SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution & Accredited by NAAC with "A" Grade)
(An Autonomous Institution)
Madagadipet, Puducherry - 605 107

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject Name: **Data Structures**
Subject Code:

Prepared by:

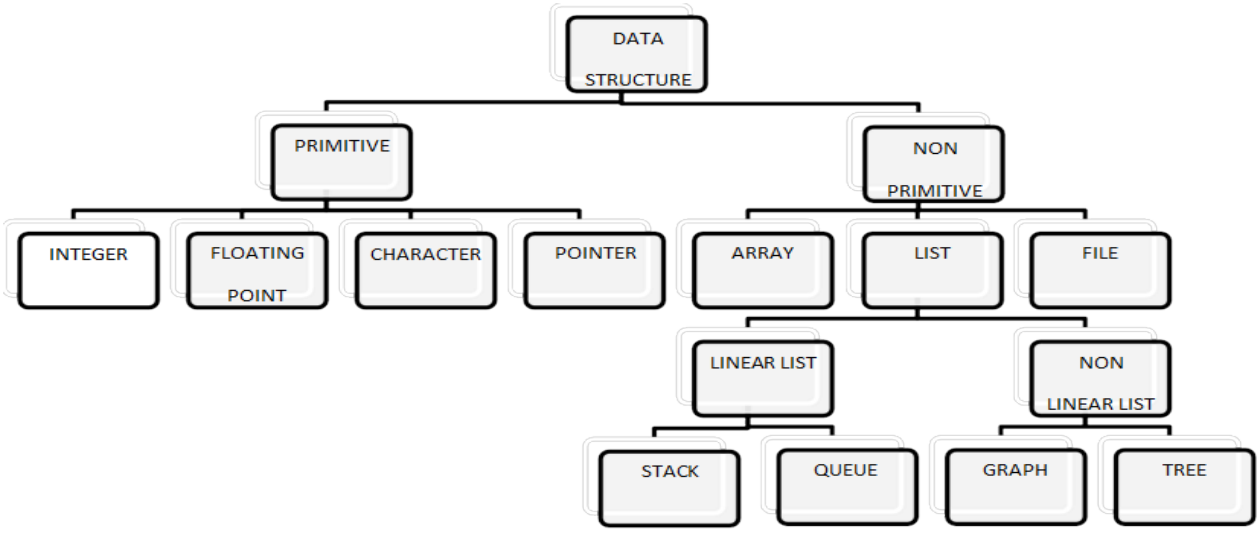
Verified by: **Approved by:**

UNIT – I

Introduction: Basic Terminologies: Elementary Data Organizations. Data Structure Operations: insertion, deletion, traversal. Analysis of an Algorithm, Asymptotic Notations, Time-Space trade off. Array and its operations. Searching: Linear Search and Binary Search Techniques and their complexity analysis.

2 Marks

1. **What is Data Structure?**
   - Data structure is a representation of the logical relationship existing between individual elements of data.
   - Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
   - We can also define data structure as a mathematical or logical model of a particular organization of data items.
   - The representation of particular data structure in the main memory of a computer is called as storage structure.
2. **Draw the classification of data structure?**



3. **What are the two main categories of data structure?**
   1. Primitive Data Structure
   2. Non-primitive data Structure
4. **Define datatype?**

A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

5. Define primitive data structure?

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.

6. Define non-primitive data structure?

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.

7. List the types of non-primitive data structure?

A Non-primitive data type is further divided into Linear and Non-Linear data structure

8. Define linear data structure?

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- Examples of Linear Data Structure are Stack and Queue.

9. Define non-linear data structure?

- Nonlinear data structures are those data structure in which data items are not arranged in a sequence.
- Examples of Non-linear Data Structure are Tree and Graph.

10. What are the ways to represent linear data structure in memory?

- There are two ways to represent a linear data structure in memory,
  - Static memory allocation
  - Dynamic memory allocation

11. Define stack?

Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only. The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation. Stack is also called as Last in First out (LIFO) data structure.

12. Define queue?

Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue. End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end. Queue is also called as First in First out (FIFO) data structure.

13. Define tree?

Tree: A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.

- Trees represent the hierarchical relationship between various elements.
- Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

14. Define graph?

Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

- A tree can be viewed as restricted graph.

15. List various types of graph?

Graphs have many types:

- Un-directed Graph
- Directed Graph
- Mixed Graph
- Multi Graph

- Simple Graph
- Null Graph
- Weighted Graph

16. Difference between linear and non-linear data structure?

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

17. What are the various operations carried out on data structures?

Create, destroy, selection, updation, searching, sorting, merging, splitting, traversal

18. Define algorithm?

- An essential aspect to data structures is algorithms.
- Data structures are implemented using algorithms.
- An algorithm is a procedure that you can write as a C function or program, or any other language.
- An algorithm states explicitly how the data will be manipulated.

19. What is complexity of an algorithm?

- Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.
- The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.

20. Define time complexity?

- Time Complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

21. Define space complexity?

- Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
- Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

22. Define worst case analysis?

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be

23. Define best case analysis?

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.

24. Define average case analysis?

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by (n+1).

25. Define array with example?

- An array is a collection of elements of the same type that are referenced by a common name.
- Compared to the basic data type (int, float & char) it is an aggregate or derived data type.
- All the elements of an array occupy a set of contiguous memory locations.
  Example: int studMark[1000];

26. What is two dimensional array with example?

A two dimensional array has two subscripts/indexes. The first subscript refers to the row, and the second, to the column. Its declaration has the following form,

data_type array_name[1st dimension size][2nd dimension size];
For examples,
int xInteger[3][4];
float matrixNum[20][25];

27. What are the basic operations supported by array?

Traverse, insertion, deletion, search and update.

28. What is searching and its types?

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories: sequential and interval.

29. Define linear search?

Linear search in C is to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.

30. Define binary search?

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

<u>5 marks</u>:

1. Explain various operations of data structure briefly?

Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly categorized into following types

- **Create**

  The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.

- **Destroy**

Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.

- **Selection**
  Selection operation deals with accessing a particular data within a data structure.

- **Updation**
  It updates or modifies the data in the data structure.
- **Searching**
  It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.
- **Sorting**
  Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.
- **Merging**
  Merging is a process of combining the data items of two different sorted list into a single sorted list.
- **Splitting**
  Splitting is a process of partitioning single list to multiple list.
- **Traversal**
  Traversal is a process of visiting each and every node of a list in systematic manner.

2. Explain the three types of algorithm analysis?

Worst Case Analysis:

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be.

Average Case Analysis:

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by (n+1).

Best Case Analysis:

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.

3. Explain two dimensional array?

A two dimensional array has two subscripts/indexes. The first subscript refers to the row, and the second, to the column. Its declaration has the following form,

data_type array_name[1st dimension size][2nd dimension size];

For examples,

```
int     xInteger[3][4];
float   matrixNum[20][25];
```

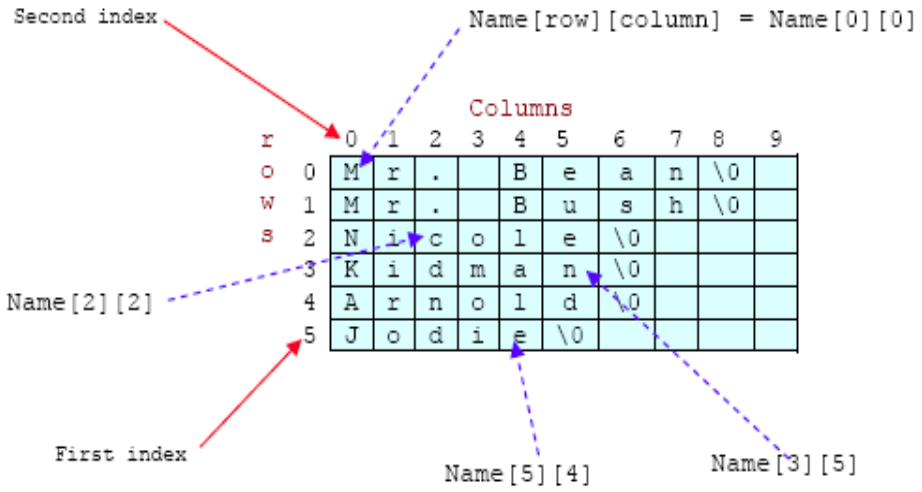
The first line declares xInteger as an integer array with 3 rows and 4 columns.

Second line declares a matrixNum as a floating-point array with 20 rows and 25 columns.

If we assign initial string values for the 2D array it will look something like the following,

```
char Name[6][10] = {"Mr. Bean", "Mr. Bush", "Nicole", "Kidman", "Arnold", "Jodie"};
```

Here, we can initialize the array with 6 strings, each with maximum 9 characters long. If depicted in rows and columns it will look something like the following and can be considered as contiguous arrangement in the memory.



- Take note that for strings the null character (\0) still needed.
- From the shaded square area of the figure we can determine the size of the array.
- For an array Name[6][10], the array size is 6 x 10 = 60 and equal to the number of the colored square. In general, for

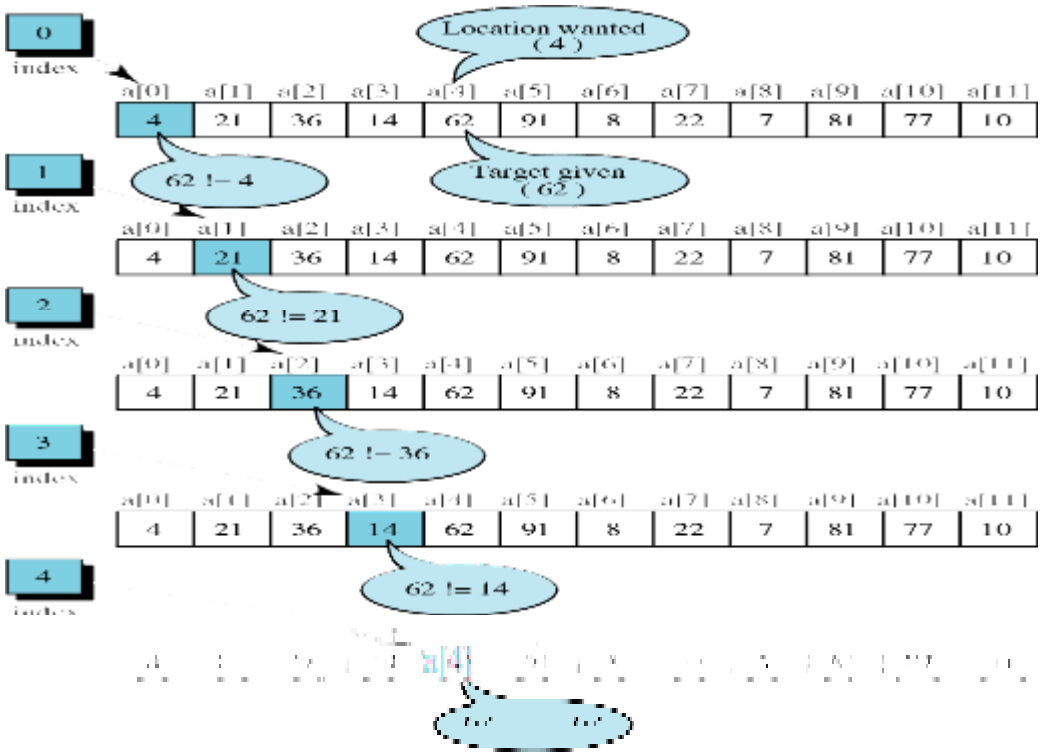
  array_name[x][y];
- The array size is = First index x second index = xy.
- This also true for other array dimension, for example three dimensional array,
- array_name[x][y][z]; => First index x second index x third index = xyz
- For example,

  ThreeDimArray[2][4][7] = 2 x 4 x 7 = 56.

