. Delete the node 60 from AVL Tree.



Time complexity of search, insert & delete in average & worst case is O(logn).

Space complexity in average & worst case is O(n).

Eg. AVL Tree

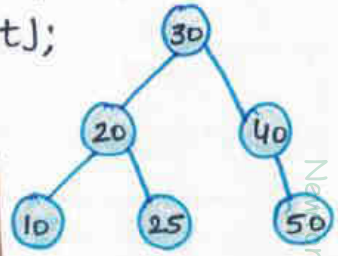
```
#include <stdio.h>
#include <stdlib.h>
/* AVL tree node */
struct Node { int key; struct
 Node * left, * right ;
  int height ; };
/* utility function to get maxim.
 of two integers */
int max ( int a , int b);
/* utility function to get the height
 of the tree */
int height ( struct Node *N){
 if ( N == NULL)
     return 0;
   return 1 + max ( height N->left)
       height ( N->right )); }
int max( int a , int b ){
  return (a > b) ? a: b; }
struct Node* newNode( int key) {
 struct Node* node = (struct Node*)
    malloc (sizeof ( struct Node));
   node -> key = key;
   node -> left = NULL;
   node -> right = NULL;
   node ->height = 0; /* new
node initially added at leaf*/
  return (node ); }
/* funtion to right rotate subtree*/
struct Node *rightRotate (
  struct Node *y) {
  struct Node *x = y left;
  struct Node * T2 = x right;
  /* peform rotation */
   x ->right = y;
   y ->left = T2;
  /* update heights */
   y ->height = height (y);
   x ->height = height (x);
    return x; }
/* function to left rotate subtree*/
struct Node * leftRotate (
 struct Node* x ) {
 struct Node *y = x ->right;
 struct Node * T2 = y-> left;
 /* perform rotation*/
  y ->left = x;
  x ->right = T2;
 /* update heights */
  x ->height = height (x);
  y ->height = height (y);
   return y; }
/* Recursive function to insert
 a key */
struct Node * insert ( struct
 Node * node, int key) {
/* 1. Perform normal BST ops*/
 if (node == NULL)
   return (newNode (key));
 if (key < node-> key)
  node->left = insert (node
    ->left , key));
```

```
else if (key > node->key)
    node->right = insert(node
        ->right, key);
else /* equal keys not allowed*/
    return node;
/* 2. Update height of this ance-
-stor node*/
 node->height = height(node);
/* 3. Get the balance factor*/
int balance = getBalance(node);
if (balance >1 && key < node->
      left->key) /* LL case*/
  return rightRotate(node);
/* Right Right case */
if (balance <-1 && key >node->
      right->key)
  return leftRotate(node);
/* Left Right case*/
if (balance >1 && key > node
    ->left->key){
  node->left = leftRotate(
      node->left);
  return rightRotate(node);}
/* Right Left Case*/
if (balance <-1 && key <node
      ->right->key){
  node->right = rightRotate(
      node->right);
  return leftRotate(node); }
  return node; }
/* get balance factor */
int getBalance(struct Node *N)
 { if (N==NULL)
     return 0;
  return height(N->left) -
      height(N->right);}
/* function to print preorder*/
void preOrder(struct Node *root)
 {  if (root != NULL){
    printf("%d", root->key);
     preOrder(root->left);
     preOrder(root->right); }}
int main()
{ struct Node* root = NULL;
  root = insert(root, 10);
  root = insert(root, 20);
  root = insert(root, 30);
  root = insert(root, 40);
  root = insert(root, 50);
  root = insert(root, 25);
  printf("Preorder traversal of
   AVL tree is");
  preOrder(root);
  return 0; }
```

Time complexity: The rotation operations take constant time as only a few pointers are changed here.

Height of balanced AVL tree is $\log n$, so the time complexity of AVL tree insert is $O(\log n)$.

o/p: Preorder traversal of AVL tree is: 30 20 10 25 40 50



B-TREE

B Tree is a specialized m-way tree that can be widely used for disk access.

A 'B-Tree' of order 'm' can have at most 'm-1' keys and 'm' children.

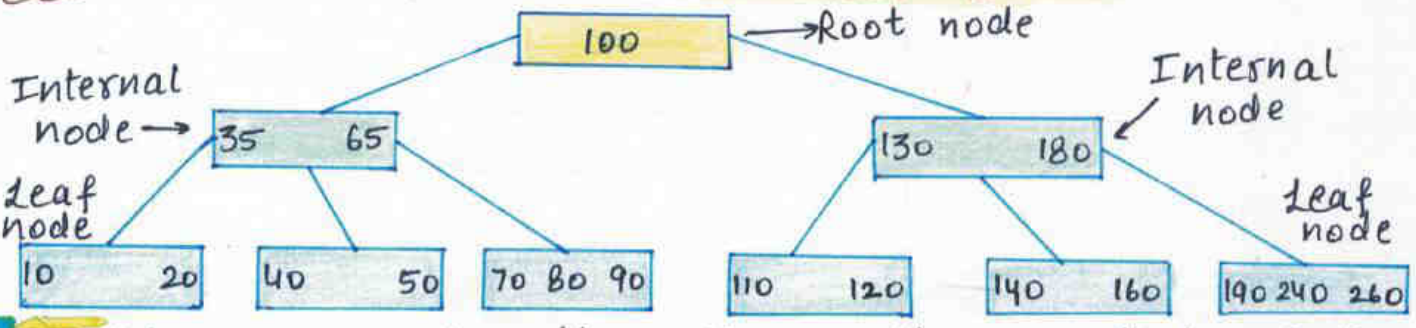
One of the main reason of using B tree is its capab-ility to store large number of keys in a single node and large keys values by keeping height of tree small.

Properties

Following are the properties of B Tree:

1. All leaves are at the same level.
2. A B-tree is defined by the term minimum degree 'm'. The value of m depends upon disk block size.
3. Every node except root must contain at least m-1 keys The root may contain minimum 1 key.
4. All nodes (including root) may contain at most 2m-1 keys.
5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are stored in increasing order. The child b/w two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows & shrinks from the root which is unlike Binary search tree. BST grows downward & shrink from down.
8. Like other BSTs time complexity to search, insert & delete is O(logn).
9. Insertion of a Node in B-Tree happens only at leaf node.

Eg: B-Tree of Minimum Order 5.



We can see in the above diagram that all the leaf nodes are at the same level and all non leaf have no empty sub tree.

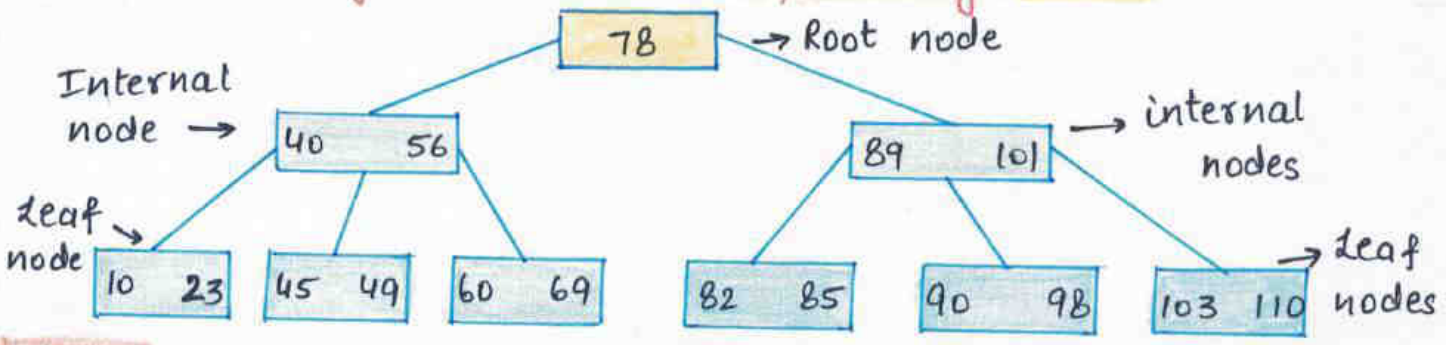
Operations

While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Searching

Searching in B Trees is similar to that in Binary search tree.

Eg: Searching an item 49 in following B tree.



Solution ⇒ Compare item 49 with root node 78. Since 49<78 hence, move to its left subtree.

⇒ since, 40<49<56, traverse right subtree of 40.

⇒ 49>45, move to right. compare 49.

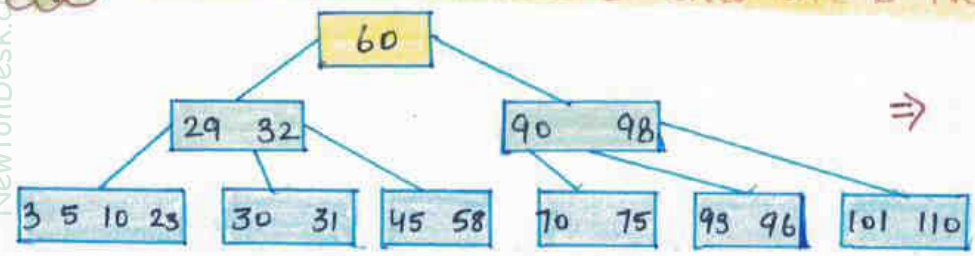
⇒ match found. return.

Inserting

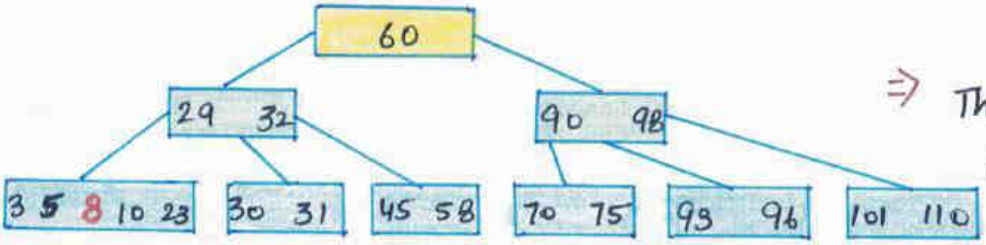
Insertions are done at the leaf node level. The following steps to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contains less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys then, insert the new element in the increasing order of elements.
4. Split the node into the two nodes at the median & push the median element upto its parent node.
5. If the parent node also contain m-1 keys, then split it too using same steps.

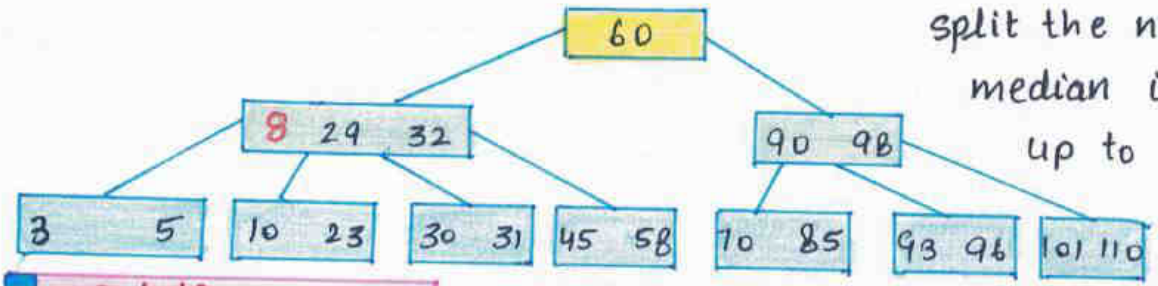
Eg. Insert the node 8 into the B Tree of order 5.



⇒ 8 will be inserted to the right of 5, therefore insert 8.



⇒ The node now contain keys which is greater than (5-1=4) keys. So split the node from the median i.e. 8 & push it up to its parent node.



Deletion

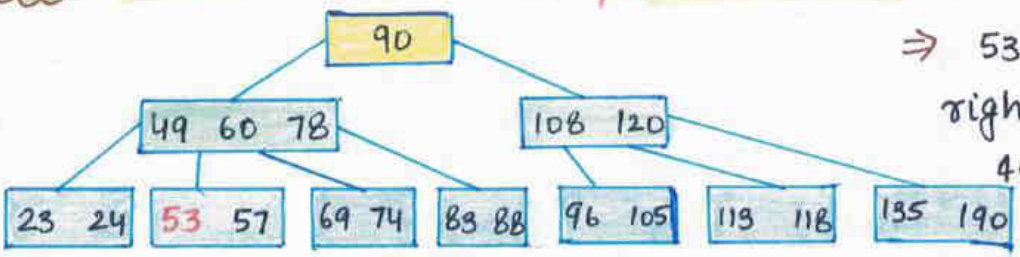
Deletion is also performed at the leaf nodes. The node which is to be deleted can either a leaf node or an internal node.

Following steps are involved in deletion operation-

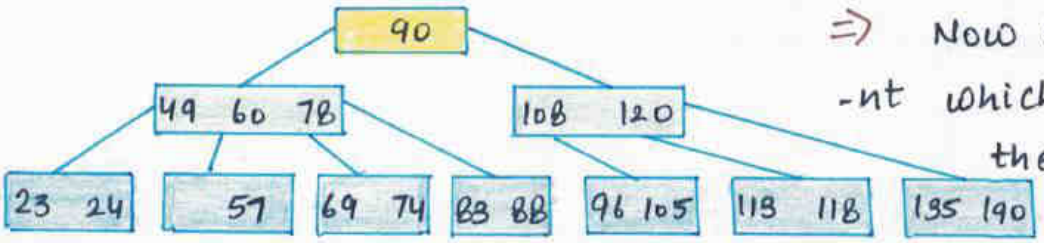
1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node does not contain m/2 keys then complete the keys by taking the element from right or left sibling.
4. If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
5. If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
6. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes & the intervening element of the parent node.
7. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

If the node which is to be deleted is an internal node, then replace the node with its inorder successor or predecessor.

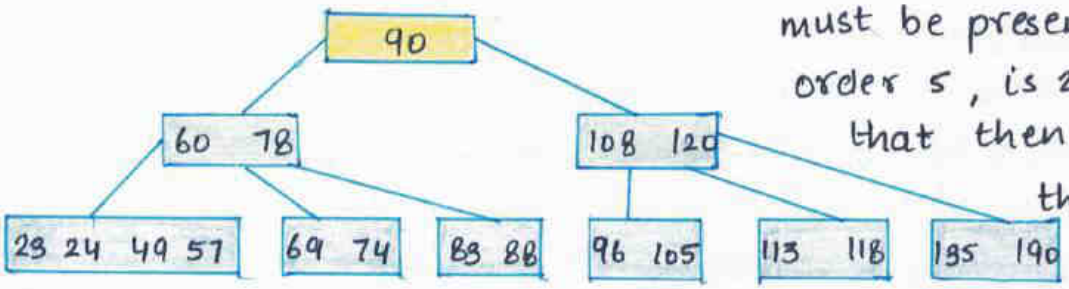
Eg: Delete the node 53 from B tree.



⇒ 53 is present in the right child of element 49. Delete it.



⇒ Now 57 is the only element which is left in the node, the minimum number of elements that must be present in a Btree of order 5, is 2. it is less than that then merge it with the left sibling.



RED-BLACK TREE

The Red-Black tree is a self balancing binary search tree.

AVL tree is also a height balancing binary search but in AVL tree, we do not know how many rotations would be required to balance the tree but in red-black tree, a maximum of 2 rotations are required.

It contains one extra bit that represents either the red or black colour of a node to ensure the balancing of the tree.

The time complexity of searching, insertion and deletion are $\log_2 n$.

SPLAY TREE

The splay tree is also binary search tree in which recently accessed element is placed at the root position of tree by performing some rotation operations.

Here splaying means the recently accessed node & it is a self balancing binary search tree.

It might be a possibility that height of the splay tree is not balanced. i.e. height of both left & right sub trees may differ.

Time complexity of the operations is $\log n$ wherein is the number of nodes.

TREAP

Treap data structure comes from the Tree and Heap data structure.

In heap data structure . both right & left subtrees contain larger keys than the root, i.e. root node contains the lowest value.

In treap data structure . each node has both key & priority where key is derived from the Binary search tree and priority is derived from the heap data structure.

The treap data structure follows two properties—

1. Right child of a node >= current node and left child of a node <= current node (binary tree)
2. Children of any subtree must be greater than the node.

Que What is the maximum height of any AVL tree with 7 nodes. Assume that the height of a tree with a single node is 0.
(A) 2 (B) 3 (C) 4 (D) 5

Solution: $n(h) = n(h-1) + n(h-2) + 1$

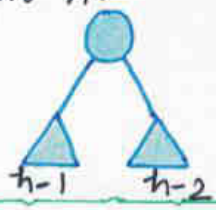
[B]. 3

Minimum no. of nodes in an AVL tree of height h.

$n(0) = 1$ $n(1) = 2$ $n(3) = 1+4+2 = 7$

$n(0) = 1$ $n(1) = 2$ AVL $h = 3$

$h(2) = 1+2+1 = 4$; Max. height → 7



Que Which of the following is true.
(A) Cost of searching an AVL tree is $\theta(\log n)$ but that of binary search tree is $O(n)$.
(B) Cost of searching an AVL tree is $\theta(\log n)$ but that of complete binary tree is $O(n \log n)$.
(C) Cost of searching a binary tree is $O(\log n)$ but that of an AVL tree is $\theta(n)$
(D) Cost of searching an AVL tree is $O(n \log n)$ but that of a binary search tree is $O(n)$.

[A] option

HEAP

A heap is a complete binary tree. and the binary tree is a tree in which the node can have utmost two children.

A complete binary tree is a binary tree in which all the levels except the last level, i.e. leaf node should be com-pletely filled, & all the nodes should be left-justified.

The heap tree is a special balanced binary tree data str-ucture where the root node is compared with its children & arranged accordingly.

There are two types of the heap:

- Min Heap
- Max Heap

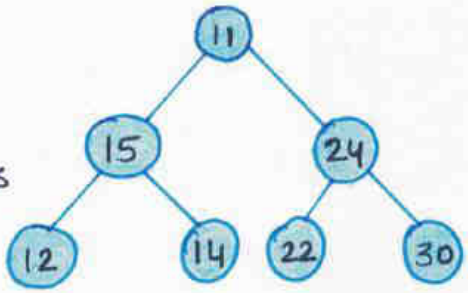
MIN HEAP

The value of the parent node should be less than or equal to either of its children.

The min heap can be defined as, for every node 'i', the value of 'i' is greater than or equal to its parent value exc-ept the root node.

Mathematically it can be defined as:-

```
A[Parent(i)] <= A[i]
```



In this figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

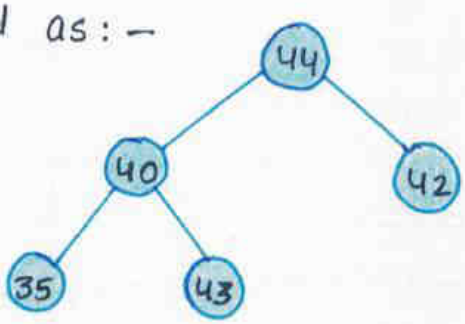
MAX HEAP

The value of the parent node is greater than or equal to its children.

The max heap can be defined as for every node 'i', the value of node 'i' is less than or equal to its parent value except the root node.

Mathematically, it can be defined as:-

```
A[Parent(i)] >= A[i]
```



This figure is a max heap tree as it satisfies the property of max heap.

The total number of comparisons required in the max heap is according to the height of tree.

The height of complete binary tree is always $\log n$; therefore the time complexity would also be $O(\log n)$.

Operations On Heap

The common operations involved using heaps are:-

Heapify

Heapify is the process of creating a heap data structure from a binary tree.

It is used to create a Min-Heap or a Max-Heap.

Eg. Max-Heap using B=[10,20,25,6,12,15,4,16].

Step 1.

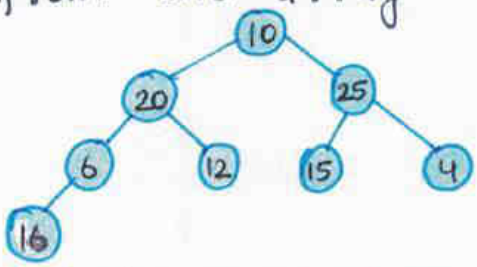
Let the input array be

10	20	25	6	12	15	4	16
0	1	2	3	4	5		

Initial array

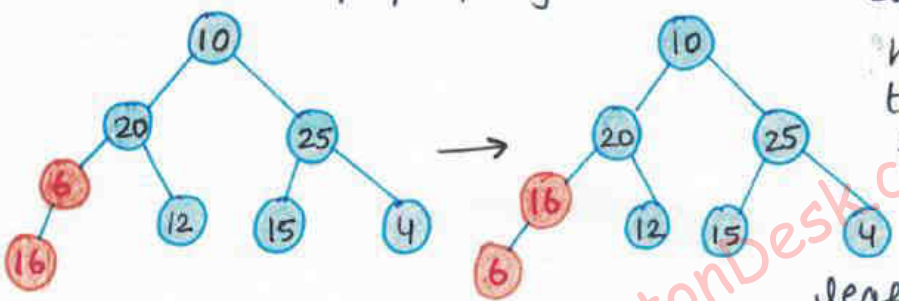
Step 2

Create a complete binary tree from the array.



Step 3

Now we will take a subtree at the lowest level & start checking whether it follows the max-heap property or not.

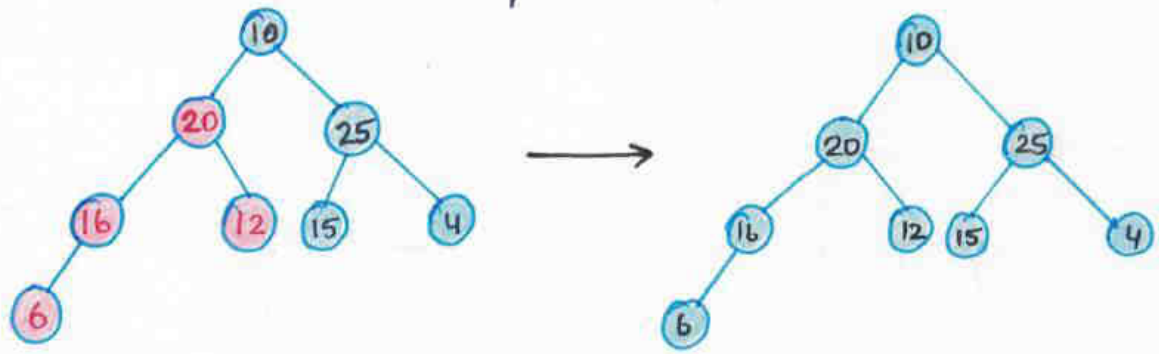


⇒ As we can see the left subtree does not follow max heap property. so we swap the key values between the children node and the parent node.

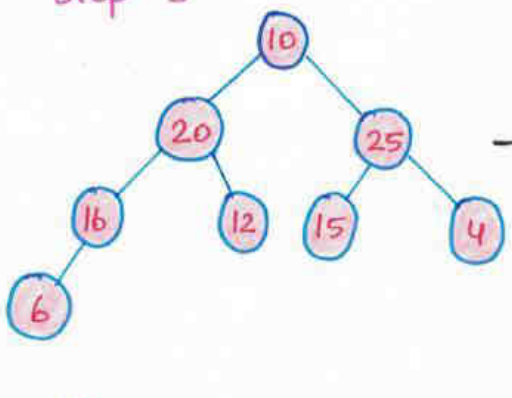
leaf nodes

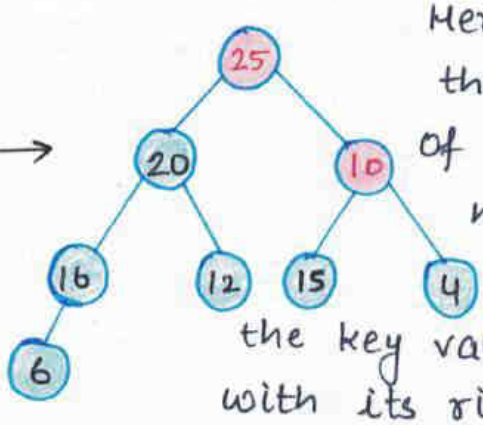
Step 4

Let's continue and examine all the subtrees from the lowest level to the top level:-



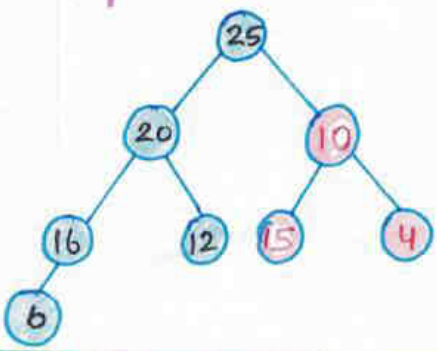
step 5

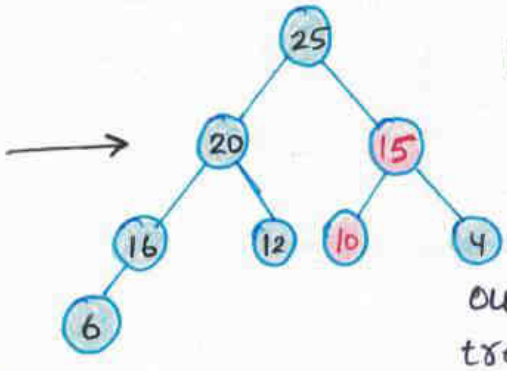




Here, we can see that the key value of the root node is not largest then we swapped the key values of root node with its right children.

step 6





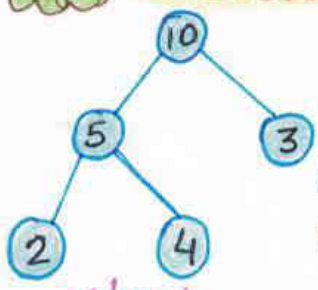
Here, whole tree satisfies the max heapify property, & we will get our final max heap tree.

Insertion

Elements can be inserted to the heap in following steps :-

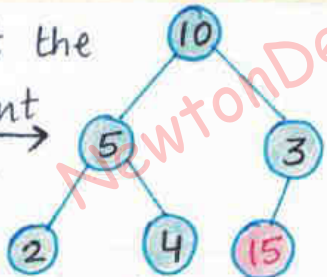
1. First increase the heap size by 1, so that it can store the new element.
2. Insert the new element at the end of the heap.
3. This newly inserted element may distort the properties of heap for its parents. So, in order to keep the properties of heap heapify this newly inserted element.

Eg. Insert a new element 15.



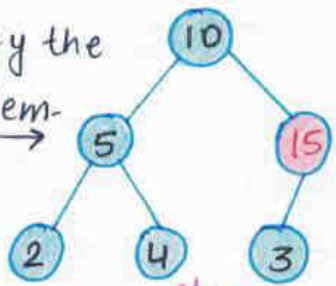
step 1

Insert the element at the end →



step 2

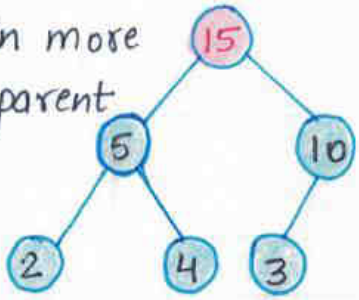
Heapify the new element →



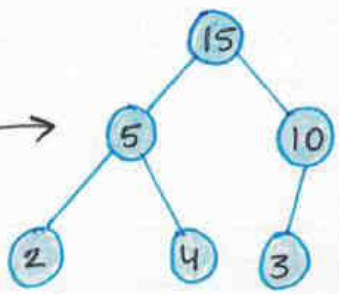
step 3

15 is more than its parent, swap them.