  And if you want to illustrate the 3D array, it could be a cube with wide, long and height dimensions.

4. Explain linear search with sample program?

Linear search in C is to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.



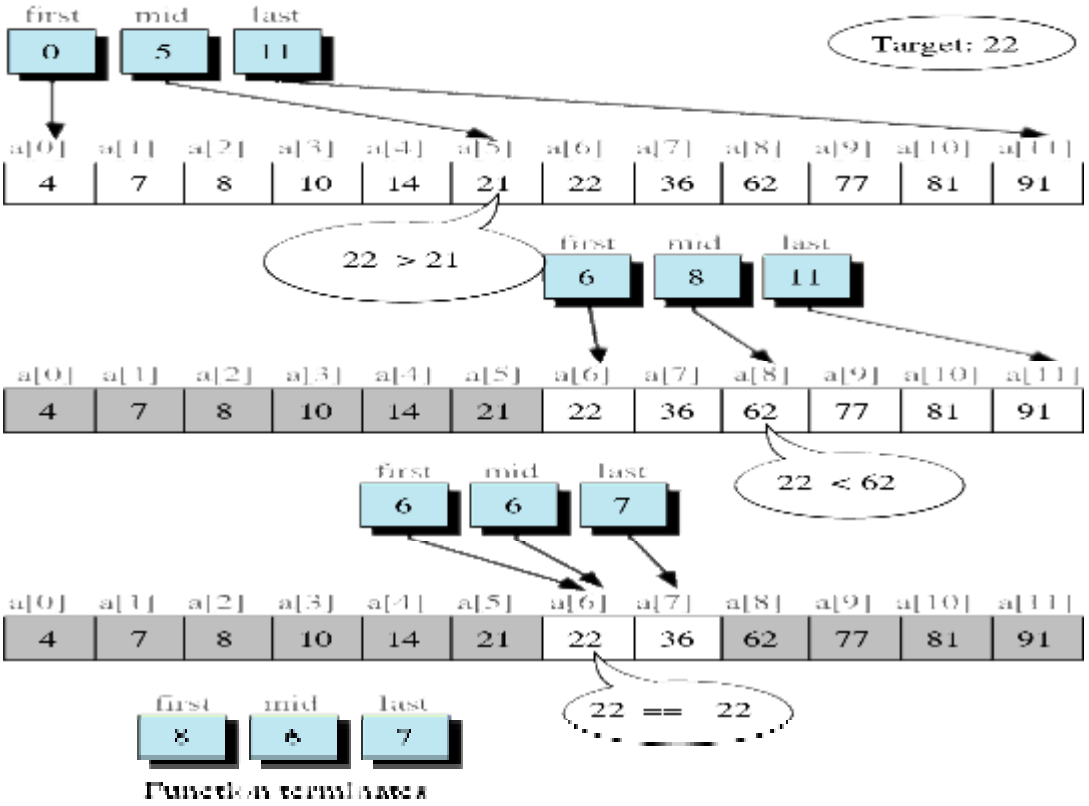
Above diagram shows the simple example to find the 62 in the given array using linear search.

Program:

```
#include <stdio.h>
int main()
{
  int array[100], search, c, n;
  printf("Enter number of elements in array\n");
  scanf("%d", &n);
  printf("Enter %d integer(s)\n", n);
  for (c = 0; c < n; c++)
  scanf("%d", &array[c]);
  printf("Enter a number to search\n");
  scanf("%d", &search);
  for (c = 0; c < n; c++)
  {
   if (array[c] == search)    /* If required element is found */
  {
   printf("%d is present at location %d.\n", search, c+1);
  break;
  }
 }
 if (c == n)
  printf("%d isn't present in the array.\n", search);
 return 0;
}
```

5. Explain binary search with sample program?

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.
5. Search an ordered array of integers for a value and return its index if the value is found. Otherwise, return -1.

The above example shows the steps to find the element 62 in a given array using binary search.

Binary search is based on the “divide-and-conquer” strategy which works as follows:

n Start by looking at the middle element of the array

– 1. If the value it holds is lower than the search element, eliminate the first half of the array from further consideration.

– 2. If the value it holds is higher than the search element, eliminate the second half of the array from further consideration.

Repeat this process until the element is found, or until the entire array has been eliminated

Algorithm:

Set first and last boundary of array to be searched

Repeat the following:

```
    Find middle element between first and last boundaries;
    if (middle element contains the search value)
  return middle_element position;
    else if (first >= last )
  return –1;
    else if (value < the value of middle_element)
            set last to middle_element position – 1;
```

```
        else
      set first to middle_element position + 1;
```

Program : Binary Search

```
#include <stdio.h>
int main()
{
  int i, first, last, middle, n, search, array[100];
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
                scanf("%d", &array[i]);
          printf("Enter value to find\n");
          scanf("%d", &search);
first = 0;
          last = n - 1;
          middle = (first+last)/2;
while (first <= last)
{
if (array[middle] < search)
    {
          first = middle + 1;
    }
else if (array[middle] == search)
   {
   printf("%d found at location %d.\n", search, middle+1);
           break;
   }
else
last = middle - 1;
   middle = (first + last)/2;
  }
  if (first > last)
   printf("Not found! %d isn't present in the list.\n", search);
  return 0;
}
```

6. Explain time complexity and space complexity?

Algorithm Efficiency

- Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.
- The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.
- Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm
- Time complexity
- Time Complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

- "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

- Space complexity
- Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
- We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.
- We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.
- Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

7. Explain single dimension array?

- An array is a collection of elements of the same type that are referenced by a common name.
- Compared to the basic data type (int, float & char) it is an aggregate or derived data type.
- All the elements of an array occupy a set of contiguous memory locations.

One Dimensional Array: Declaration

Dimension refers to the array's size, which is how big the array is. A single or one dimensional array declaration has the following form,

array_element_data_type array_name[array_size];

Here, array_element_data_type define the base type of the array, which is the type of each element in the array. array_name is any valid C / C++ identifier name that obeys the same rule for the identifier naming. array_size defines how many elements the array will hold.

For example, to declare an array of 30 characters, that construct a people name, we could declare,

```
char   cName[30];
```

Which can be depicted as follows,

In this statement, the array character can store up to 30 characters with the first character occupying location cName[0] and the last character occupying cName[29]. Note that the index runs from 0 to 29. In C, an index always starts from 0 and ends with array's (size-1). So, take note the difference between the array size and subscript/index terms.

Examples of the one-dimensional array declarations,

```
int      xNum[20], yNum[50];
float    fPrice[10], fYield;
char     chLetter[70];
```

The first example declares two arrays named xNum and yNum of type int. Array xNum can store up to 20 integer numbers while yNum can store up to 50 numbers. The second line declares the array fPrice of type float. It can store up to 10 floating-point values. fYield is basic variable which shows array type can be declared together with basic type provided the type is similar. The third line declares the array chLetter of type char. It can store a string up to 69 characters. Why 69 instead of 70? Remember, a string has a null terminating character (\0) at the end, so we must reserve for it.

Array Initialization

- An array may be initialized at the time of declaration.
- Giving initial values to an array.
- Initialization of an array may take the following form,
- type array_name[size] = {a_list_of_value};
- For example:
- int idNum[7] = {1, 2, 3, 4, 5, 6, 7};
- float fFloatNum[5] = {5.6, 5.7, 5.8, 5.9, 6.1};
- char chVowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};

- The first line declares an integer array idNum and it immediately assigns the values 1, 2, 3, ..., 7 to idNum[0], idNum[1], idNum[2],..., idNum[6] respectively.
- The second line assigns the values 5.6 to fFloatNum[0], 5.7 to fFloatNum[1], and so on.
- Similarly the third line assigns the characters 'a' to chVowel[0], 'e' to chVowel[1], and so on. Note again, for characters we must use the single apostrophe/quote (') to enclose them.
- Also, the last character in chVowel is NULL character ('\0').

Initialization of an array of type char for holding strings may take the following form,
char array_name[size] = "string_lateral_constant";
 For example, the array chVowel in the previous example could have been written more compactly as follows,
char chVowel[6] = "aeiou";
 When the value assigned to a character array is a string (which must be enclosed in double quotes), the compiler automatically supplies the NULL character but we still have to reserve one extra place for the NULL.
 For unsized array (variable sized), we can declare as follow,
char chName[ ] = "Mr. Dracula";
 C compiler automatically creates an array which is big enough to hold all the initializer.

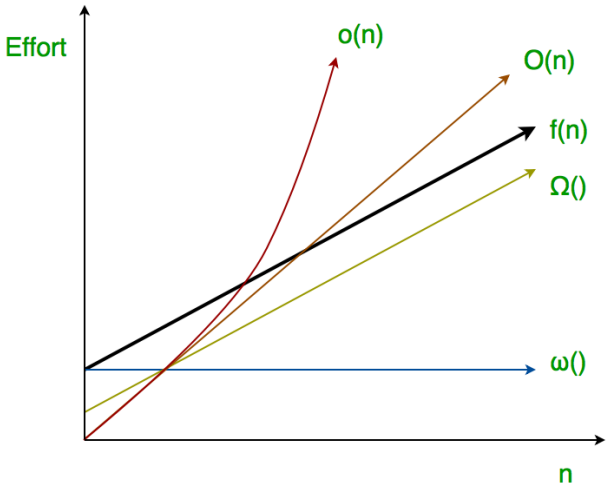
8. Explain little O and little ω notations?

The following 2 more asymptotic notations are used to represent time complexity of algorithms.

Little o asymptotic notation

Big-O is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function f(n)), even though, as written, it can also be a loose upper-bound. "Little-o" (o()) notation is used to describe an upper-bound that cannot be tight.

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) E o(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that 0 ≤ f(n) < c*g(n).
Thus, little o() means loose upper-bound of f(n). Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.