15 is again more than its parent so swap them.



step 4



Final tree

Deletion

The standard deletion operation on Heap is to delete the element present at the root node of the Heap.

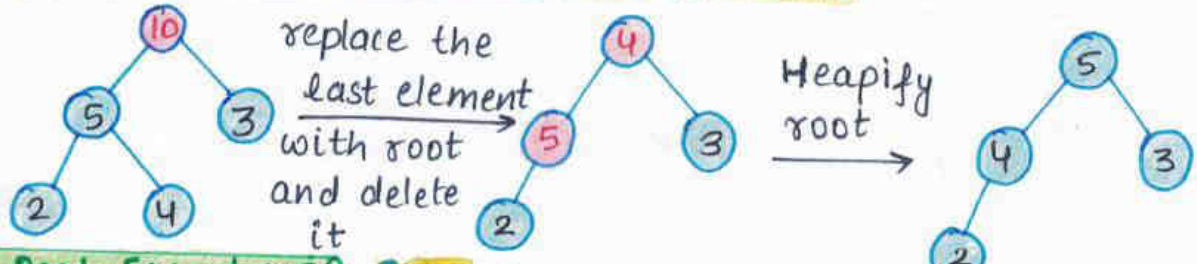
If it is a Max heap, the standard deletion operation will delete the maximum element and if it is a Min heap, it will delete the minimum element.

To delete the intermediary element first replace the root or element to be deleted by last element.

Delete the last element from the heap.

Since, the last element is now placed at the position of the root node. so, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.

Eg: Delete the element root node 10.



Peek [Max/Min]

Peek operation returns the maximum element from Max heap or minimum element from Min heap without deleting the node.

For both Max heap and Min heap → return rootNode;

Extract [Max/Min]

Extract - Max returns the node with maximum value after removing it from a max heap.

Extract - Min returns the node with minimum value after removing it from a min heap.

Eg: Max - Heap Ds.

```
/* Max-Heap data structure */
#include <stdio.h>
int size = 0;
void swap ( int *a, int* b)
{ int temp = *b;
   *b = *a;
   *a = temp; }
void heapify ( int array[ ],
  int size, int i) {
  if (size == 1)
printf ("Single element in the heap");
  else {
    int largest = i;
    int l = 2* i + 1;
    int r = 2* i + 2;
    if ( l < size && array[l] >
     array [largest ])
     largest = l;
```

```
if (r< size && array[r] >
    array[largest])
  largest = r;
if (largest != i) {
  swap (&array[i], &array[
    largest]);
  heapify (array, size, largest);
    }}}
/* insert new element */
void insert (int array[],
      int newNum) {
  if (size == 0) {
    array[0] = newNum;
    size += 1; }
  else {
    array[size] = newNum;
    size += 1;
    for {int i= size/2 -1; i>=0;
        i--) {
      heapify (array, size, i); }}}
/* delete the element */
void deleteRoot (int array[],
      int num) {
    int i;
  for(i=0; i < size; i++)
  {  if (num == array[i])
      break ; }
    swap (&array[i], &array[size-1]);
    size -= 1;
  for (int i= size/2 - 1; i>=0; i--)
    { heapify (array, size, i); }}
void printArray (int array[ ],
      int size) {
for (int i=0; i<size; ++i) {
  printf ("%d", array[i]);
  printf ("\n"); }}
int main ()
{ int array[10];
  insert (array, 3);
  insert (array, 4);
  insert (array, 9);
  insert (array, 5);
  insert (array, 2);
  printf ("Max-Heap array");
  printArray (array, size);
  deleteRoot (array, 4);
  printf ("After deleting an ele-
    -ment:");
  printArray (array, size); }
```

o/p: Max-Heap array: 9,5,4,3,2
After deleting an element: 9,5,2,3

HEAP SORT

Heapsort is a popular and efficient sorting algorithm.

The concept of heap sort is to eliminate (elements) one by one from the heap part of the list and then insert them into the sorted part of the list.

Heap-sort is the in-place sorting algorithm.

In heap sort, basically there are two phases involved in the sorting of elements.

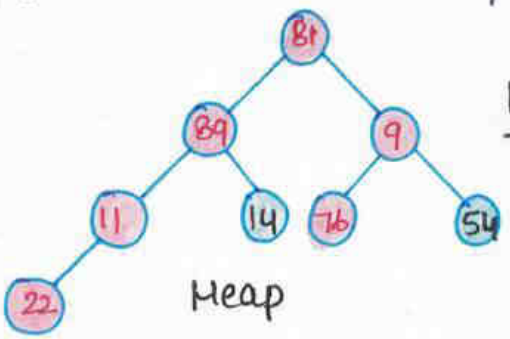
1. The first step includes the creation of heap by adjusting the elements of the array.

2. After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, & then store the heap structure with the remaining elements.

Eg. Sorting an unsorted array using Heap sort.

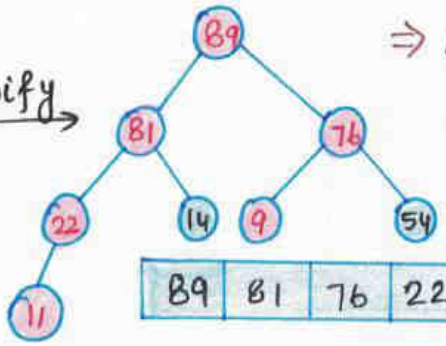
81	89	9	11	14	76	54	22

First, we have to construct a heap from the given array & convert it into max heap.



Heap

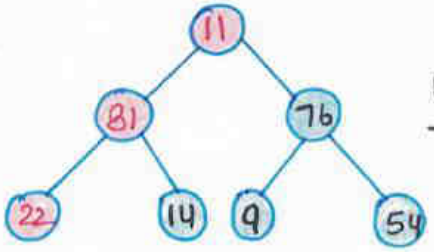
Heapify



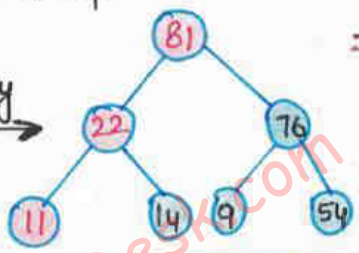
⇒ After converting the given heap into max heap, the array elements are-

89	81	76	22	14	9	54	11

⇒ Next, we have to delete the root element (89) from the max heap. To delete this node, swap it with the last node (11). After deleting the root element, we again have to heapify it to convert into Max heap.



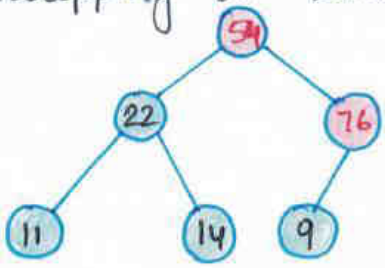
Heapify



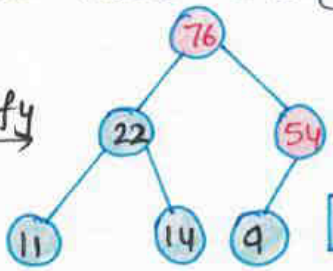
⇒ After swapping the array element (89) with (11) Elements of array are-

81	22	76	11	14	9	54	89

⇒ Next, we have to delete element (81) from max heap by swapping it with last node . i.e. (54).



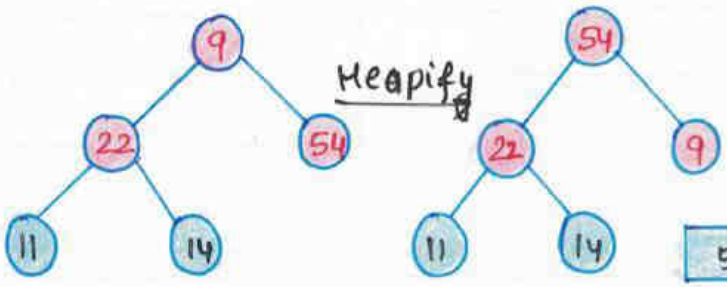
Heapify



⇒ After swapping, the elements of array are:-

76	22	54	11	14	9	81	89

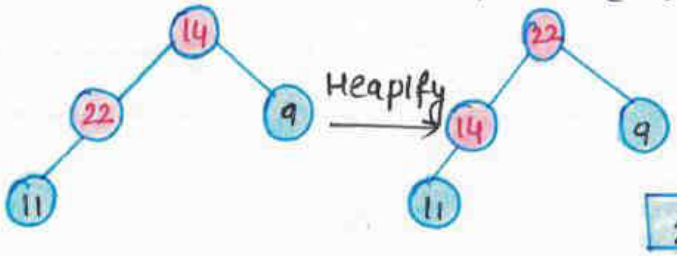
⇒ Next, we have to delete the root element (76) from max heap. by swapping it with last node, i.e. (9).



⇒ After swapping the array element (76) with (9) the elements of array are-

54	22	9	11	14	76	81	89

⇒ Next we have to delete the root element (54) by swapping it with the last node, i.e. (14).



⇒ After swapping the element (54) with (14), the elements of array are-

22	14	9	11	54	76	81	89

⇒ Next, we have to delete the root element (22) by swapping it with the last node, i.e. (11).

⇒ after swapping elements of array are-

14	11	9	22	54	76	81	89

⇒ Next, we have to delete the root element (14) by swapping it with the last node, i.e. (9).

⇒ After swapping array elements-

11	9	14	22	54	76	81	89

⇒ Next, we have to delete the root element (11) by swapping it with the last node, i.e. (9).

⇒ array elements are-

9	11	14	22	54	76	81	89

⇒ Now, heap has only one element left. After deleting it, heap will be empty.

9 ⟶ empty

⇒ After completion of sorting, the array elements are-

9	11	14	22	54	76	81	89

⇒ Array is completely sorted.

Complexity 👉 The best-case time complexity of heap sort is O(n log n), it occurs when no sorting required.

👉 The average-case time complexity of heap sort is O(n log n), it occurs when the array elements are in jumbled that is not properly ascending or descending.

The worst time complexity of heap sort is O(nlogn), it occurs when the array elements are required to be sorted in reverse order.

The space complexity of heap sort is O(1).

Eg: Implementation of Heap Sort.

```
#include <stdio.h>
/*function to heapify tree*/
void heapify (int a[], int n, int i)
{ int largest= i;  /* root */
  int left = 2*i+1; /* left child*/
  int right = 2*i+2; /*right child*/
  if (left < n && a[left] > a [largest])
      largest = left;
  if (right <n && a[right]>a[largest])
      largest = right;
  if ( largest != i ) {
      temp = a[i];
    a[i] = a [largest ];
   a [largest] = temp;
    heapify ( a, n, largest);}}
/* function to implement the
  heap sort */
void heapSort ( int a[], int n)
 { for(int i= n/2-1; i>=0; i--)
      heapify (a, n, i);
   for( int i = n-1; i>=0; i--){
      temp = a[0];
      a[0] = a[i];
      a[i] = temp;
      heapify (a, i, 0); }}
/* function to print array elem-
-ents*/
void printArr ( int arr[], int n)
 { for ( int i=0; i<n; ++i ){
   printf ("%d", arr[ ]);
   printf (" \n "); } }
int main ( ) {
  int a[ ] ={48,10,23,43,28,26,1};
int n = sizeof (a)/sizeof(a[0]);
printf(" Before sorting array
    elements are :\n");
  printArr (a, n) ;
  heapSort (a, n) ;
printf(" After sorting array
    elements are:\n");
  printArr (a, n);
  return 0; }
```

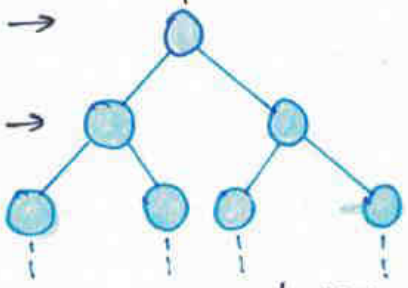
O/P: Before sorting array elements are- 48 10 23 43 28 26 1
After sorting array elements are- 1 10 23 26 28 43 48

Que: In a heap with n elements with the smallest element at the root, the 7th smallest element can be found in time – (A) θ(nlogn) (B) θ(n) (C) θ(logn) (D) θ(1)

[D] θ(1)

Solution: Min-heap

1st smallest →
2nd smallest →
3rd smallest →



⇒ The 7th smallest element must be in first 7 levels. Total no. of nodes in any Binary Heap in first 7th levels is at most 1+2+4+8+16+32+64 which is const.

⇒ Therefore we can always find 7th smallest element in O(1) time.

Que Suppose there are [logn] sorted lists of [n/logn] elements each. The time complexity of producing a sorted list of all these elements is –

(A) O(n log logn) (B) Θ(nlogn) (C) Ω(nlogn) (D) Ω(n3/2)

Solution: We can merge x arrays of each size y in O(xy* logy) time using min heap.

x = logn → We get O(n/log n* Logn* loglogn) which is O(nloglogn).

[A] O(n Log Logn)

Que The number of possible min heaps containing each value from {1,2,3,4,5,6,7} exactly once is ——

(A) 80 (B) 8 (C) 20 (D) 210

Solution: Set minimum element as root, i.e. (1), now 6 are remaining & left subtree will have 3 elements, each left subtree combination can be permuted in 2! ways –

Total ways to design min heap with 7 elements –

$^6C_3 * 2! * 2! = 20 * 2 * 2 = 80$

[A] 80

Que An operator delete(i) for a binary heap data structure is to be designed to delete the item in the ith node. Assume that the heap is implemented in an array and i refers to the ith index of the array. If the heap tree has depth d (no. of edges on the path from the root to the farthest leaf) then what is the time complexity to refix the heap efficiently, after the removal of the element.

(A) $O(1)$ (B) $O(d)$ but not $O(1)$

(C) $O(2^d)$ but not $O(d)$ (D) $O(d2^d)$ but not $O(2^d)$

Solution: For this, we have to slightly tweak the **[B] option** delete_min() operation of heap data structure to implement the delete(i) operation.

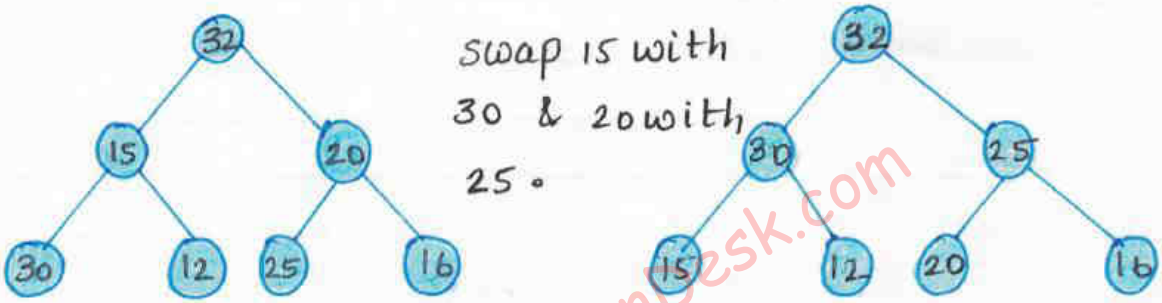
⇒ This idea is to empty the spot in the array at the index i and replace it with the last leaf in the heap, decreme. the heap size and now starting from the current position i, shift it up in case newly replaced item is greater than the parent of old item.

⇒ If it's not greater than the parent, then percolate it down by comparing with the child's value.

⇒ The newly added item can percolate up/down a maximum of d items which is the the depth of heap.

⇒ Thus we can say that complexity of delete(i) would be $O(d)$ but not $O(1)$.

Que The elements 32, 15, 20, 30, 12, 25, 16 are inserted one by one in the given order into a max heap. Resultant max heap is_



HASH TABLE

Hash table is one of the most important data structure that uses a special function known as a hash function that maps a given value with a key to access the elements faster.

A Hash table is a data structure that stores some information has basically two main components, i.e. key & value.

The hash table can be implemented with the help of an associative array.

The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

It is a data structure in which insertion and search operations are very fast irrespective of the size of the data.

Eg: Suppose, the key value is John & the value is phone number, so when we pass the key value in hash function-

Hash (key) = index; When we pass the key in the hash function, then it gives the index.

Hash (john) = 3; This example adds the john at index 3.

Hash function assigns each value with a unique key.

Sometimes hash tables uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.

HASHING

Hashing is one of the searching technique that uses a constant time.

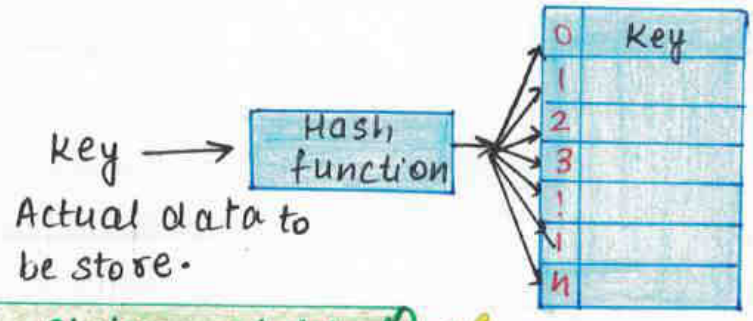
The time complexity in hasing is O(1).

Linear Search & Binary Search, both searching techniques, depends upon the number of elements, worst time complexity are O(n) & O(logn) respectively.

In hashing technique, the hash table & hash function are used. Using hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored.

It can be written as –



There are three ways of calculating the hash function –

- Division Method
- Folding Method
- Mid square Method

Divison Method

This is the easiest method to create a hash function. The hash function can be described as –

$h(K) = k \bmod n$.

Here, h(k) is the hash value obtained by dividing the key value k by size of hash table n using the remainder.

It is best that n is a prime number as that makes sure the keys are distributed with more uniformity.

Eg. Example of division Method.

$k = 1276$ $\quad h(1276) = 1276 \bmod 10$

$n = 10$ $\quad = 6$; Hash value obtained is 6.

A disadvantage of the division method is that consecutive keys map to consecutive hash values in the hash table. This leads to a poor performance.

Folding Method

The key k is partitioned into a number of parts $k_1, k_2 \dots k_n$ where each part except possibly the last, has the same number of digits as the required address.

Then the parts are added together, ignoring the last carry.

$H(K) = k^1 + k^2 + \dots + k^n$

Mid-Square Method

The key k is squared.

The function H is defined by – $H(K) = L$

Here L is obtained by deleting digits from both ends of k^2. We emphasize that the same position of k^2 must be used for all the keys.

Eg. $k = 50$ $\quad h(50) = 50$

$k * k = 2500$ $\quad$ The hash value obtained is 50.

Eg. Company has 68 employees, and each is assigned a unique four-digit employee number. Suppose L consist of 2-digit addresses: 00, 01 & 02 ... 99. We apply the above hash functions to each of the following employee numbers: 3205, 7148, 2345

(a) Division method: choose a prime number m close to 99, such as m = 97, then H(3205) = 4; H(7148) = 67, H(2345) = 17. That is dividing 3205 by 17 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, dividing 2345 by 97 gives a remainder of 17.

(b) Mid-Square Method:

k =	3205	7148	2345
k^2 =	10272025	51093904	5499025
h(k) =	72	93	99

Observe that forth & fifth digits, counting from right are choosen for hash address.

(c) Folding Method: Divide the key k into 2 parts and adding yields the following the hash address:

H(3205) = 32+50 = 82 H(7148) = 71+84 = 55

H(2345) = 23+45 = 68.

Collision

When the two different values have the same value, then the problem occurs between the two values, known as a collision.

When the hash function generates the same index for multiple keys, there will be a conflict, to resolve the hash collision using following techniques.

- Collision resolution by chaining
- Open Addressing: Linear / Quadratic Probing & Double Hashing

These techniques are also known as:

- Open Hashing: It is also known as closed addressing.
- Closed Hashing: It is also known as open addressing.

Open Hashing

In open hashing, one of the methods used to resolve the collision is known as a chaining method.

Time complexity for searching is O(n).

Maximum chain length (in general) is n (no. of keys) all keys goes to same slot.

Eg: Suppose we have a list of key values → A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, & h(k) = 2k+3.

Solution: In this case, we cannot directly use h(k) = k%m as h(k) = 2k+3.

⇒ The index of key value '3' is: index = h(3) = (2(3)+3)%10 = 9
The value 3 would be stored at the index 9.

⇒ The index of key value '9' is: h(9) = (2(9)+3)%10 = 1
The value 9 would be stored at the index 1.

⇒ The index of key value '6' is: h(6) = (2(6)+3)%10 = 5
The value 6 would be stored at the index 5.

⇒ The index of key value '11' is: h(11) = (2(11)+3)%10 = 5
The value 11 would be stored at the index 5. Now, we have two values (6, 11) stored at the same index, i.e. 5, this leads the collision problem, so we will use the chaining method to avoid the collosion.

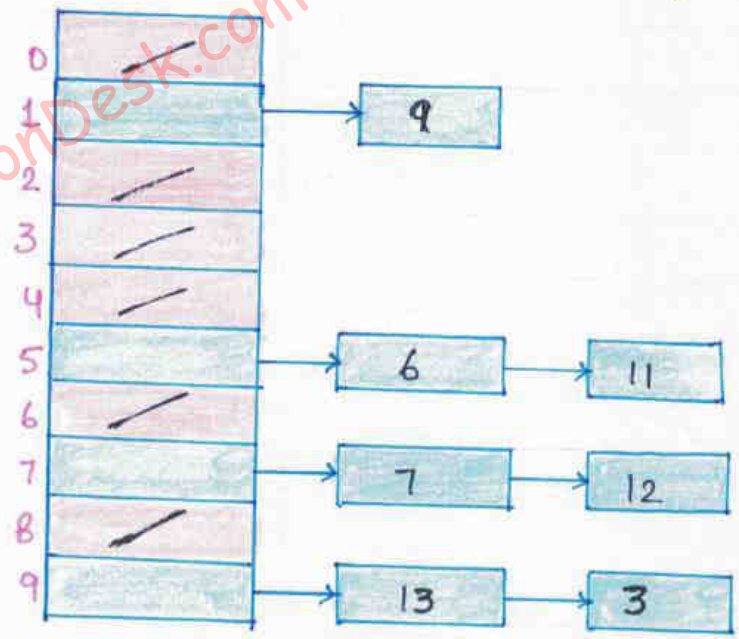
⇒ We will create one more list and add the value 11 to this list. After the creation of the new list, the newly created list be linked to the list having value 6.

⇒ The index of key value '13' is: h(13) = (2(13)+3)%10 = 9
Like above for same index we create new list & add the elements to that.

⇒ The calculated index value associated with each key value is shown below:

Key Location (L)

3 → ((2*3)+3)%10 = 9
2 → ((2*2)+3)%10 = 7
9 → ((2*9)+3)%10 = 1
6 → ((2*6)+3)%10 = 5
11 → ((2*11)+3)%10 = 5
13 → ((2*13)+3)%10 = 9
7 → ((2*7)+3)%10 = 7
12 → ((2*12)+3)%10 = 7



Closed Hashing

In closed hashing, three techniques are used to resolve the collision:

- Linear Probing
- Quadratic Probing
- Double Hashing

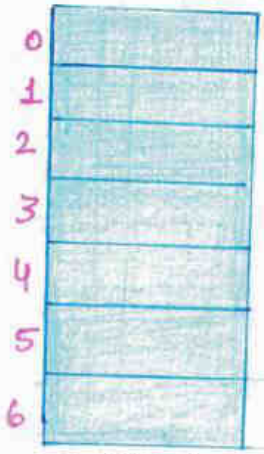
Linear Probing

Linear probing is one of the forms of open addressing.

As we know that each cell in the hash table contains a key-value pair, so when the collision occurs by mapping a new-key to the cell already occupied by another key, then linear probing technique searches for the closet free locations and adds a new key to that empty cell.

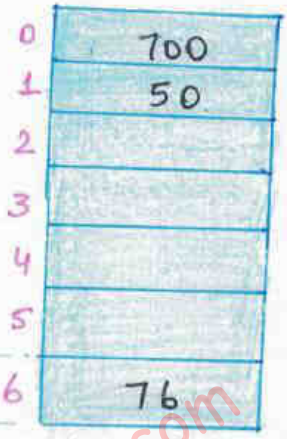
In this case, searching is performed sequentially, st-arting from the position where the collision occurs till the empty cell is not found.

Eg: Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.



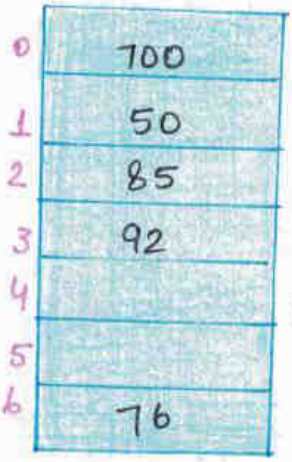
Initial empty table

insert 50

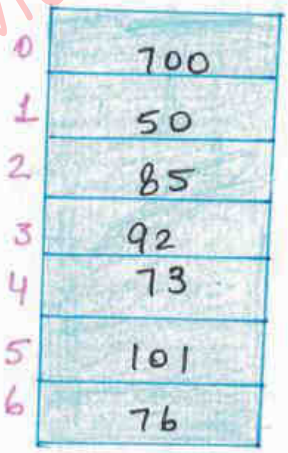


insert 700 & 76

insert 85 at next slot



Insert 92, collision occurs as 50 is there at index.



insert 73 & 101

There are two challanges occured in Linear Probing:

1. One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.

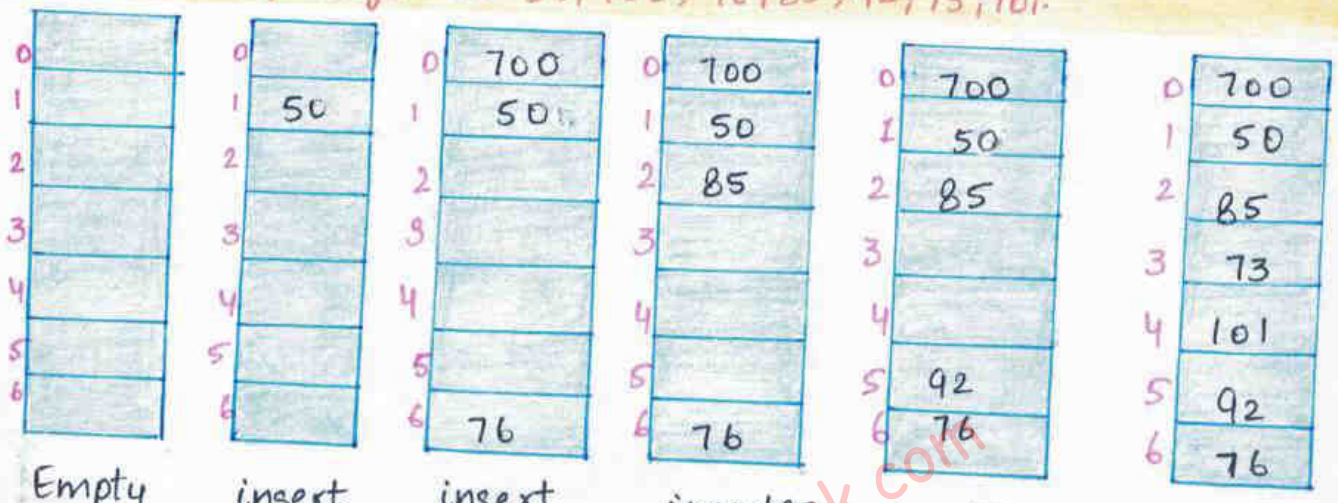
2. Secondary clustering is less severe, two records only have the same collision chain if their initial position is same.

Quadratic Probing

In case of linear probing, searching is performed linearly, but quadratic probing is an open addressing technique that uses quadratic polynomial for searching until a empty slot is found.