In mathematical relation,
f(n) = o(g(n)) means
lim f(n)/g(n) = 0
n→∞

Examples:

Is 7n + 8 ∈ o(n2)?
In order for that to be true, for any c, we have to be able to find an n0 that makes f(n) < c * g(n) asymptotically true.
lets took some example,
If c = 100,we check the inequality is clearly true. If c = 1/100 , we'll have to use a little more imagination, but we'll be able to find an n0. (Try n0 = 1000.) From these examples, the conjecture appears to be correct.
then check limits,
lim f(n)/g(n) = lim (7n + 8)/(n2) = lim 7/2n = 0 (l'hospital)
n→∞ n→∞ n→∞

hence 7n + 8 ∈ o(n2)

Little ω asymptotic notation

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is ω(g(n)) (or f(n) ∈ ω(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n) > c * g(n) ≥ 0 for every integer n ≥ n0.

f(n) has a higher growth rate than g(n) so main difference between Big Omega (Ω) and little omega (ω) lies in their definitions.In the case of Big Omega f(n)=Ω(g(n)) and the bound is 0<=cg(n)<=f(n), but in case of little omega, it is true for 0<=c*g(n)<f(n).

The relationship between Big Omega (Ω) and Little Omega (ω) is similar to that of Big-O and Little o except that now we are looking at the lower bounds. Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth. We use ω notation to denote a lower bound that is not asymptotically tight.
And, f(n) ∈ ω(g(n)) if and only if g(n) ∈ o((f(n)).

In mathematical relation,
if f(n) ∈ ω(g(n)) then,

lim f(n)/g(n) = ∞
n→∞

Example:
Prove that 4n + 6 ∈ ω(1);
the little omega(o) running time can be proven by applying limit formula given below.
if lim f(n)/g(n) = ∞ then functions f(n) is ω(g(n))
n→∞
here,we have functions f(n)=4n+6 and g(n)=1
lim (4n+6)/(1) = ∞
n→∞
and,also for any c we can get n0 for this inequality 0 <= c*g(n) < f(n), 0 <= c*1 < 4n+6
Hence proved.

9. Explain lower bound and upper bound theory of algorithm?

The Lower and Upper Bound Theory provides a way to find the lowest complexity algorithm to solve a problem. Before understanding the theory, first lets have a brief look on what actually Lower and Upper bounds are.

Lower Bound –
Let L(n) be the running time of an algorithm A(say), then g(n) is the Lower Bound of A if there exist two constants C and N such that L(n) >= C*g(n) for n > N. Lower bound of an algorithm is shown by the asymptotic notation called Big Omega (or just Omega).
Upper Bound –
Let U(n) be the running time of an algorithm A(say), then g(n) is the Upper Bound of A if there exist two constants C and N such that U(n) <= C*g(n) for n > N. Upper bound of an algorithm is shown by the asymptotic notation called Big Oh(O) (or just Oh).
1. Lower Bound Theory:
According to the lower bound theory, for a lower bound L(n) of an algorithm, it is not possible to have any other algorithm (for a common problem) whose time complexity is less than L(n) for random input. Also every algorithm must take at least L(n) time in worst case. Note that L(n) here is the minimum of all the possible algorithm, of maximum complexity.

The Lower Bound is a very important for any algorithm. Once we calculated it, then we can compare it with the actual complexity of the algorithm and if their order are same then we can declare our algorithm as optimal. So in this section we will be discussing about techniques for finding the lower bound of an algorithm.

Note that our main motive is to get an optimal algorithm, which is the one having its Upper Bound Same as its Lower Bound (U(n)=L(n)). Merge Sort is a common example of an optimal algorithm.

Trivial Lower Bound –
It is the easiest method to find the lower bound. The Lower bounds which can be easily observed on the basis of the number of input taken and the number of output produces are called Trivial Lower Bound.

Example: Multiplication of n x n matrix, where,

Input: For 2 matrix we will have 2n2 inputs
Output: 1 matrix of order n x n, i.e., n2 outputs
In the above example its easily predictable that the lower bound is O(n2).

Computational Model –
The method is for all those algorithms that are comparison based. For example in sorting we have to compare the elements of the list among themselves and then sort them accordingly. Similar is the case with searching and thus we can implement the same in this case. Now we will look at some examples to understand its usage.

Ordered Searching –
It is a type of searching in which the list is already sorted.

Example-1: Linear search
Explanation –
In linear search we compare the key with first element if it does not match we compare with second element and so on till we check against the nth element. Else we will end up with a failure.
Using Lower bond theory to solve algebraic problem:

Straight Line Program –
The type of programs build without any loops or control structures is called Straight Line Program. For example,
//summing to nos
Sum(a, b)

```
{
        //no loops and no control structures
        c:= a+b;
        return c;
}
```

Algebraic Problem –

Problems related to algebra like solving equations inequalities etc., comes under algebraic problems. For example, solving equation ax2+bx+c with simple programming.

```
Algo_Sol(a, b, c, x)
{
        //1 assignment
        v:=a*x;

        //1 assignment
        v:=v+b;

        //1 assignment
        v:=v*x;

        //1 assignment
        ans:=v+c;
        return ans;
}
```

The above example shows us a simple way to solve an equation for 2 degree polynomial i.e., 4 thus for nth degree polynomial we will have complexity of O(n2).

Let us demonstrate via an algorithm.

Example: anxn+an-1xn-1+an-2xn-2+…+a1x+a0 is a polynomial of degree n.

```
pow(x, n)
{
        p := 1;

        //loop from 1 to n
        for i:=1 to n
                p := p*x;

        return p;
}

polynomial(A, x, n)
{
        int p, v:=0;
        for i := 0 to n

                //loop within a loop from 0 to n
                v := v + A[i]*pow(x, i);
        return v;
}
```

Loop within a loop => complexity = O(n2);

Now to find an optimal algorithm we need to find the lower bound here (as per lower bound theory). As per Lower Bound Theory, The optimal algorithm to solve the above problem is the one having complexity O(n). Lets prove this theorem using lower bounds.

Theorem: To prove that optimal algo of solving a n degree polynomial is O(n)
Proof: The best solution for reducing the algo is to make this problem less complex by dividing the polynomial into several straight line problems.

=> anxn+an-1xn-1+an-2xn-2+...+a1x+a0
can be written as,
((..(anx+an-1)x+..+a2)x+a1)x+a0
Now, algorithm will be as,
v=0
v=v+an
v=v*x
v=v+an-1
v=v*x
...
v=v+a1
v=v*x
v=v+a0

```
polynomial(A, x, n)
        {
        int p, v=0;

        // loop executed n times
        for i = n to 0
                        v = (v + A[i])*x;

                return v;
        }
```

Clearly, the complexity of this code is O(n). This way of solving such equations is called Horner's method. Here is were lower bound theory works and give the optimum algorithm's complexity as O(n).

2. Upper Bound Theory:
According to the upper bound theory, for an upper bound U(n) of an algorithm, we can always solve the problem in at most U(n) time.Time taken by a known algorithm to solve a problem with worse case input gives us the upper bound.

10. Describe primitive data structure?

A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.
Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
- Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
- Float: It is a data type which use for storing fractional numbers.
- Character: It is a data type which is used for character values.
- Pointer: A variable that holds memory address of another variable are called pointer.

11. Describe non-primitive data structure?

Non primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure
- Array: An array is a fixed-size sequenced collection of elements of the same data type.
- List: An ordered set containing variable number of elements is called as Lists.
- File: A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
- Static memory allocation
- Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear Data Structure are Stack and Queue.

• Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only.

- The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation.
- Stack is also called as Last in First out (LIFO) data structure.
- Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
- End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
- Queue is also called as First in First out (FIFO) data structure.

Nonlinear data structures

Nonlinear data structures are those data structure in which data items are not arranged in a sequence. Examples of Non-linear Data Structure are Tree and Graph.

Tree: A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement. Trees represent the hierarchical relationship between various elements. Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

Graph: Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

o A tree can be viewed as restricted graph.

o Graphs have many types:

- Un-directed Graph
- Directed Graph
- Mixed Graph
- Multi Graph
- Simple GraphNull Graph
- Weighted Graph
-

12. Write a program to sum all elements of array?

```
#include <stdio.h>
int main()
{
   int a[1000],i,n,sum=0;

   printf("Enter size of the array : ");
   scanf("%d",&n);

   printf("Enter elements in array : ");
   for(i=0; i<n; i++)
   {
      scanf("%d",&a[i]);
   }


   for(i=0; i<n; i++)
   {

      sum+=a[i];
   }
    printf("sum of array is : %d",sum);

   return 0;
}
```

13. Write a program to obtain transpose of a matrix?

```
#include <stdio.h>
int main() {
   int a[10][10], transpose[10][10], r, c, i, j;
   printf("Enter rows and columns: ");
   scanf("%d %d", &r, &c);

   // Assigning elements to the matrix
   printf("\nEnter matrix elements:\n");
   for (i = 0; i < r; ++i)
      for (j = 0; j < c; ++j) {
         printf("Enter element a%d%d: ", i + 1, j + 1);
         scanf("%d", &a[i][j]);
      }

   // Displaying the matrix a[][]
   printf("\nEntered matrix: \n");
   for (i = 0; i < r; ++i)
      for (j = 0; j < c; ++j) {
         printf("%d  ", a[i][j]);
         if (j == c - 1)
            printf("\n");
      }

   // Finding the transpose of matrix a
   for (i = 0; i < r; ++i)
      for (j = 0; j < c; ++j) {
```

```
            transpose[j][i] = a[i][j];
        }

    // Displaying the transpose of matrix a
    printf("\nTranspose of the matrix:\n");
    for (i = 0; i < c; ++i)
        for (j = 0; j < r; ++j) {
            printf("%d  ", transpose[i][j]);
            if (j == r - 1)
                printf("\n");
        }
    return 0;
}
```

14. Write a program that will convert 2D matrix representation to Sparse representation?

```
#include <stdio.h>
#define MAX 20

void read_matrix(int a[10][10], int row, int column);
void print_sparse(int b[MAX][3]);
void create_sparse(int a[10][10], int row, int column, int b[MAX][3]);

int main()
{
    int a[10][10], b[MAX][3], row, column;
    printf("\nEnter the size of matrix (rows, columns): ");
    scanf("%d%d", &row, &column);

    read_matrix(a, row, column);
    create_sparse(a, row, column, b);
    print_sparse(b);
    return 0;
}

void read_matrix(int a[10][10], int row, int column)
{
    int i, j;
    printf("\nEnter elements of matrix\n");
    for (i = 0; i < row; i++)
    {
        for (j = 0; j < column; j++)
        {
            printf("[%d][%d]: ", i, j);
            scanf("%d", &a[i][j]);
        }
    }
}

void create_sparse(int a[10][10], int row, int column, int b[MAX][3])
{
    int i, j, k;
    k = 1;
    b[0][0] = row;
```

```
    b[0][1] = column;
    for (i = 0; i < row; i++)
    {
        for (j = 0; j < column; j++)
        {
            if (a[i][j] != 0)
            {
                b[k][0] = i;
                b[k][1] = j;
                b[k][2] = a[i][j];
                k++;
            }
        }
        b[0][2] = k - 1;
    }
}

void print_sparse(int b[MAX][3])
{
    int i, column;
    column = b[0][2];
    printf("\nSparse form - list of 3 triples\n\n");
    for (i = 0; i <= column; i++)
    {
        printf("%d\t%d\t%d\n", b[i][0], b[i][1], b[i][2]);
    }
}
```

15. Write a program for matrix multiplication?

```
#include<stdio.h>
#include<stdlib.h>
int main(){
int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
system("cls");
printf("enter the number of row=");
scanf("%d",&r);
printf("enter the number of column=");
scanf("%d",&c);
printf("enter the first matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("enter the second matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&b[i][j]);
}
```

```
}

printf("multiply of the matrix=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
printf("%d\t",mul[i][j]);
}
printf("\n");
}
return 0;
}
```

16. Explain insertion and deletion operations in array?

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 the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

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]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

For other variations of array insertion operation click here

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the Kth position of LA.