It can also be defined as that it allows insertion ki at first free location from $(u+i^2)\%m$ where $i=0$ to $m-1$.

Eg: Let us consider a simple hash function as 'key mod 7' & sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Empty table | insert 50 | insert 700 & 76 | insert 85 collision occurs then insert at $1+1*1$. | insert 92 coll. occurs at 1 insert at $1+2*2$. | insert 73 & 101.

Double Hashing

Double hashing is an open addressing technique which is used to avoid collisions.

When the collision occurs then this technique uses the secondary hash of the key.

It uses one hash value as an index to move forward until the empty location is found.

In double hashing, two hash functions are used.

Double hashing can be done using : (hash1(key) + i*hash2(key)) %. Table-SIZE. hash1() & hash2() are hash functions & Table_size is size of hash table.

The advantage of double hashing is that it is one of the best form of probing, producing a uniform distribution of records throughout a hash table.

This technique does not yield any clusters.

In double hashing first hash function is hash1(key) = key %. Table-SIZE & second hash function is hash2(key) = PRIME-(key %. PRIME) where PRIME is a prime smaller than the TABLE-SIZE.

A good second Hash function is that it must never evaluate to zero & it must make sure that all cells can be probed.

Eg: Hash1(key) = key %. 13 & Hash2(key) = 7-(key %. 7)

Hash1(19) = 19 %. 13 = 6

Hash1(27) = 27 %. 13 = 1

Hash1(36) = 36 %. 13 = 10

Hash1(10) = 10 %. 13 = 10

Hash2(10) = 7-(10 %. 7) = 4

((Hash1(10) + 1*Hash2(10)) %. 13 = 1 — Collision

((Hash1(10) + 2*Hash2(10)) %. 13 = 5

Eg: Implementation of hash table

```
#include<stdio.h>
#include<stdlib.h>
struct set {
  int key; int data;};
struct set *array;
int capacity = 10;
int size = 0;
int hashFunction(int key){
return (key %. capacity);}
int checkPrime(int n){
  int i;
  if(n==1||n==0)
    return 0;
  for(i=2; i<n/2; i++){
  if(n %. i==0)
    return 0;}
    return 1;}
int getPrime(int n){
  if(n %. 2==0)
    n++;
while(!checkPrime(n))
  {n+=2;}
  return n;}
void init_array(){
  capacity = getPrime(capacity);
```

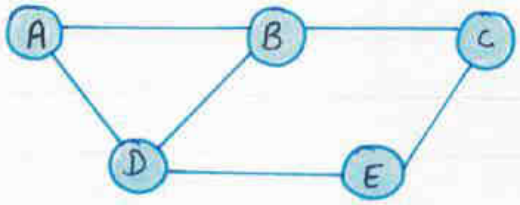
```
array = (struct set*) malloc (
  capacity * sizeof (struct set));
for (int i = 0; i < capacity ; i++)
{ array[i]. key = 0;
  array [i] . data = 0; }}
void insert ( int key , intdata )
{ int index = hashFunction ( key);
  if (array [index].data == 0 ){
   array[index].key = key;
   array [index]. data = data;
   size ++;
   printf(" \n key (%d) has been
inserted \n"; key ); }
  else if (array [index ].key == key)
   { array [ index]. data = data;
   }
   else {
   printf(" \n collision occured");
    }}
void remove_element (int key)
{  int index = hashFunction (key);
  if (array [index].data == 0)
  {
  printf (" \n This key does not
  exist \n "); }
  else {
  array[index]. key = 0;
  array[ index]. data = 0;
  size--;
 printf(" \n key (%d) has been
  removed \n"; key); }}
void display ( )
{  int i;
  for (i = 0; i < capacity; i++)
  { if (array[ i ].data == 0)
   {  printf (" \n array[%d]: "; i);}
  else {
printf (" \n key: %d array [%d]:
%d\t", array[ ].key , i, array[ ].
   data ); }}}
int size_of_hashtable ( )
{ return size; }
   int main ( ){
int choice, key, data, n;
 int c = 0;
 init_array ( );
do {
printf (" 1. Insert item","\n2.
remove item","\n3. check size",
"\n 4. Display ", "Enter choice:");
scanf ("%d", & choice);
switch (choice)
{ case 1:
  printf (" Enter key -: ");
  scanf ("%d", & key);
 printf(" Enter data -: ");
 scanf (" %d", & data );
 insert (key , data );
  break ;
 case 2:
printf (" Enter the key to
    delete -: " );
scanf ("%d"  , & key) ;
 remove - element (key );
  break ;
```

```
case 3:
  n= size-of-hashtable ( );
printf("size of Hash table
   is -: %d", n);
  break ;
case 4:
    display( )
    break;
   default;
  printf ("Invalid Input ");}
printf( " Do you want to cont-
 -inue (press 1 for yes): ");
scanf ( "%d", &c);
 } }
while (c==1);
}
```

```
O/P: 1. Insert Item.
2. Remove Item
3. Check size
4. Display hash table
Please enter your choice: 1
Enter key-: 23
Enter data: 56
Key (23) has been inser-
-ted.
```

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.
- A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.
- A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.
- A graph G(V,E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B, C), (C,E), (E,D), (D,B), (D,A)) is shown in this figure →



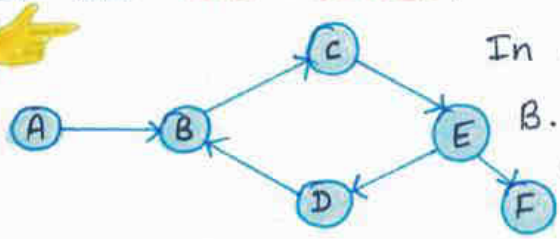
- There are two types of graph-
  - Directed Graph
  - Undirected graph

Directed Graph

When a graph has an ordered pair of vertexes, it is called as a directed graph.

The edges of a graph represent a specific direction from one vertex to another.

When there is an edge representation as (V_1, V_2), the direction is from V_1 to V_2. The first element V_1 is the initial node or the start vertex & the second element V_2 is the terminal node or the end vertex.



In this graph, vertex A connects to vertex B. A is the initial node and node B is the terminal node & the direction is from A to B, not B to A.

Similarly, the connected vertexes have specific directions.

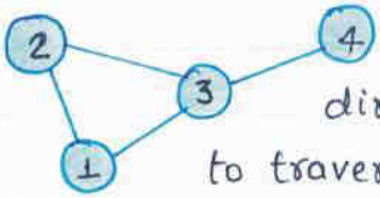
set of vertices (V) – {A, B, C, D, E, F}

set of edges (E) – {(A,B), (B,C), (C,E), (E,D), (D,E), (E,F)}

Undirected Graph

When a graph has an unordered pair of vertexes, it is an undirected graph, there is no specific direction to represent the edges.

The vertexes connect together by undirected arcs, which are edges without arrows.



This is an undirected graph. There is no direction in any of the edges. It is possible to traverse from 2 to 3, 3 to 2, 1 to 3, 3 to 1 etc.

Set of vertices (V) = {1, 2, 3, 4}

Set of edges (E) = {(1,2), (2,1), (2,3), (3,2), (1,3), (3,1), (3,4), (4,3)}

Graph Terminology

Following terminology is used in Graphs –

Path

A path can be defined as sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path A path can be called as closed path if the initial node is same as terminal node. A path will be closed if $V_0 = V_N$.

Simple path If all the nodes of the graph are distinct with an exception $V_0 = V_N$, then such path P is called as closed simple path.

Cycle A cycle can be defined as the path which has no repeated edges or vertices except first & last vertices.

Connected Graph A connected graph is one in which some path exists between every two vertices (u, v) in V.

There are no isolated nodes in connected graph.

Complete Graph A complete graph is the one in which every node is connected with all other nodes.

A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph.

Weighted Graph In a weighted graph, each edge is assigned with some data such as length or weight.

The weight of an edge e can be given as $w(e)$ which must be a positive (+) value indicating the cost of traversing the edge.

Digraph A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in specified direction.

Loop An edge that is associated with the similar end points can be called as loop.

Adjacent Nodes If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of Node A degree of a node is the number of edges that are connected with that node.

A node with degree 0 is called as isolated node.

Graph Representation

By graph representation, we mean the technique to be used to store some graph into the computer's memory.

There are two ways to store Graphs –

- Sequential representation (Adjacency matrix representation)
- Linked List representation (Adjacency list representation)

Sequential Representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph.

We can use an adjacency matrix to represent the directed graph, undirected graph, weighted directed graph and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry Aij in the adjacency matrix representation of an undirected graph G will be '1' if an edge exists between Vi and Vj.

If an undirected Graph G consists of n vertices, then the adjacency matrix for that graph is nxn, & the

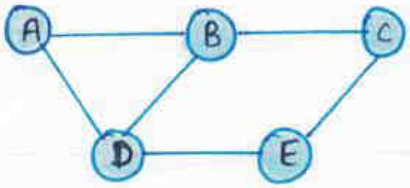
matrix $A = [a_{ij}] =$ $a_{ij} = 1$ { if path exists from Vi to Vj }

$a_{ij} = 0$ { otherwise }

It means that, '0' represents that there is no association exists between the nodes, whereas '1' represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be '0'.

Following is the adjacency matrix representation of an undirected graph –



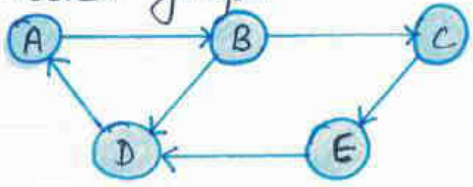
Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency matrix

In this, an image shows the mapping among the vertices (A, B, C, D, E) & is represented by adjacency matrix.

Following is the adjacency matrix representation for directed graph –



Directed Graph

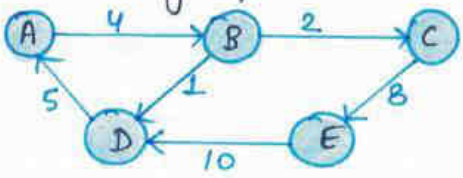
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

In this graph, no self-loops present, so the diagonal entries of the adjacent matrix are 0.

It is similar to an adjacency matrix representation of weighted directed graph with direct graph except that instead of using the '1' for the existence of path, here we have to use weight associated with the edge.

Following is the adjacency matrix representation of weighted directed graph –



Weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

In this graph, the non-zero values are replaced by the actual weight assigned to the edges.

An adjacency matrix can be used when the graph is dense and a number of edges are large.

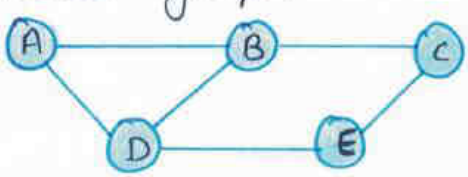
It is advantageous to use an adjacency matrix, but it consumes more space, even if graph is sparse, the matrix still consumes the same space.

Linked List Representation

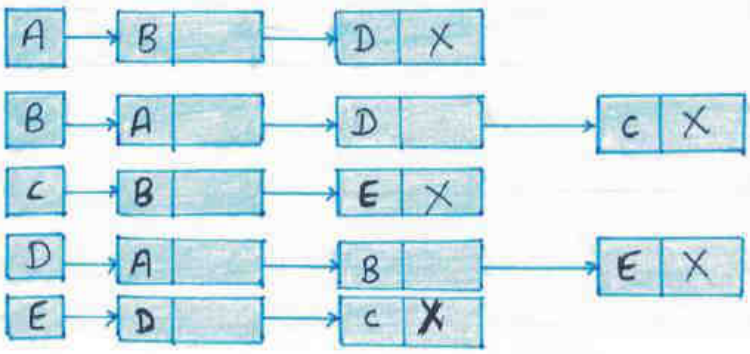
An adjacency list is used in the linked list representation to store the graph.

It is efficient in terms of storage as we have to store the values for edges.

Following is the adjacency list representation of an undirected graph –



undirected Graph



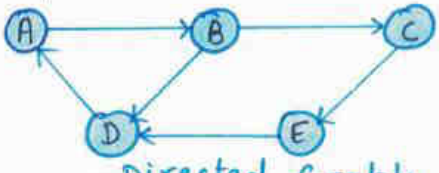
Adjacency List

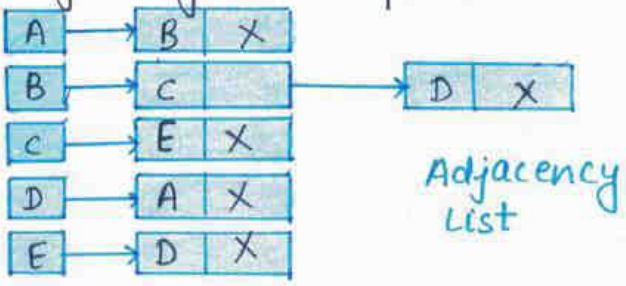
An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node.

If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

The sum of the lengths of adjacency lists is equal to twice the no. of edges present in undirected graph.

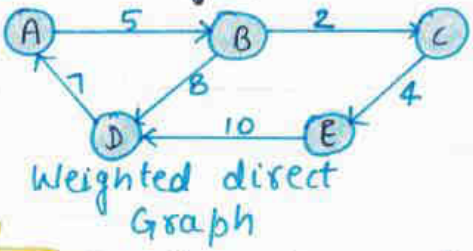
Following is the adjacency list representation of directed graph -

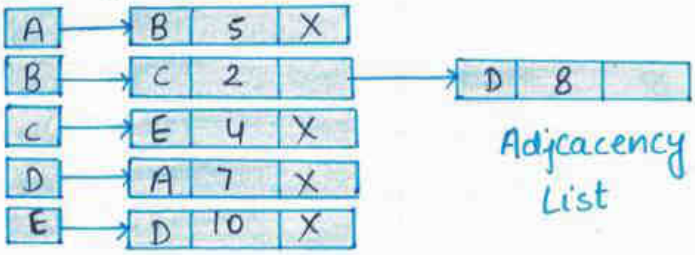




for a directed graph, the sum of lengths of adjacency list is equal to no. of edges present in graph.