1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop

Example

Following is the implementation of the above algorithm −

```
#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 after deletion :\n");

  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

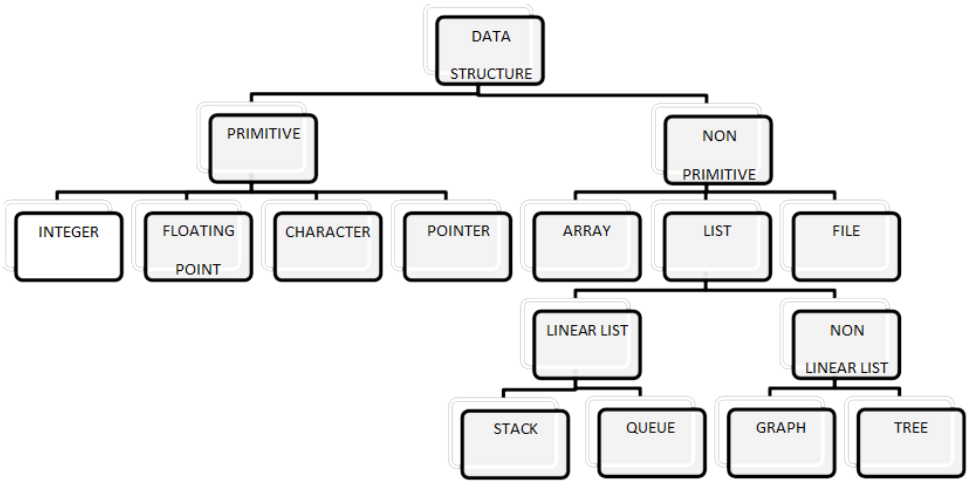
When we compile and execute the above program, it produces the following result −

Output

The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8

10 marks:

1. **Explain classification of datastructures with a neat diagram?**



Data Structures are normally classified into two broad categories

1. Primitive Data Structure

2. Non-primitive data Structure

Data types
A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.
A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.
Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
- Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
- Float: It is a data type which use for storing fractional numbers.
- Character: It is a data type which is used for character values.
- Pointer: A variable that holds memory address of another variable are called pointer.

Non primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure
- Array: An array is a fixed-size sequenced collection of elements of the same data type.
- List: An ordered set containing variable number of elements is called as Lists.
- File: A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
- Static memory allocation
- Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear Data Structure are Stack and Queue.

• Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only.

- The insertion operation is referred to as ‘PUSH’ and deletion operation is referred to as ‘POP’ operation.
- Stack is also called as Last in First out (LIFO) data structure.
- Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
- End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
- Queue is also called as First in First out (FIFO) data structure.

Nonlinear data structures

Nonlinear data structures are those data structure in which data items are not arranged in a sequence. Examples of Non-linear Data Structure are Tree and Graph.

Tree: A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement. Trees represent the hierarchical relationship between various elements. Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

Graph: Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

- o A tree can be viewed as restricted graph.
- o Graphs have many types:
  - Un-directed Graph
  - Directed Graph
  - Mixed Graph
  - Multi Graph
  - Simple Graph Null Graph
  - Weighted Graph

2. Explain array and two dimensional array with diagrams?

- An array is a collection of elements of the same type that are referenced by a common name.
- Compared to the basic data type (int, float & char) it is an aggregate or derived data type.
- All the elements of an array occupy a set of contiguous memory locations.

Why need to use array type?Consider the following issue:

We have a list of 1000 students' marks of an integer type. If using the basic data type (int), we will declare something like the following…"

```
int  studMark0, studMark1, studMark2, ..., studMark999;
```

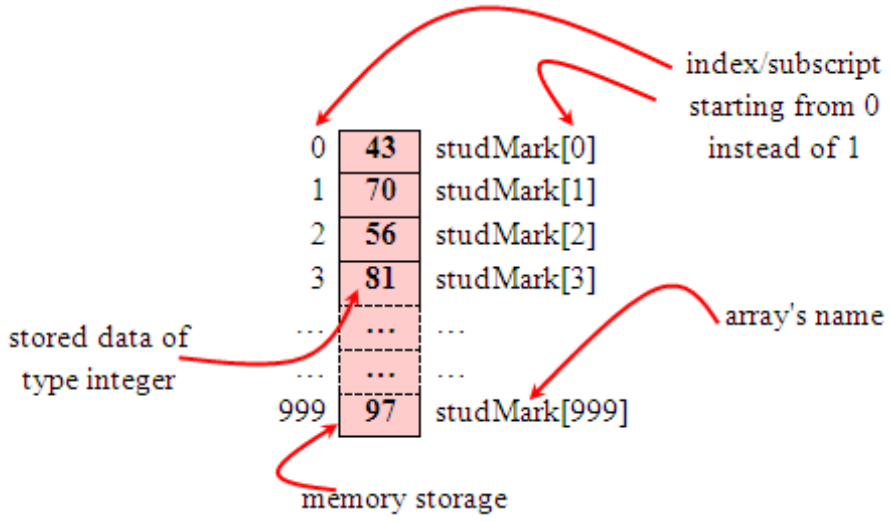
Can you imagine how long we have to write the declaration part by using normal variable declaration?

```
int main(void)
{
int studMark1, studMark2, studMark3, studMark4, …, …, studMark998, stuMark999, studMark1000;
…
…
return 0;
}
```

By using an array, we just declare like this,

```
int  studMark[1000];
```

This will reserve 1000 contiguous memory locations for storing the students' marks. Graphically, this can be depicted as in the following figure.



This absolutely has simplified our declaration of the variables. We can use index or subscript to identify each element or location in the memory. Hence, if we have an index of jIndex, studMark[jIndex] would refer to the jIndexth element in the array of studMark. For example, studMark[0] will refer to the first element of the array. Thus by changing the value of jIndex, we could refer to any element in the array. So, array has simplified our declaration and of course, manipulation of the data.

One Dimensional Array: Declaration

Dimension refers to the array's size, which is how big the array is. A single or one dimensional array declaration has the following form,

array_element_data_type array_name[array_size];
Here, array_element_data_type define the base type of the array, which is the type of each element in the array. array_name is any valid C / C++ identifier name that obeys the same rule for the identifier naming. array_size defines how many elements the array will hold.
For example, to declare an array of 30 characters, that construct a people name, we could declare,
char cName[30];
Which can be depicted as follows,
In this statement, the array character can store up to 30 characters with the first character occupying location cName[0] and the last character occupying cName[29]. Note that the index runs from 0 to 29. In C, an index always starts from 0 and ends with array's (size-1). So, take note the difference between the array size and subscript/index terms.

Examples of the one-dimensional array declarations,
int xNum[20], yNum[50];
float fPrice[10], fYield;
char chLetter[70];
The first example declares two arrays named xNum and yNum of type int. Array xNum can store up to 20 integer numbers while yNum can store up to 50 numbers. The second line declares the array fPrice of type float. It can store up to 10 floating-point values. fYield is basic variable which shows array type can be declared together with basic type provided the type is similar. The third line declares the array chLetter of type char. It can store a string up to 69 characters. Why 69 instead of 70? Remember, a string has a null terminating character (\0) at the end, so we must reserve for it.

Array Initialization
- An array may be initialized at the time of declaration.
- Giving initial values to an array.
- Initialization of an array may take the following form,
- type array_name[size] = {a_list_of_value};
- For example:
- int idNum[7] = {1, 2, 3, 4, 5, 6, 7};
- float fFloatNum[5] = {5.6, 5.7, 5.8, 5.9, 6.1};
- char chVowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
- The first line declares an integer array idNum and it immediately assigns the values 1, 2, 3, ..., 7 to idNum[0], idNum[1], idNum[2],..., idNum[6] respectively.
- The second line assigns the values 5.6 to fFloatNum[0], 5.7 to fFloatNum[1], and so on.
- Similarly the third line assigns the characters 'a' to chVowel[0], 'e' to chVowel[1], and so on. Note again, for characters we must use the single apostrophe/quote (') to enclose them.
- Also, the last character in chVowel is NULL character ('\0').

Initialization of an array of type char for holding strings may take the following form,
char array_name[size] = "string_lateral_constant";
- For example, the array chVowel in the previous example could have been written more compactly as follows,

char chVowel[6] = "aeiou";
- When the value assigned to a character array is a string (which must be enclosed in double quotes), the compiler automatically supplies the NULL character but we still have to reserve one extra place for the NULL.
- For unsized array (variable sized), we can declare as follow,

char chName[ ] = "Mr. Dracula";
- C compiler automatically creates an array which is big enough to hold all the initializer.

Two Dimensional Array

A two dimensional array has two subscripts/indexes. The first subscript refers to the row, and the second, to the column. Its declaration has the following form,

data_type array_name[1st dimension size][2nd dimension size];

For examples,

int xInteger[3][4];

float matrixNum[20][25];

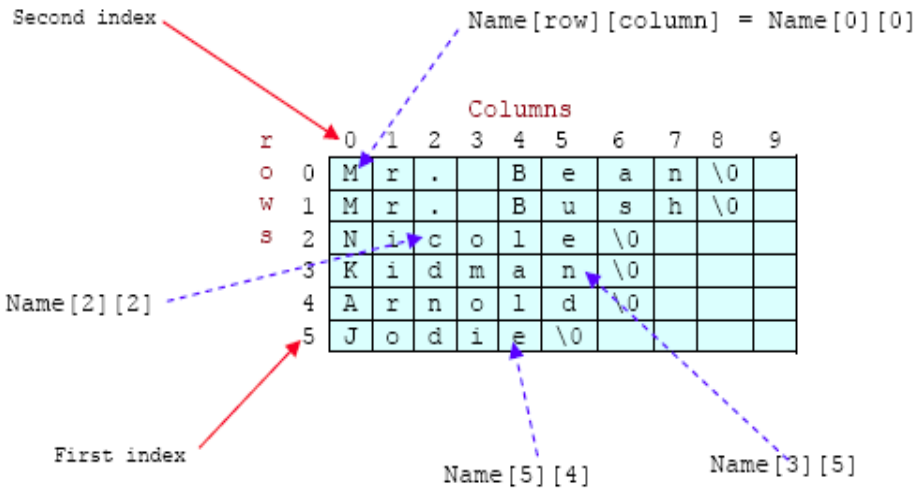
The first line declares xInteger as an integer array with 3 rows and 4 columns.

Second line declares a matrixNum as a floating-point array with 20 rows and 25 columns.

If we assign initial string values for the 2D array it will look something like the following,

char Name[6][10] = {"Mr. Bean", "Mr. Bush", "Nicole", "Kidman", "Arnold", "Jodie"};

Here, we can initialize the array with 6 strings, each with maximum 9 characters long. If depicted in rows and columns it will look something like the following and can be considered as contiguous arrangement in the memory.



- Take note that for strings the null character (\0) still needed.
- From the shaded square area of the figure we can determine the size of the array.
- For an array Name[6][10], the array size is 6 x 10 = 60 and equal to the number of the colored square. In general, for
- array_name[x][y];
- The array size is = First index x second index = xy.
- This also true for other array dimension, for example three dimensional array,
- array_name[x][y][z]; => First index x second index x third index = xyz
- For example,
- ThreeDimArray[2][4][7] = 2 x 4 x 7 = 56.
- And if you want to illustrate the 3D array, it could be a cube with wide, long and height dimensions.

3. Explain the various operation on array with sample programs?

Following are the basic operations supported by an array.

- Traverse − print all the array elements one by one.
- Insertion − Adds an element at the given index.
- Deletion − Deletes an element at the given index.
- Search − Searches an element using the given index or by the value.
- Update − Updates an element at the given index.

Traverse Operation

This operation is to traverse through the elements of an array.

Example

Following program traverses and prints the elements of an array:

```
#include <stdio.h>
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]);
  }
}
```

When we compile and execute the above program, it produces the following result −

Output

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

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 the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

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]);
  }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
```

```
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

For other variations of array insertion operation click here

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the Kth position of LA.

1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop

Example

Following is the implementation of the above algorithm −

```
#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 after deletion :\n");

  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
```

```
  }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#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\n", item, j+1);
}
```

When we compile and execute the above program, it produces the following result −

Output

```
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

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

1. Start
2. Set LA[K-1] = ITEM
3. Stop

Example

Following is the implementation of the above algorithm −

```
#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]);
   }
}
```

When we compile and execute the above program, it produces the following result −

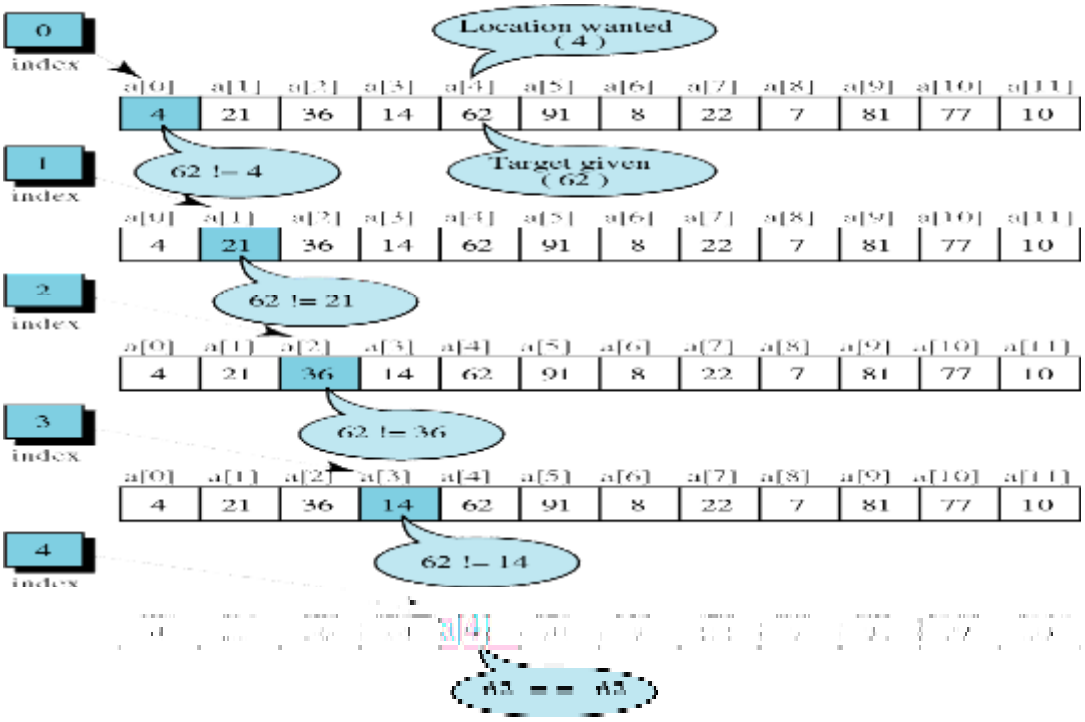
Output

```
The original array 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
```

4. Explain linear search and binary search?

Linear Search

Linear search in C is to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.



Above diagram shows the simple example to find the 62 in the given array using linear search.

Program:

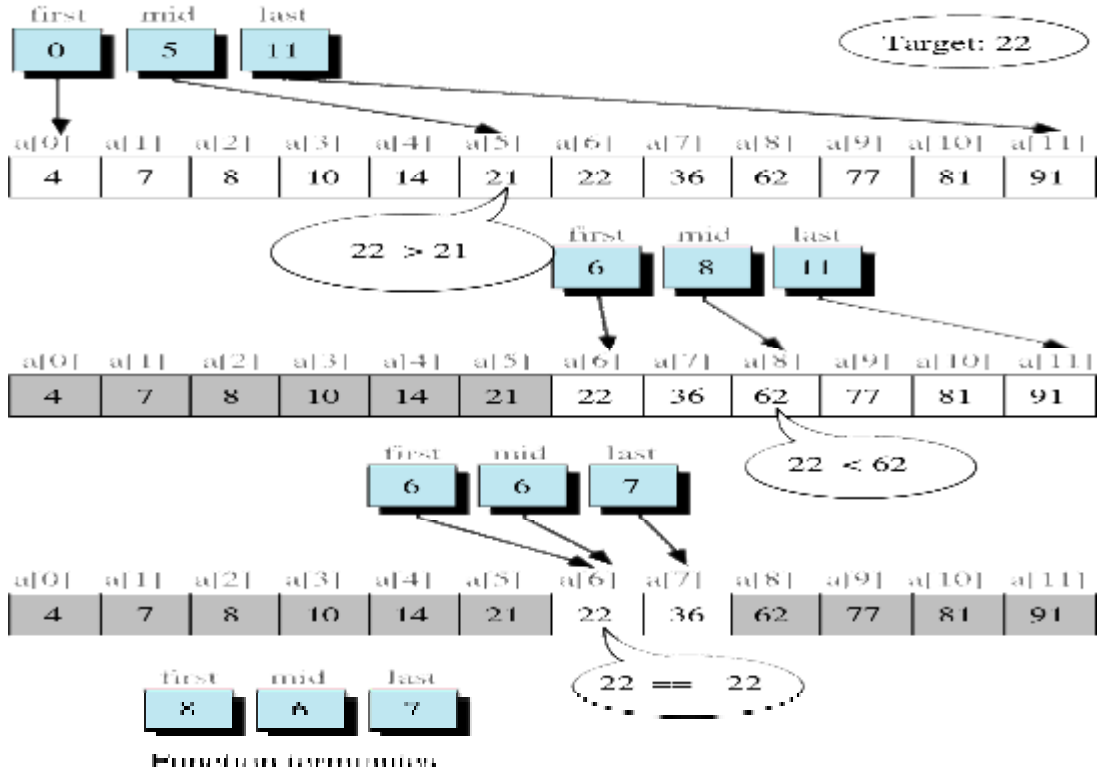
```
#include <stdio.h>
int main()
{
   int array[100], search, c, n;
   printf("Enter number of elements in array\n");
   scanf("%d", &n);
   printf("Enter %d integer(s)\n", n);
   for (c = 0; c < n; c++)
   scanf("%d", &array[c]);
   printf("Enter a number to search\n");
   scanf("%d", &search);
   for (c = 0; c < n; c++)
   {
    if (array[c] == search)    /* If required element is found */
   {
    printf("%d is present at location %d.\n", search, c+1);
   break;
   }
  }
 if (c == n)
  printf("%d isn't present in the array.\n", search);
 return 0;
}
```

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.
5. Search an ordered array of integers for a value and return its index if the value is found. Otherwise, return -1.



The above example shows the steps to find the element 62 in a given array using binary search.

Binary search is based on the “divide-and-conquer” strategy which works as follows:

n Start by looking at the middle element of the array

– 1. If the value it holds is lower than the search element, eliminate the first half of the array from further consideration.

– 2. If the value it holds is higher than the search element, eliminate the second half of the array from further consideration.

Repeat this process until the element is found, or until the entire array has been eliminated

Algorithm:

Set first and last boundary of array to be searched

Repeat the following:

```
    Find middle element between first and last boundaries;
    if (middle element contains the search value)
  return middle_element position;
    else if (first >= last )
  return –1;
    else if (value < the value of middle_element)
            set last to middle_element position – 1;
```

```
        else
       set first to middle_element position + 1;
```

Program : Binary Search

```
#include <stdio.h>
int main()
{
  int i, first, last, middle, n, search, array[100];
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
                scanf("%d", &array[i]);
        printf("Enter value to find\n");
        scanf("%d", &search);
first = 0;
        last = n - 1;
        middle = (first+last)/2;
while (first <= last)
{
if (array[middle] < search)
    {
        first = middle + 1;
    }
else if (array[middle] == search)
   {
   printf("%d found at location %d.\n", search, middle+1);
         break;
   }
else
last = middle - 1;
   middle = (first + last)/2;
  }
  if (first > last)
   printf("Not found! %d isn't present in the list.\n", search);
  return 0;
}
```

5. Explain complexity analysis (time and space) and best, average and worst case analysis?

- Time complexity
- Time Complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

- Space complexity
- Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
- We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

 We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.

 Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

Worst Case Analysis

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be.

Average Case Analysis

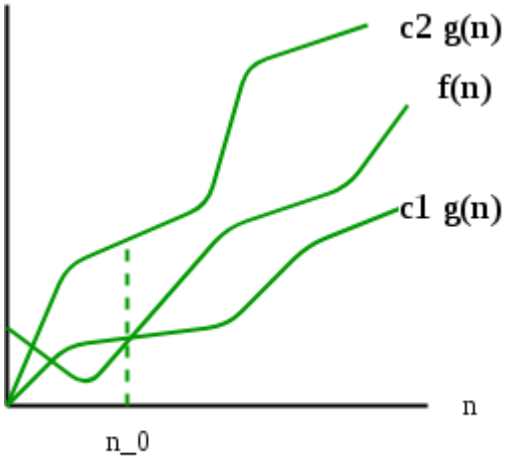
In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by (n+1).

Best Case Analysis

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.

6. Write about asymptotic notations and their properties?

Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.



f(n) = theta(g(n))

1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.