Following is the adjacency list representation of weighted directed graph -





In the case of a weighted directed graph, each node contains an extra field that is called weight of the node.

In an adjacency list, it is easy to add a vertex, because of using the linked list, it also saves space.

Eg: Adjacency matrix representation of Graph.

```
#include <stdio.h>
# define V4   no. of vertices
  function to initialize the
  matrix to zero
void init ( int arr[][V] ){
 int i, j;
 for (i=0; i<V; i++)
 for (j=0; j<V; j++)
      arr[i][j] = 0; }
/* function to add edges to
  the graph */
void insertEdge ( int arr[][V],
  int i, int j ){
   arr[i][j] = 1;
   arr[j][i] = 1; }
```

```
/* function to print the matrix
 elements */
void printAdjMatrix ( int arr[]
    [V] ) {
  int i, j;
  for (i=0; i<V; i++) {
  printf ("%d", i);
  for (j=0; j<V; j++){
  printf ("%d ", arr[i][j]); }
  printf ("\n"); } }
int main ( ) {
  int adjMatrix[V][V];
  init(adjMatrix);
 insertEdge (adjMatrix, 0, 1);
insertEdge (adjMatrix, 0, 2);
insertEdge (adjMatrix, 1, 2);
insertEdge (adjMatrix, 2, 0);
insertEdge (adjMatrix, 2, 3);
printAdjMatrix (adjMatrix);
 return 0; }
```

```
o/p:
0 : 0 1 1 0
1 : 1 0 1 0
2 : 1 1 0 1
3 : 0 0 1 0
```

Eg: Adjacency list representation of undirected Graph.

```
#include <stdio.h>
# include <stdlib.h>
/* structure to represent a node
 of adjacency list */
 struct AdjNode {
   int dest; struct AdjNode *
   next; };
/* structure to represent an adja-
 -cency list */
 struct AdjList { struct AdjNode*
   head; };
/* structure to represent the
 graph */
 struct Graph { int V;
   struct AdjList* array; };
struct AdjNode* newAdjNode
  ( int dest ) {
  struct AdjNode* newNode =
  ( struct Graph*) malloc (
   sizeof (struct Graph));
   newNode->dest = dest;
   newNode->next = NULL;
return newNode; }
struct Graph* createGraph (int V){
 struct Graph* graph = (struct Graph*
   malloc ( sizeof( struct Graph));
  graph->V = V;
  graph->array = ( struct AdjList*)
  malloc ( V* sizeof( struct AdjList));
/* initialize each adjacency list
 as empty by making head */
    int i;
 for(i=0; i<V; ++i)
 graph->array[i].head = NULL;
  return graph; }
/* function to add an edge to
 an undirected graph */
```

```
void addEdge (struct Graph*
 graph, int src, int dest) {
/* Add an edge from src to
 dest. node is added at the
 beginning */
struct AdjNode* check = NULL;
struct AdjNode *newNode = newAdj
        -Node (dest);
 if (graph->array [src].head
   == NULL) {
  newNode->next = graph->array
     [src]. head;
  graph->array [src].head = newNode;
 }
 else {
      check = graph->array [src].
                  head;
while (check->next != NULL) {
       check = check->next; }
      check->next = newNode;
   }
/* since graph is undirected, add
 an edge from dest to src */
  newNode = newAdjNode(src);
  if (graph->array [dest]. head
    == NULL) {
    newNode->next = graph->array
     [dest]. head;
   graph->array[dest].head = newNode;
  }
  else {
    check = graph->array[dest]. head;
  while (check->next != NULL) {
      check = check-> next; }
    check->next = newNode; }
```

```
/* function to print the adjacency
 list representation of graph */
 void print (struct Graph* graph){
   int v;
 for (v= 0; v<graph->v; ++v) {
  struct AdjNode * pCrawl = graph
   ->array [v].head;
  printf (" \n Adjacency list of
   vertex %d is: \n head", v);
  while (pCrawl) {
  printf ("->%d", pCrawl->dest);
     pCrawl = pcrawl->next; }
     printf ("\n"); } }
 int main() {
  int V= 4;
 struct Graph* g = createGraph(V);
   addEdge (g, 0, 1 );
   addEdge (g, 0, 3);
   addEdge (g, 1, 2);
   addEdge (g, 1, 3);
   addEdge (g, 2, 4);
   addEdge (g, 2, 3);
   addEdge (g, 3, 4);
   print (g);
   return 0; }
```

```
O/P: Adjacency list of vertex 0 is:
head -> 1 -> 3
Adjacency list of vertex 1 is:
head -> 0 -> 2 -> 3
Adjacency list of vertex 2 is:
head -> 1 -> 4 -> 3
Adjacency list of vertex 3 is:
head -> 0 -> 1 -> 2 -> 4
```

BFS Algorithm

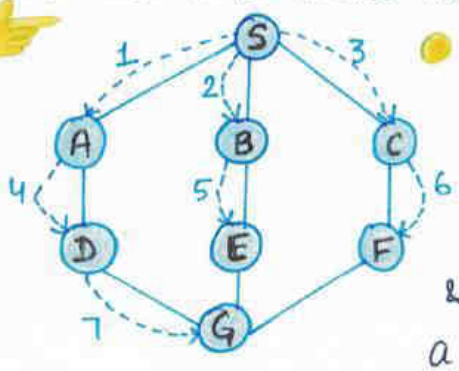
Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores the neighboring nodes.

After that, it selects the nearest node and explores all the unexplored nodes.

While using BFS for traversal, any node in the graph can be considered as the root node.

It is a recursive algorithm to search all the vertices.

BFS puts every vertex of the graph into two categories - visited and non-visited, it selects a single node after that visits all the nodes adjacent to the selected node.



As in this example, BFS algorithm traverses from A to B to E to F first then to C & G lastly to D. It employs the following rules-

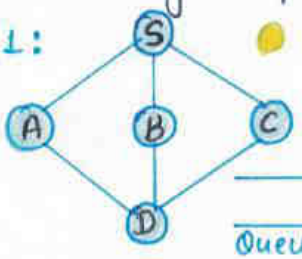
Rule 1: Visit the adjacent unvisited vertex & mark it as visited. Display it. Insert it in a queue.

Rule 2: If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3: Repeat Rule 1 & Rule 2 until the queue is empty.

Following steps are involved in BFS algorithm.

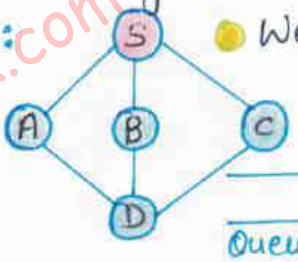
step 1:



Queue

Initialize the queue.

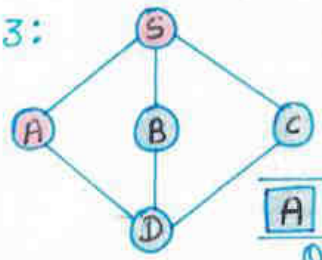
step 2:



Queue

We start from S and mark it as visited.

step 3:



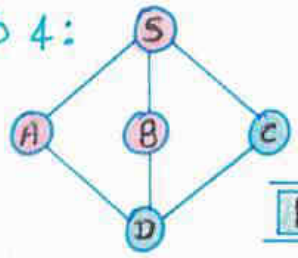
A

Queue

We then see an unvisited adjacent node from S.

In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.

step 4:

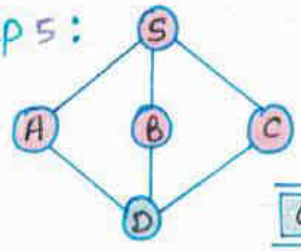


- Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.

B	A

Queue

step 5:

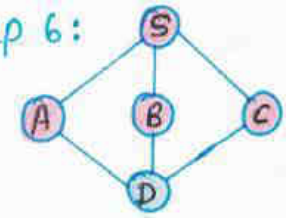


- Next, the unvisited node node from S is C, We mark it as visited and enqueue it.

C	B	A

Queue

step 6:

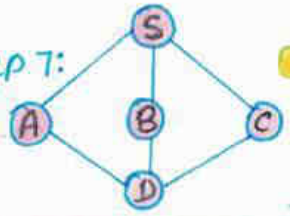


- Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.

C	B

Queue

step 7:



- From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

D	C	B

Queue

Applications

The applications of BFS are given as follows-

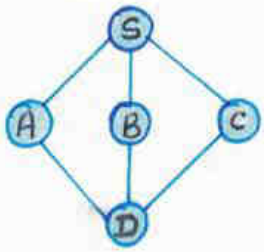
1. BFS can be used to find the neighboring locations from a given source location.
2. In a peer-to-peer Network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
3. BFS can be used in web crawlers to create web page index. It starts traversing from the source page and follows the links associated with the page, every web page is considered as node.
4. BFS is used to determine the shortest path and minimum spanning tree.
5. BFS is also used in Cheney's technique to duplicate the garbage collection.
6. It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Time complexity of BFS depends on the data structure used to represent the graph.

The time complexity of BFS algorithm is O(V+E), since in the worst case, it explores every node and edge.

The space complexity of BFS is O(V), V is no. of vertices.

Eg: Implementation of BFS Algorithm.




```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
struct Vertex { char label;
   bool visited; };
/* queue variables */
int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;
/* Graph variables */
/* array of vertices */
struct Vertex* lstVertices[MAX];
/* adjacency Matrix */
int adjMatrix[MAX][MAX];
/* vertex count */
int vertexCount = 0;
/* queue functions */
void insert ( int data) {
    queue[++rear] = data;
    queueItemCount++; }

int removeData( ) {
    queueItemCount--;
    return queue[front++]; }
bool isQueueEmpty ( ) {
  return queueItemCount == 0; }
```

```
/* Graph functions */
/*add vertex to the vertex list */
void addVertex ( char label) {
   struct Vertex * vertex = (struct
   vertex*) malloc ( sizeof(struct vertex));
   vertex->label = label;
   vertex->visited = false;
   lstVertices[vertexCount++] =
     vertex ; }
/* add edge to edge array */
void addEdge ( int start, int end)
{ adjMatrix[start][end] = 1;
  adjMatrix[end][start] = 1; }
/* display the vertex */
void displayVertex ( int vertexIndex){
  printf ("%c ", lstVertices[vertex
    Index]->label); }
/* get the adjacent unvisited vertex */
int getAdjUnvisitedVertex ( int
     vertexIndex) {
   int  i;
 for(i=0; i<vertexCount; i++){
     if ( adjMatrix[vertexIndex]
     [i]==1 && lstVertices[i]->
   visited == false)
     return i; }
   return -1; }
```

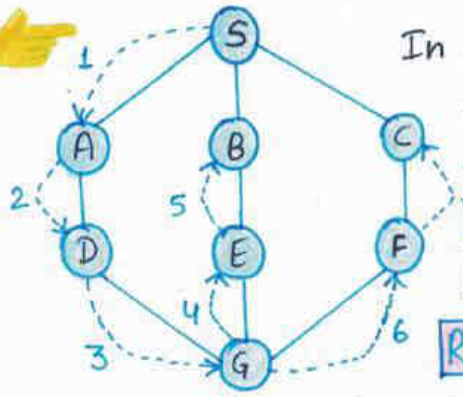
```
void breadthFirstSearch ( ) {
  int i;
/* mark first node as visited*/
 lstVertices[0] →visited = true;
/* display the vertex */
  displayVertex (0);
/* insert vertex index in queue */
  insert (0);
   int unvisitedVertex;
 while ( ! isQueueEmpty ( ) ) {
/* get the unvisited vertex which is
 at front of the queue*/
  int tempVertex = removeData ( );
/*no adjacent vertex found*/
  while (( unvisitedVertex = getAdj
 UnvisitedVertex (tempVertex)) ! = -1){
    lstVertices[unvisitedVertex]
   →visited = true;
  displayVertex (unvisitedVertex);
  insert (unvisitedVertex); }}
```

```
/*queue is empty, search is comple-
 -te, reset the visited flag */
 for (i=0; i<vertexCount; i++){
   lstVertices[i] →visited = false;
  }}
int main ( ) {
  int i, j;
 for(i = 0; i<MAX; i++)
    for ( j=0; j<MAX; j++)
      adjMatrix[i][j] = 0; }
 addVertex ('S') ; /* 0 */
 addVertex ( A ); /* 1 */
 addVertex ( B ); /* 2 */
 addVertex ( C ); /* 3 */
 addVertex ( D ); /*4 */
 addEdge ( 0, 1 ); /* S-A */
 addEdge ( 0, 2 ); /* S-B */
 addEdge ( 0, 3 ); /*S-C */
 addEdge ( 1, 4 ); /*A-D */
 addEdge ( 2, 4 ); /* B-D */
 addEdge ( 3, 4 ); /*C-D */
  printf (" Breadth first Search:");
   breadthFirstSearch ( );
    return 0; }
```

```
O/P: Breadth first Search:
     S A B C D
```

DFS Algorithm

- The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.
- It is a recursive algorithm to search all the vertices of a tree data structure or a graph.
- Because of the recursive nature, stack data structure can be used to implement the DFS algorithm.
- The process of implementing the DFS is similar to BFS.



In this example, DFS algorithm traverses from S to A to D to G to E to B first, then to F & lastly to C. It employs following rules.

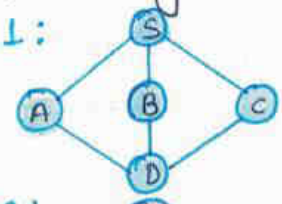
Rule 1: Visit the adjacent unvisited vertex. Mark it as visited. Display & push it in stack.

Rule 2: If no adjacent vertex is found, pop up a vertex from stack.

Rule 3: Repeat Rule 1 & 2 until the stack is empty.

Following steps are involved in DFS algorithm –

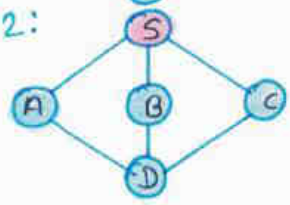
step 1:



stack

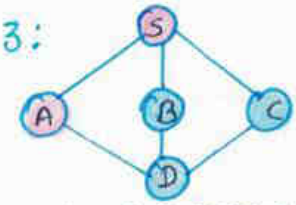
- Initialize the stack.

step 2:



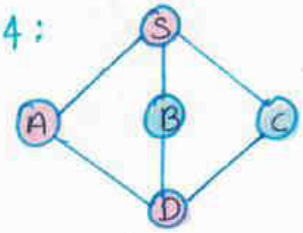
- Mark S as visited & put it onto the stack & explore any unvisited adjacent node from S.

step 3:



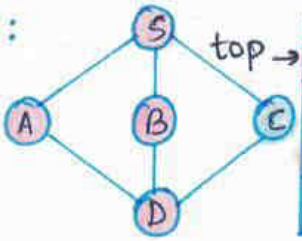
- Mark A as visited & put it onto stack. Explore any unvisited adjacent node from A & both S & D are adjacent to A but we are concerned for unvisited nodes only.

step 4:



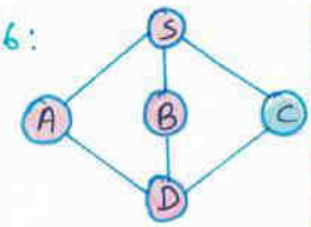
- Visit D and mark it as visited and put onto the stack.
- Here, B & C nodes which are adjacent to D and both are unvisited.

step 5:



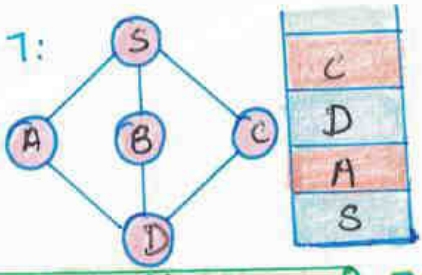
- We choose B, mark it as visited & put onto the stack.
- Here, B does not have any unvisited adjacent node, so we pop B from stack

step 6:



- We check the stack top for return to the previous node and check if it has any unvisited nodes.
- Here, we find D to be on the top of the stack.

Step 7:



stack

- only unvisited adjacent node is from D is C now, so we visit C, mark it as visited and put it onto the stack.

Applications

The applications of using DFS algorithm are given as follows-

1. DFS algorithm can be used to implement topological sorting.
2. It can be used to find paths between two vertices.
3. It can also be used to detect cycles in the graph.
4. DFS algorithm is also used for one solution puzzles.
5. DFS is used to determine if a graph is bipartite or not.

The time complexity of DFS algorithm is $O(V+E)$, where V is the number of vertices and E is the no. of edges.

The space complexity of DFS algorithm is $O(V)$.

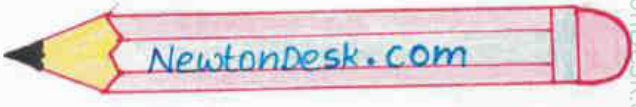
Eg: Implementation of DFS Algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 5
struct Vertex { char label;
      bool visited; };
/* stack variables */
int stack[MAX];
int top = -1;
/* graph variables */
/* array of vertices */
struct Vertex* lstVertices[MAX];
/* adjacency matrix */
 int adjMatrix[MAX][MAX];
/* vertex count */
 int vertexCount = 0;
/* stack functions*/
  void push ( int item) {
    stack[++top] = item; }
int pop( ) {
  return stack[top--]; }
int peek( ) {
  return stack[top ]; }
bool isStackEmpty ( ) {
  return top == -1; }
/* graph functions */
/*add vertex to the vertex list*/
 void addVertex ( char label) {
    struct Vertex* vertex = (struct
     Vertex*) malloc (sizeof( struct
     Vertex));
    vertex->label = label;
    vertex->visited = false;
    lstVertices[vertexCount++]
     = vertex; }
```

```
/* add edge to edge array */
void addEdge ( int start , int end)
{   adjMatrix[ start ][ end ] = 1;
    adjMatrix[end][start ] = 1; }
/* display the vertex */
void displayVertex ( int vertexIndex)
{  printf (" %c ", lstVertices [
   vertexIndex] -> label); }
/* get the adjacent unvisited vertex */
int getAdjUnvisitedVertex ( int
     vertexIndex) {
    int  i;
 for (i=0; i<vertexCount; i++) {
     if (adjMatrix[vertexIndex]
[i] == 1 && lstVertices[i] -> visited
    == false  ) {
      return  i; }}
      return -1; }
void depthFirstSearch ( ) {
    int i;
/* mark first node as visited */
  lstVertices [0] -> visited = true ;
/* display the vertex */
  displayVertex (0);
/* push vertex index in stack */
   push ( 0);
   while (!isStackEmpty ( )) {
/* get the unvisited vertex of vertex
which is at top of the stack */
   int unvisitedVertex = getAdj-
       UnvisitedVertex (peek ( ));
/* no adjacent vertex found */
    if (unvisitedVertex == -1) {
      pop ( ); }
else  {
   lstVertices [unvisitedVertex] ->
visited = true   ;
 displayVertex (unvisitedVertex);
 push (unvisitedVertex ); }
/* stack is empty, search is com-
-plete, reset the visited flag */
 for (i=0; i<vertexCount; i++){
   lstVertices[i ] -> visited = false;
 } }
 int main ( ) {
   int i,  j ;
   for(i= 0; i<MAX; i++)
     for ( j=0; j<MAX; j++)
      adjMatrix[i][j ] = 0; }
  addVertex ('S'); /* 0 */
  addVertex ('A'); /* 1 */
  addVertex ('B'); /* 2 */
  addVertex ('C'); /* 3 */
  addVertex ('D'); /* 4 */
  addEdge (0, 1 ); /* S-A */
  addEdge (0, 2 ); /* S- B */
  addEdge (0, 3 ); /* S- C */
  addEdge (1, 4 ); /* A- D */
  addEdge ( 2, 4 ); /* B- D */
  addEdge ( 3, 4 ); /* C- D */
   printf ("Depth First Search: ");
   depthFirstSearch ( );
    return 0; }
```

O/P: Depth First Search

S A D B C



Spanning Tree

A spanning tree can be defined as subgraph of undirected connected graph.

It includes all the vertices along with the least possible number of edges, if any vertex missed, then it is not spanning tree.

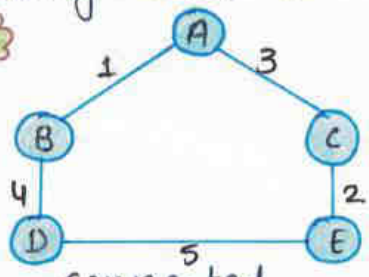
A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of 'n-1' edges, where 'n' is the number of vertices (or nodes).

All the possible spanning trees created from the given Gr-aph 'G' would have the same number of vertices, but the no. of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

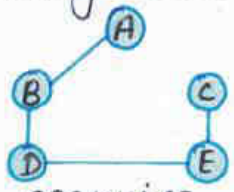
A complete undirected graph can have (n^{n-2}) number of spanning trees where 'n' is the number of vertices.

Eg:

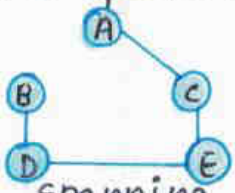


Connected Graph

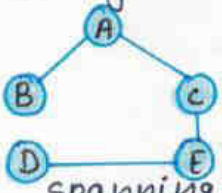
There will be 5 vertices & 4 edges in spa-nning tree. Some spanning trees given below-



Spanning tree 1



Spanning tree 2



Spanning tree 3

Properties

Some of the properties of spanning tree are given as follows-

1. There can be more than one spanning trees of a graph.
2. A spanning tree does not have any cycles or loop.
3. A spanning tree is minimally connected, so removing one edge from the tree will make the graph disconnected.
4. A spanning tree is maximally acyclic, so adding one edge to the tree will create a loop.
5. If the graph is a complete graph, then the spanning tree can be constructed by removing maximum $(e-n+1)$ edges, where 'e' is the number of edges and 'n' is the number of vertices.

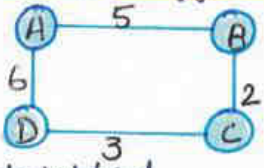
A spanning tree is a subset of connected graph & there is no spanning tree of a disconnected graph.

Minimum Spanning Tree

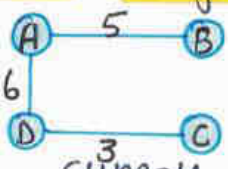
A minimum spanning tree can be defined as the spanning tree in which sum of weights of edge is minimum.

In the real world, this weight can be considered as the distance, traffic load, congestion or any random value.

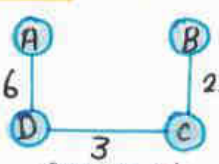
Eg:



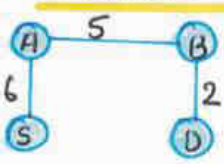
Weighted Graph



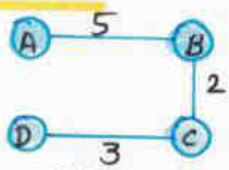
sum=14



sum=11

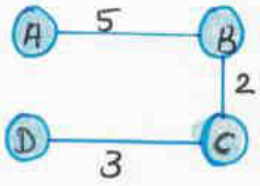


sum=13



sum=10

So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Sum = 10

Applications

The applications of the minimum spanning tree are given as follows-

1. Spanning trees are used in Cluster analysis, Civil network planning, Computer network routing protocol.
2. Minimum spanning tree can be used to design water supply networks, telecommunication networks and electrical grids.
3. It can be used to find paths in the map.

Algorithms For Minimum Sp. Tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below-

- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm is a greedy algorithm that starts with an empty spanning tree, this algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Kruskal's Algorithm also follows greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.

SPARSE MATRIX

- A matrix is a two-dimensional data object made of 'm' rows and 'n' columns, therefore having total mxn values.
- If most of the elements of the matrix have 0 value, then it is called a sparse matrix.
- We use sparse matrix instead of simple matrix because of storage and computing time.
- There are lesser non-zero elements than zeros, thus lesser memory can be used to store only those elements.
- Computing time can be saved by logically designing a data structure traversing only non-zero elements.
- Representing a sparse matrix by a 2D array leads to wastage of memory as zeros in the matrix are of no use.
- Instead of storing zeros with non-zero elements, we only store non zero elements with triples - (Row, Column, Value).
- Sparse matrix representations can be done in following ways-
  - Array representation
  - linked list representation

Array Representation

- 2D array is used to represent a sparse matrix in which there are three rows named as -

Row - Index of row, where non-zero element is located.

Column - Index of column, where non zero element is located.

Value - value of the non zero element located at index - (row, column)

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Eg: Implementation of sparse matrix using array.

```
#include<stdio.h>
int main( ){
  int sparseMatrix[4][5]=
  { {0, 0, 3, 0, 4},
    {0, 0, 5, 7, 0},
    {0, 0, 0, 0, 0},
    {0, 2, 6, 0, 0} };
  size=0;
  for(int i=0; i<4; i++)
    for(int j=0; j<5; j++)
      if(sparseMatrix[i][j]!=0)
        size++;
  number of columns equal to no. of non-zero elements
  int compactMatrix[3][size];
  /* making of new matrix */
```

```
int k = 0;
for (int i=0; i< 4; i++)
    for (int j= 0; j<5; j++)
      if (sparseMatrix[i][j]!=0)
{     compactMatrix[0][k]=i;
      compactMatrix[1][k]=j;
      compactMatrix[2][k]=
        sparseMatrix[i][j];
      k++; }
```

```
for (int i=0; i<3; i++)
    for (int j=0; j<size; j++)
      printf ("%d", compactMatrix
              [i][j]);
printf ("\n"); }
  return 0; }
```

OIP:

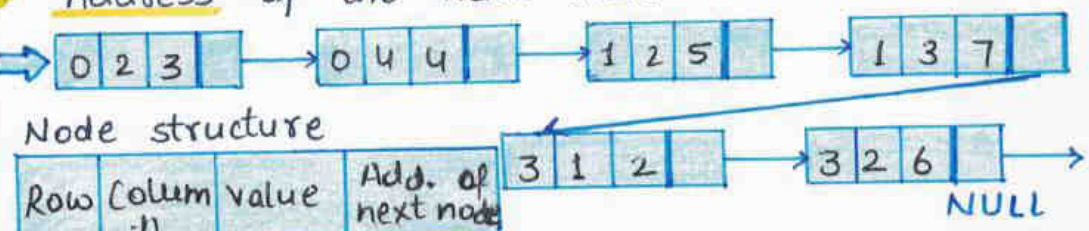
0	0	1	1	3	3
2	4	2	3	1	2
3	4	5	7	2	6

Linked List Representation

In linked list, each node has four fields, these are defined as –

- **Row** – Index of row, where non-zero element is located.
- **Column** – Index of column, where non-zero element is located.
- **Value** – value of the non zero element located at index.
- **Next-Node** – Address of the next node.

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0



Eg: Implementation of sparse Matrix using LL.