A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

3n3 + 6n2 + 6000 = Θ(n3)

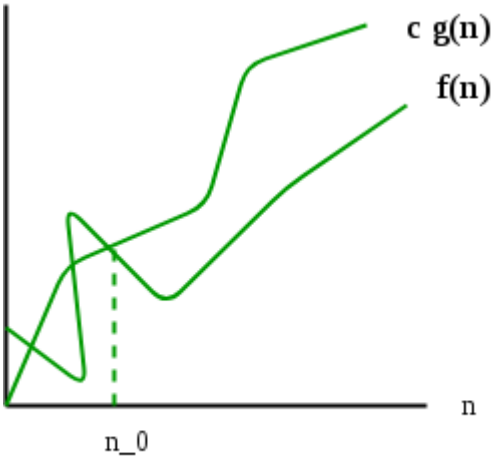
Dropping lower order terms is always fine because there will always be a n0 after which Θ(n3) has higher values than Θn2) irrespective of the constants involved.

For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such

that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



f(n) = O(g(n))

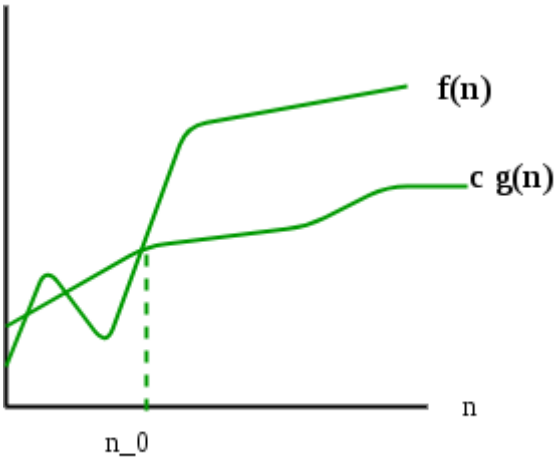
2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and
n0 such that 0 <= f(n) <= c*g(n) for
all n >= n0}



f(n) = Omega(g(n))

3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and
n0 such that 0 <= c*g(n) <= f(n) for
all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

Properties of Asymptotic Notations :
As we have gone through the definition of this three notations let's now discuss some important properties of those notations.

General Properties :
If f(n) is O(g(n)) then a*f(n) is also O(g(n)) ; where a is a constant.

Example: f(n) = $2n^2+5$ is $O(n^2)$
then 7*f(n) = $7(2n^2+5)$
= $14n^2+35$ is also $O(n^2)$

Similarly this property satisfies for both Θ and Ω notation.
We can say
If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)) ; where a is a constant.
If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)) ; where a is a constant.

Reflexive Properties :
If f(n) is given then f(n) is O(f(n)).

Example: f(n) = n^2 ; $O(n^2)$ i.e O(f(n))

Similarly this property satisfies for both Θ and Ω notation.
We can say
If f(n) is given then f(n) is Θ(f(n)).
If f(n) is given then f(n) is Ω (f(n)).

Transitive Properties :
If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

Example: if f(n) = n , g(n) = n^2 and h(n)=n^3
n is $O(n^2)$ and n^2 is $O(n^3)$
then n is $O(n^3)$

Similarly this property satisfies for both Θ and Ω notation.
We can say
If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

Symmetric Properties :
If f(n) is $\Theta(g(n))$ then g(n) is $\Theta(f(n))$.

Example: $f(n) = n^2$ and $g(n) = n^2$
then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$

This property only satisfies for Θ notation.

Transpose Symmetric Properties :
If f(n) is $O(g(n))$ then g(n) is $\Omega (f(n))$.

Example: $f(n) = n$, $g(n) = n^2$
then n is $O(n^2)$ and n^2 is $\Omega (n)$

This property only satisfies for O and Ω notations.

Some More Properties :
If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$
If $f(n) = O(g(n))$ and $d(n)=O(e(n))$
then $f(n) + d(n) = O(\max(g(n), e(n)))$
Example: $f(n) = n$ i.e $O(n)$
$d(n) = n^2$ i.e $O(n^2)$
then $f(n) + d(n) = n + n^2$ i.e $O(n^2)$
If $f(n)=O(g(n))$ and $d(n)=O(e(n))$
then $f(n) * d(n) = O(g(n) * e(n))$
Example: $f(n) = n$ i.e $O(n)$
$d(n) = n^2$ i.e $O(n^2)$
then $f(n) * d(n) = n * n^2 = n^3$ i.e $O(n^3)$

SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution & Accredited by NAAC with "A" Grade)
(An Autonomous Institution)
Madagadipet, Puducherry - 605 107

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject Name: **Data Structures**
Subject Code:

Prepared by:

Verified by: **Approved by:**

UNIT – II

Stacks and Queues: ADT Stack and its operations, Applications of Stacks: Expression Conversion and evaluation. ADT Queue: Types of Queue: Simple Queue, Circular Queue, Priority Queue. Operations on each type of Queues.

2 MARKS

1. What is Stack?
- Stack is a linear and static data structure.
- Stack is an ordered collection of elements in which we can insert and delete the elements at one end called Top.
- Initial condition of the stack Top=-1.
- It is otherwise called as LIFO (Last In First Out).

2. What are the various operations that can be performed on stack?
In Stack, we can perform two operations namely Push and Pop.
Push: Push means inserting a new element into the stack. Insertion can be done by incrementing top by 1.
Pop: Pop means deleting an element from the stack. Deletion can be done by decrementing top by 1.
Peek: Returns the top element and an error occurs if the stack is empty.

3. What do you mean by Top in Stack?
- Top is the pointer which always points the top element of the Stack.
- If Top =-1, then Stack is Empty.
- We can insert a new element into stack by incrementing top by 1.
- We can delete an element from stack by decrementing top by 1.

4. What are the applications of Stack?
Some of the applications of Stack are:
i. Matching of nested parenthesis in a mathematical expression.
ii. Conversion of infix to postfix.
iii. Evaluation of postfix.

5. What is ADT in data structure with example?
The **Data** Type is basically a type of **data** that can be used in different computer program. The **ADT** is made of with primitive datatypes, but operation logics are hidden. Some **examples** of **ADT** are Stack, Queue, List etc.

6. Why stack and queue is called ADT?
Stack and Queue are referred as abstract datatype because in stack there are, mainly two operations push and pop and in queue there are insertion and deletion. Which are when operated on any set of data then it is free from that which type of data that must be contained by the set.

7. What are types of Expression?

Expression is classified as three types according to position of operator with respect to the operands. They are:

i. **Infix**: The operator is placed in between two operands. E.g.: A+B.
ii. **Postfix**: The operator is placed after the operands. E.g.: AB+.
iii. **Prefix**: The operator is placed before the operands. E.g.: +AB.

8. What is Queue?

- Queue is a linear and static data structure.
- Queue is an ordered collection of elements in which we can insert an element at one end called Rear and delete an element at another end called Front.
- Initial condition of the stack Rear=Front=-1. It is otherwise called as FIFO (First In First Out).

9. What are the various operations that can be performed on Queue?

In Queue, we can perform two operations namely Insertion and Deletion.

Insertion: Insertion can be done by incrementing Rear by 1.

Deletion: Deletion can be done by incrementing Front by 1

10. What do you mean by Queue empty?

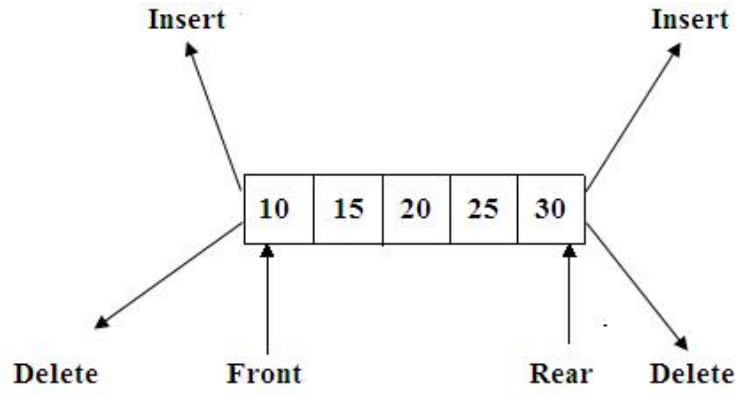
- If Queue has no data, then it is called as Queue Empty.
- Queue may be empty on two conditions.
  i. Front==Rear==-1 and
  ii. Front==Rear.

11. Difference between Stack and Queue.

Stack	Queue
➢ A Stack is an ordered collection of data items, where all insertion and deletions always are made at the end of the sequence called Top.	➢ A Queue is an ordered collection of items, where all insertions are made at the end of the sequence called Rear and all deletions always are made from the beginning of the sequence called Front.
➢ In Stack, we get data items out in reverse order compared to the order they have been put into the stack.	➢ In Queue, we get data items out in same order compared to the order they have been put into the queue.
➢ Stack is otherwise called as FIFO.	➢ Queue is otherwise called as LIFO.

12. Define Dequeue?

- Dequeue is otherwise called as Double ended queue.
- In Dequeue, we can insert and delete at both ends either front or rear.



13. What are the types of Queue?

- Simple Queue
- Circular Queue
- Priority Queue
- Double ended Queue

14. What is difference between Queue and Dequeue?

- A queue is designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue.
- Whereas Dequeue represents a queue where you can insert and remove elements from both ends of the queue.

15. What is Priority Queue with example?

- A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority.
- Generally, the value of the element itself is considered for assigning the priority.
- For example: The element with the highest value is considered as the highest priority element.

16. Define Ascending Priority Queue.

- In this elements are placed in ascending order.
- The first smallest element is placed in first position and second smallest element in second position and so on.
- The new data item is inserted in priority queue without affecting the ascending order of queue.

17. Define Descending Priority Queue.

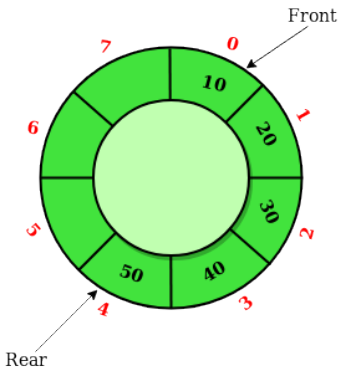
- In this elements are placed in descending order.
- The first highest element is placed in first position and second highest element is placed in second position and so on.
- The new data item is inserted in priority queue without affecting the descending order of queue.

18. What are the application of Priority Queue?

- Priority queues are used to sort heaps.
- Priority queues are used in operating system for load balancing and interrupt handling.
- Priority queues are used in huffman codes for data compression.
- In traffic light, depending upon the traffic, the colors will be given priority.

19. What is Circular Queue?

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle.
- The last position is connected back to the first position to make a circle.
- Circular Queue is used in memory management and scheduling process.
- It is also called 'Ring Buffer'.



20. How do you determine the size of Circular Queue?

Assuming you are using array of size N for queue implementation, then size of queue would be **size = (N-front + rear) mod N**. This formula work for both liner and circular queues.