```
#include <stdio.h>
#include <stdlib.h>
  struct Node { int value; int
      row_position; int column_
    position; struct Node *next;};
/*function to create new Node*/
 void create_new_node (struct
   Node** start, intnon_zero_
 element, int row_index,
    int column_index)
   {
     struct Node *temp, *r;
       temp = *start;
       if (temp == NULL) {
/*create new node dynamically*/
  temp = (struct Node*) malloc
    (sizeof (struct Node));
 temp->value = non_zero_element;
 temp->row_position = row_index;
 temp->column_position = column_index;
 temp->next = NULL;
  *start = temp; }
  else {
  while (temp->next != NULL)
      temp = temp->next;
/*create new node dynamically*/
```

```
    r = (struct Node*) malloc
    (sizeof(struct Node));
    r->value = non_zero_element;
    r->row_position = row_index;
    r->column_position= column_index;
    r->next = NULL;
    temp->next = r; }
/* this function prints contents of
  linked list */
void PrintList (struct Node* start)
  {  struct Node *temp ,*r, *s;
     temp = r = s = start;
   printf (" row-position: ");
   while (temp != NULL ) {
    printf ("%d ", temp->row_
       position );
     temp = temp->next;}
    printf  ("\n ");
     printf ("column_position:");
   while  (r != NULL){
     printf ("%d ", r->column_
        position );
      r = r->next; }
    printf ("\n ");
    printf ("value:");
   while (s != NULL) {
     printf ("%d ", s->value);
       s = s->next; }
    printf ("\n ");
/* Driver of the program */
 int main ( ) {
/* assume 4x5 sparse matrix */
   int sparseMatrix[4][5] =
 { {0, 0, 3, 0, 4},
    {0, 0, 5, 7, 0},
    {0, 0, 0, 0, 0},
    {0, 2, 6, 0, 0} };
/* start with the empty list */
 struct Node * start = NULL;
  for (int i=0; i<4; i++)
    for (int j=0; j<5; j++)
/* pass only those values which
  are non zero */
     if (sparseMatrix[i][j] != 0)
      create_new_node (& start ,
  sparseMatrix[i][j], i, j );
  PrintList (start );
   return 0; }
```

```
O/P: row-position: 0 0 1 1 3 3
Column_position: 2 4 2 3 1 2
value: 3 4 5 7 2 6
```

TOWER OF HANOI

The tower of Hanoi is a mathematical puzzle containing 3 pillars / towers with n disks each of different size.

These disks can slide onto any pillar.

The following diagram shows the initial state of the problem.

Eg:

- Here, we have stacked the disks over each other in the ascending order of their sizes.
- We can have only 3 towers and any number of disks.

Our objective in this puzzle is to move all these disks to another pillar without changing the order in which the disks are placed in initial state.

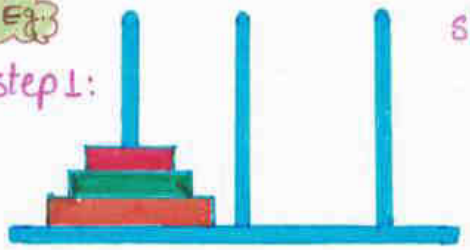
A few steps to be followed for tower of hanoi are-

1. Only one disk can be moved among the towers at any given time.
2. Only the 'top' disk can be moved.
3. No large disk can sit over a small disk.

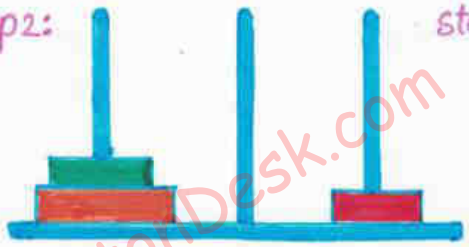
Tower of Hanoi puzzle with n disks can be solved in minimum $2^n - 1$ steps.

Eg:

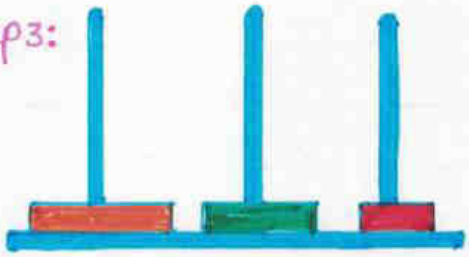
step1:



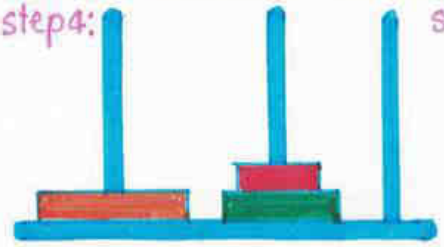
step2:



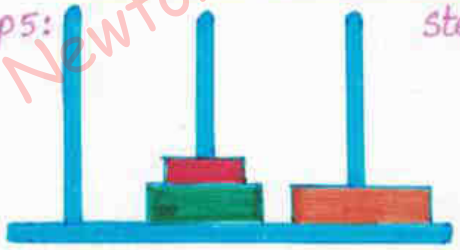
step3:



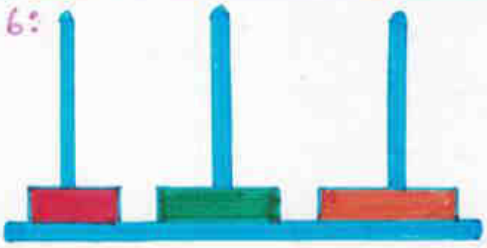
step4:



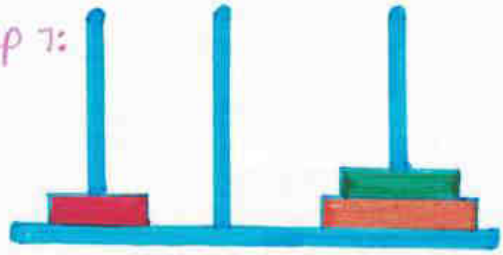
step5:



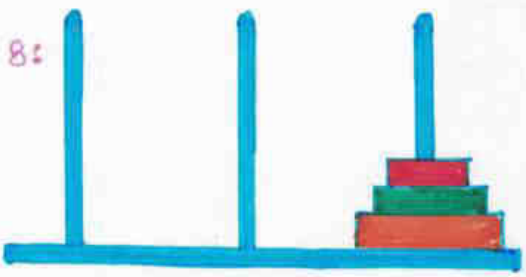
step 6:



step 7:



step 8:



Eg: Implementation of Towers of Hanoi.

```
#include<stdio.h>
void toh(int, char, char, char);
  int main( ){
   charsource = 'A', destination
   = 'B', Auxiliary = 'C';
    int n;
   printf("Enter value of n:");
   scanf("%d", &n);
     toh(n, source, destination,
     Auxiliary);
       return 0; }
void toh( int n, char source,
   char dest, char aux) {
  if (n==1) {
   printf("%c→%c\n", source, dest);
     return ; }
   toh (n-1, source, aux; dest);
   printf ("%c→%c\n", source,
     dest );
   toh (n-1, aux, dest , source); }
```

O/P: Enter value of n: 3

A → B
A → C C → A
B → C C → B
A → B A → B

Que Given that input (4322, 1334, 1471, 9674, 1989, 6171, 6173, 4199) & the hash function x mod 10 which of the following are true -

(A) 9679, 1989, 4199 has to same value.
(B) 1471, 6171 hash to same value.
(C) All elements hash to same value.
(D) Each element has to a differentvalue.

(i) (A) only
(ii) (B) only
(iii) (A) & (B) only
(iv) (C) & (D)

Solution $h(K) = K \bmod 10$

9679, 1989 & 4199 all these give same hash value i.e. 9.
& 1471 & 6171 give hash value 1.

(iii) A & B only

Que Consider the hash table with 100 slots. Collisions are resolved using chaining assuming simple uniform hashing. What is the probability that the first three slots are unfilled after the first 3 insertion.

(A) $(97 \times 97 \times 97)/100^3$
(B) $(99 \times 98 \times 97)/100^3$
(C) $(97 \times 96 \times 95)/100^3$
(D) $(97 \times 96 \times 95)/(3! \times 100^3)$

Solution: Probability that the first 3 slots are unfilled after the first 3 insertions = $(97/100) * (97/100) * (97/100)$

(A) option

Que Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for ranging from 0 to 2020.

(A) $h(i) = i^2 \bmod 10$ (B) $h(i) = i^3 \bmod 10$

(C) $h(i) = (11 * i^2) \bmod 10$ (D) $h(i) = (12 * i^2) \bmod 10$

Solution: $\{0, 1, 4, 5, 6, 9\} \rightarrow$ 6 slots

(B) $h(i) = i^3 \bmod 10$

$i^2 \bmod 10$	$i^3 \bmod 10$
$1 \rightarrow 1 \rightarrow 1$	$1 \rightarrow 1 \rightarrow 1$
$2 \rightarrow 4 \rightarrow 4$	$2 \rightarrow 8 \rightarrow 8$
$3 \rightarrow 9 \rightarrow 9$	$3 \rightarrow 27 \rightarrow 7$
$4 \rightarrow 16 \rightarrow 6$	$4 \rightarrow 64 \rightarrow 4$
$5 \rightarrow 25 \rightarrow 5$	$5 \rightarrow 125 \rightarrow 5$
$6 \rightarrow 36 \rightarrow 6$	$6 \rightarrow 216 \rightarrow 6$
$7 \rightarrow 49 \rightarrow 9$	$7 \rightarrow 343 \rightarrow 3$
$8 \rightarrow 64 \rightarrow 4$	$8 \rightarrow 512 \rightarrow 2$
$9 \rightarrow 81 \rightarrow 1$	$9 \rightarrow 729 \rightarrow 9$
$10 \rightarrow 100 \rightarrow 0$	$10 \rightarrow 1000 \rightarrow 0$

no collision

0	10
1	1
2	8
3	7
4	4
5	5
6	6
7	3
8	2
9	9

Que Which of the following is true -

(1) A hash function takes a message of arbitrary length and generate a fixed length code.

(2) A hash function takes a message of fixed length and generate a code of variable length.

(3) A hash function may give the same hashvalue for distinct messages.

(A) I only (B) II & III (C) 1 & 3 (D) 2 only

Solution: $h(k) \rightarrow h: \{u \rightarrow \{0, 1, 2, \ldots m-1\}$

any variable length I/p & generate fixed length output.

(C) 1 & 3

Que Given a hash table T with 25 slots that stored 2000 elements, the load factor α for T is -

Solution: load factor $(\alpha) = \frac{n}{m}$ = avg. no. of items that are hashed to same slot assuming simple uniform hashing.

$$\alpha = \frac{2000}{25} = 80$$

80

Que

Consider a hash function that distributes key uniformly. The hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5.

(A) 5 (B) 6 (C) 7 (D) 10

Solution: For each entry probability of collision is 1/20 { as possible total spaces = 20, and an entry will go into only 1 place }.

⇒ After inserting x values probability becomes 1/2.

⇒ $(1/20) \cdot x = 1/2 \Rightarrow x = 10$

(D) 10

Que

The characters of the string K, R, P, C, S, N, Y, T, J, M are inserted into a hash table of size 10. Using hash function

$h(x) = (\text{Ord}(x) - \text{Ord}("a") + 1) \bmod 10$

If linear probing is used to resolve collisions, then the following insertion causes collision.

(A) Y (B) C (C) M (D) P

Solution:

A^1	B^2	C^3	D^4	E^5	F^6	G^7	H^8	I^9	J^{10}
K^{11}	L^{12}	M^{13}	N^{14}	O^{15}	P^{16}	Q^{17}	R^{18}	S^{19}	T^{20}
U^{21}	V^{22}	W^{23}	X^{24}	Y^{25}	Z^{26}				

$h(x) = (\text{ord}(x) - \text{ord}(A) + 1) \bmod 10$

$= K - A + 1 \Rightarrow 11 - 1 + 1 \bmod 10 = 1$

(C) M

R mod 10 = 8

P mod 10 = 6

C mod 10 = 3

S mod 10 = 9

N mod 10 = 4

Y mod 10 = 5

T mod 10 = 0

J mod 10 = 0 // first collision

M mod 10 = 3 // second collision

Only J & M are causing the collision.

Final Hash table →

0	T
1	K
2	J
3	C
4	N
5	Y
6	P
7	M
8	R
9	S

Formula Sheet

•NewtonDesk.com•

PHYSICS - GRADE 11

Physical Constants: Name + representation + values

- Speed of light: $c = 3 \times 10^8$ m/s [constant in all inertial frames]
- Gravitation constant: $G = 6.67 \times 10^{-11}$ $m^3 kg^{-1} s^{-2}$ [always constant]
- Boltzmann constant: $k = k_B = 1.38 \times 10^{-23}$ J/k
- Molar gas constant: $R = 8.314$ J $mol^{-1} K^{-1}$
- Avogadro's number: $N_A = 6.023 \times 10^{23}$ mol^{-1} [const. for all things]
- Acceleration due to gravity: $g = 9.8$ m/s² [changes with height]
- Radius of Earth: $R_E = 6400$ km $= 6.4 \times 10^6$ m [nearly constant]
- Eccentricity of Earth's orbit: $e_E = 0.017$ [changes with time]
- Standard atmospheric pressure: atm $= 1.013 \times 10^5$ Pa $= 76$ cm-Hg
- Density of water (STP): $\rho_w = 1 \times 10^3$ kg/m³ [changes with T]
- Reynold number: $N_R < 2000$; $2000 < N_R < 3000$; $N_R > 3000$ [changes with the type of flow; velocity]
- Adiabatic Index or ratio of heat capacity: $\gamma > 1$ [$C_P > C_V$]
- Absolute zero temperature: $T_0 = 0$ K $= -273.15°C$ [absolute]
- Limiting value of specific heat: $C_v = 3R = 24.94$ J/mol/k.
- k_BT value at room temperature: $k_BT = 0.026$ eV
- Adiabatic bulk modulus of elasticity of gas: $K = \gamma P = 2 \times 10^5$ N/m²
- Speed of sound in air: $V_s = 330$ m/s

Unit I: Measurement and vectors:

NewtonDesk.com

- Base quantities and SI units:

S. No.	Base Quantity	Symbols	other/alternate units	dimensional Representation
1.	Length	metre; m	centimetre; cm; kilometre;	$[M^0 L^1 T^0]$
2.	Mass	kilogrm; kg	Grams; g / Tonn etc.	$[M^1 L^0 T^0]$
3.	Time	second; s	Minutes; min / hours; hr	$[M^0 L^0 T^1]$
4.	Electric current	Ampere; A	Coulomb per sec; C/s	$[M^0 L^0 T^0 A^1]$
5.	Thermodynam Temperature	kelvin; K	degree celcius/fahrenhite; °C	$[M^0 L^0 T^0 K^1]$