21. How a stack is implemented using queue?

A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1′ and 'q2′. Stack 's' can be implemented in two ways:

- Method 1 (By making push operation costly)
- Method 2 (By making pop operation costly)

22. Write the limitations of stack.

- Only a small number of operations can be performed on it.
- It contains only a bounded capacity

23. Write the conditions to test "Queue is empty", "Queue is full" for a linear queue implemented in linear array?

If REAR = N (no. of elements), Queue is full

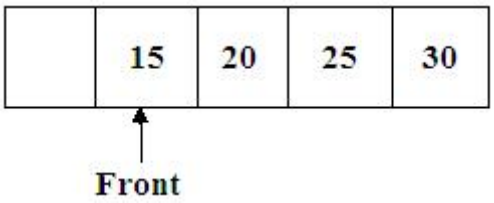
If FRONT =REAR, Queue is empty.

24. How to insert an element at the beginning of the Dequeue.

- First check the Dequeue is full or not.
- We have to check whether the element is first to be inserted if it is true then perform the following steps
  1. Shift the elements from left to right
  2. So that we need the number of elements present in the Dequeue , the position of the last element i.e. the position of the rear.
  3. Now insert any element in zeroth position.

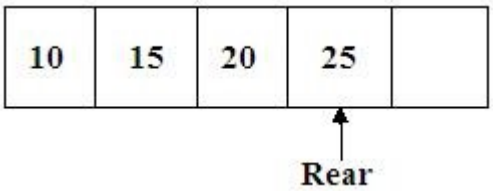
25. How to delete an element at the beginning of Dequeue.

- Delete the first element from the front position of the Dequeue
- The index of next element is stored in front pointer.
- Increment the front pointer by one.



26. How to delete an element at the end of Dequeue.

- The last element is deleted.
- The index of next element is stored in rear pointer.
- Decrement the rear pointer by one.



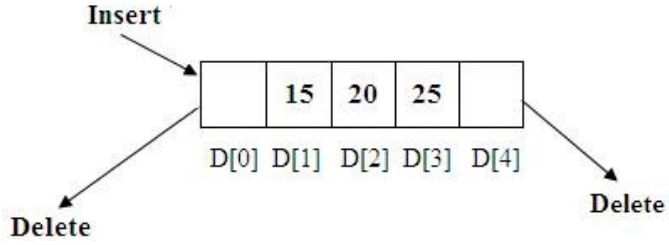
27. What are the types of Dequeue?

Two types of Dequeue are

1. Input Restricted Dequeue
2. Ouput Restricted Dequeue.

28. Define Input restricted Dequeue.

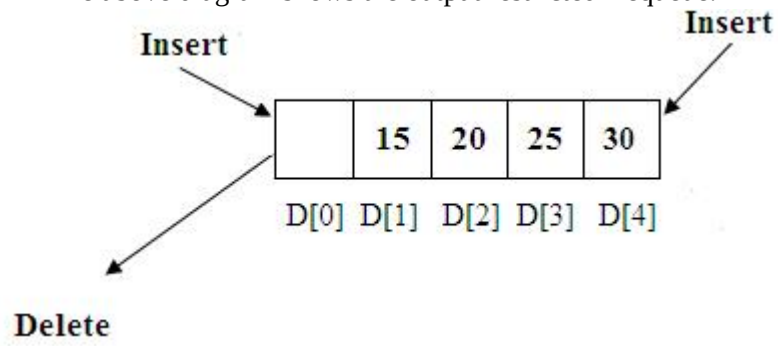
- It means we can insert the element only at one end and delete the elements at both ends.
- The above diagram shows the input restricted Dequeue.



29. Define Output restricted Dequeue.

- It means we can insert the elements in both ends and delete the elements at only one end.

- The above diagram shows the output restricted Dequeue.



30. State the advantages of using infix notations.

The advantages of using infix notations are,

- It is the mathematical way of representing the expression.
- It is easier to see visually which operation is done from first to last.

31. State the advantages of using postfix notations.

The advantages of using infix notations are

- We need not worry about the rules of precedence.
- We need to worry about the rules for right to left associatively.
- We need not parenthesis to override the above rules.

32. Write down the application of queue.

- Printing
- CPU scheduling
- Mail service
- Elevator
- Keyboard buffering

5 Marks

1. What is Stack? Explain with an example.

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Operations:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

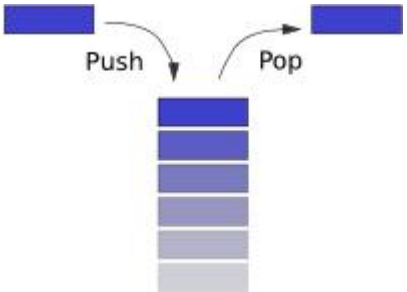


Fig: Simple representation of a Stack

Example for Stack:

Converting a decimal number into a binary number:

To solve this problem, we use a stack. We make use of the *LIFO* property of the stack. Initially we *push* the binary digit formed into the stack, instead of printing it directly. After the entire digit has been converted into the binary form, we *pop* one digit at a time from the stack and print it. Therefore, we get the decimal number is converted into its proper binary form.

Algorithm:

```
      1. Create a stack
      2. Enter a decimal number which has to be converted into its
equivalent binary form.
      3. iteration1 (while number > 0)
            3.1 digit = number % 2
            3.2 Push digit into the stack
            3.3 If the stack is full
                  3.3.1 Print an error
                  3.3.2 Stop the algorithm
            3.4 End the if condition
            3.5 Divide the number by
      4. End iteration1
      5. iteration2 (while stack is not empty
            5.1 Pop digit from the stack
            5.2 Print the digit
      6. End iteration2
      7. STOP
```

2. Explain the push operation on stack.

Stack operations may involve initializing the stack, using it and then de-initalizing it. Appart from these basic stuffs, a stack is used for the following two primary operations –

- **push() –** Pushing (storing) an element on the stack.
- **pop() –** Removing (accessing) an element from the stack.

Push Operation:

The process of putting a new data element onto stack is known as Push Operation. Push operation involves a series of steps –

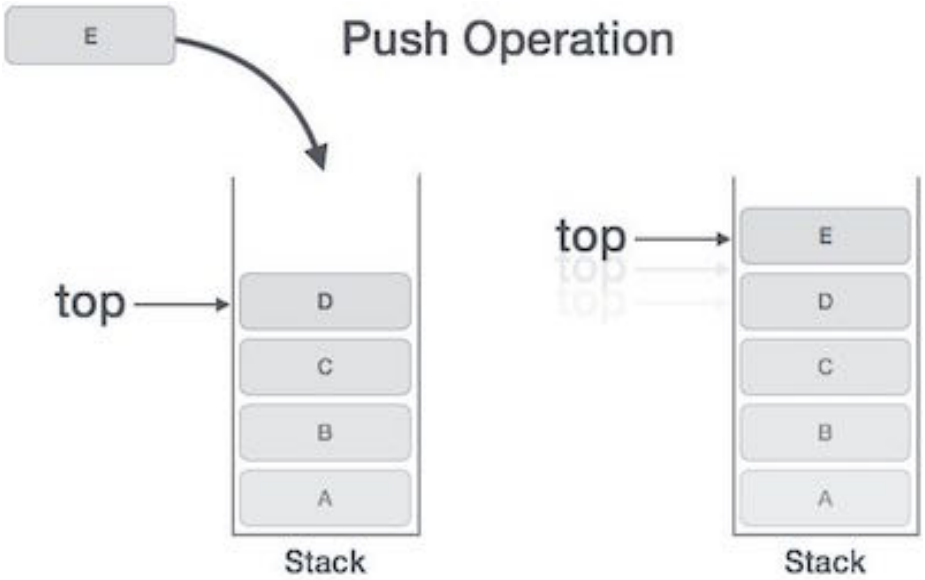
Step 1 - Checks if the stack is full.

Step 2 - If the stack is full, produces an error and exit.

Step 3 - If the stack is not full, increments **top** to point next empty space.

Step 4 - Adds data element to the stack location, where top is pointing.

Step 5 - Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation:

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

       if stack is full
              return null
       endif

       top ← top + 1
       stack[top] ← data

end procedure
```

Example:

```
void push(int data) {
     if(!isFull()){
          top += 1;
          stack[top] = data;
     } else{
          printf("Could not insert, Stack  is full. \n");
     }
}
```

3. Explain the pop operation on stack.

Stack operations may involve initializing the stack, using it and then de-initalizing it. Appart from these basic stuffs, a stack is used for the following two primary operations –

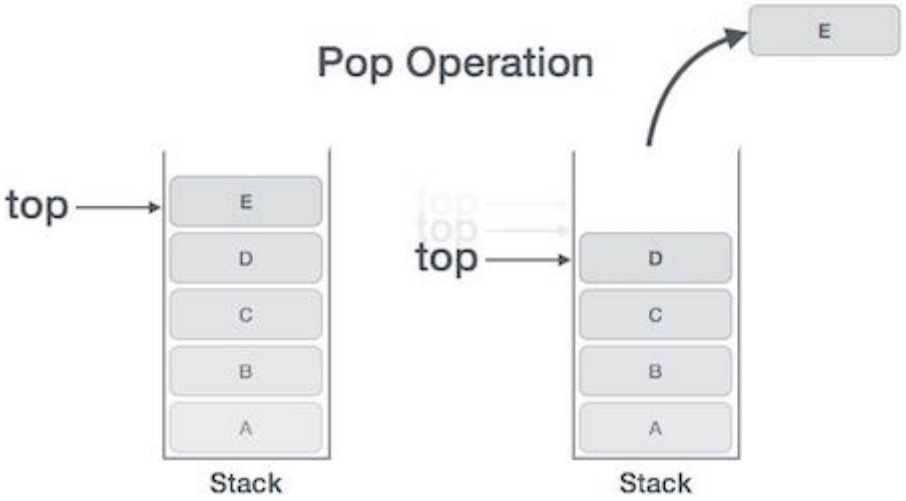
- **push() –** Pushing (storing) an element on the stack.
- **pop() –** Removing (accessing) an element from the stack.

POP Operation:

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation:

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack
     if stack is empty
          return null
     end if

     data ← stack[top]
     top ← top – 1
     return data
```

Implementation of this algorithm in C, is as follows –

Example:

```
int pop(int data){
     if(!isempty()){
          data = stack[top];
          top = top – 1;
          return data;
     }else{
          printf("Could not retrieve, Stack is empty.\n");
     } }
```

4. **Explain the conversion of Infix to Postfix expression using Algorithm**

Infix expression: The expression of the form a op b. When an operator in-between every pair of operands.

Postfix expression: The expression of the form **a b op.** When an operator is followed for every pair of operands.

EVALUATION OF POSTFIX EXPRESSION USING STACK:

1. initialize empty stack with top=-1.
2. scan the given postfix expression from left to right till the last character of the expression.
3. if the character is operand than push it into the stack.
4. if the operator is unary operator then pop up one operand from the stack.
5. if the operator is binary operator then pop two operator from the stack and perform the operation and store the result in the stack.
6. repeat 3&4 till the last character of the postfix expression .

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where ^ is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

5. Show how the fundamental operation of a queue can be implemented.

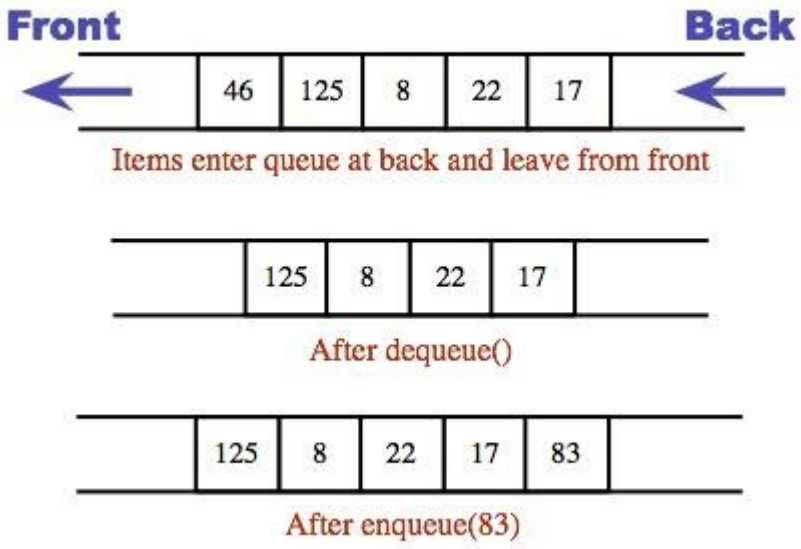
QUEUE:

A queue is an ordered collection of items from which items may be deleted at one end called the front and the items may be inserted at the other end called rear of the queue.

PRINCIPLE:

The first element inserted into a queue is the first element to be removed. Queue is called First In First Out (FIFO) list.

BASIC OPERATIONS INVOLVED IN A QUEUE:



1.Create a queue
2.Check whether a queue is empty or full
3.Add an item at the rear end

4.Remove an item at the

front end

5.Read the front of the

queue

6.Print the entire queue

INSERTION OPERATION

An attempt to push an item onto a queue, when the queue is full, causes an overflow.

1. Check whether the queue is full before attempting to insert another element.

2. Increment the rear pointer & 3. Insert the element at the rear pointer of the queue.

ALGORITHM:

Rear – Rear end pointer, Q – Queue, N – Total number of elements & Item – The element to be inserted

1. if(Rear=N)

[Overflow?] Then

Call QUEUE_FULL

Return

2. Rear<-Rear+1 [Increment rear pointer]

3. Q[Rear]<-Item [Insert element]

End INSERT

DELETION OPERATION:

An attempt to remove an element from the queue when the queue is empty causes an underflow.

Deletion operation involves:

1. Check whether the queue is empty.

2. Increment the front pointer.

3. Remove the element.

ALGORITHM:

Q – Queue, Front – Front end pointer , Rear – Rear end pointer & Item – The element to be deleted.

1.if(Front=Rear) [Underflow?]

Then Call QUEUE_EMPTY

2. Front<-Front+1 [Incrementation]

3. Item<-Q [Front] [Delete element]

Thus queue is a dynamic structure that is constantly allowed to grow and shrink and thus changes its size, when implemented using linked list.

6. Explain the different types of Queue.

QUEUE:

A queue is an ordered collection of items from which items may be deleted at one end called the front and the items may be inserted at the other end called rear of the queue.

There are four different types of queue in data structure.

- Simple queue
- Circular queue
- Priority queue
- Double Ended queue or Dequeue

Simple Queue:

In a simple queue, insertion takes palce at the rear and removal occurs at the front. It strictly follows FIFO rule.

Queue Specifications

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations.

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue

- **IsEmpty**: Check if the queue is empty.
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

Circular Queue:

In a circular queue, the last element points to the first element making a circular link.

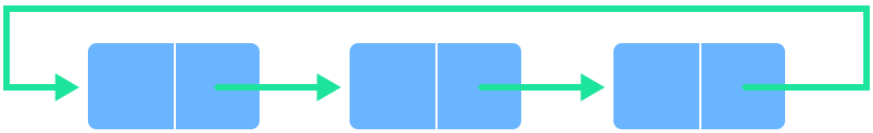


Fig: Circular Queue

The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty then, an element can be inserted in the first position. This action is not possible in a simple queue.

Priority Queue:

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

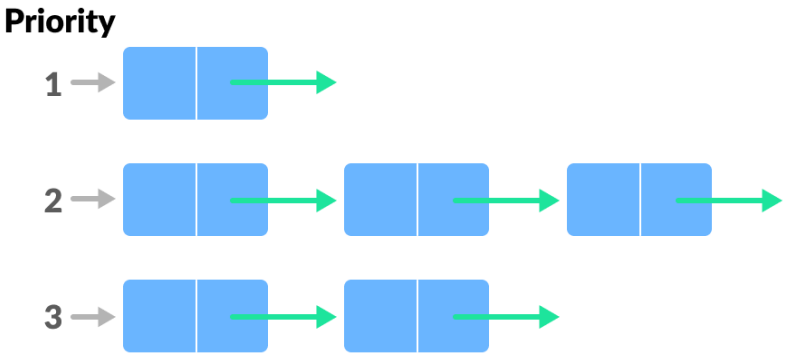


Fig: Priority Queue

Insertion occurs based on the arrival of the values and removal occurs based on priority.

Double Ended queue or Dequeue:

Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).

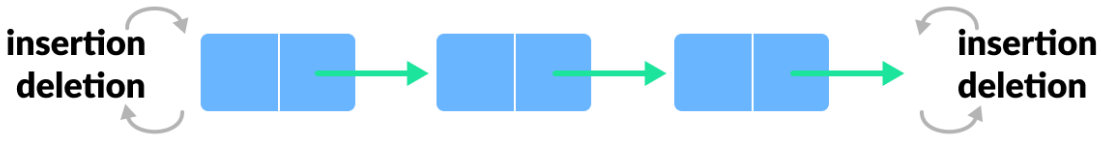
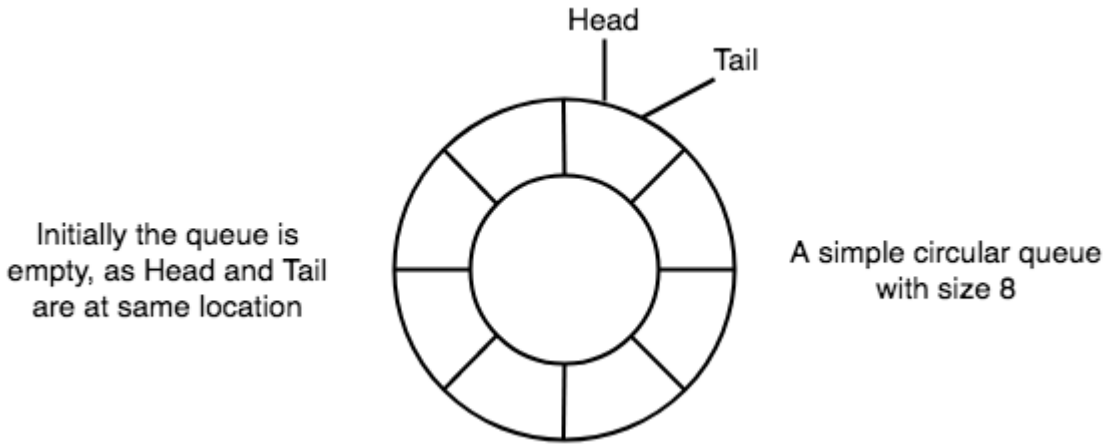


Fig: Dequeue

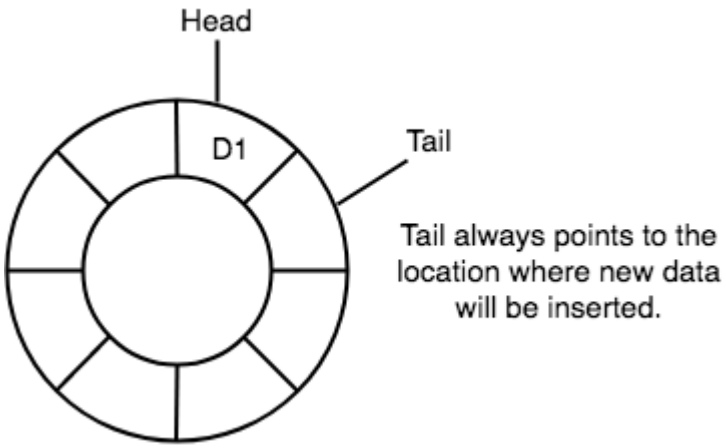
7. Explain Circular Queue with an example.

Circular Queue is also a linear data structure, which follows the principle of **FIFO** (First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

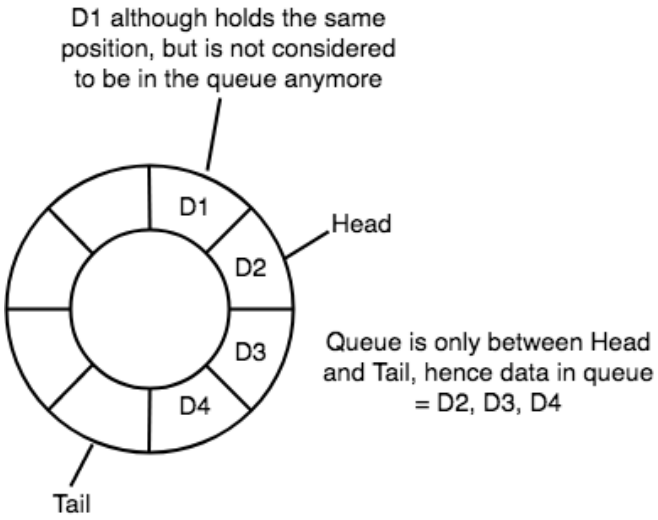
- In case of a circular queue, `head` pointer will always point to the front of the queue, and `tail` pointer will always point to the end of the queue.
- Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.



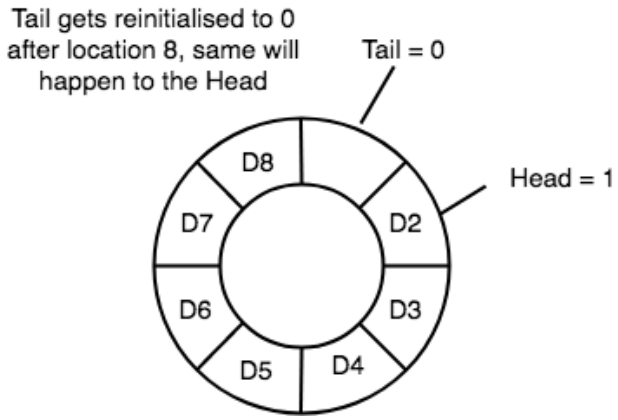
- New data is always added to the location pointed by the `tail` pointer, and once the data is added, `tail` pointer is incremented to point to the next available location.



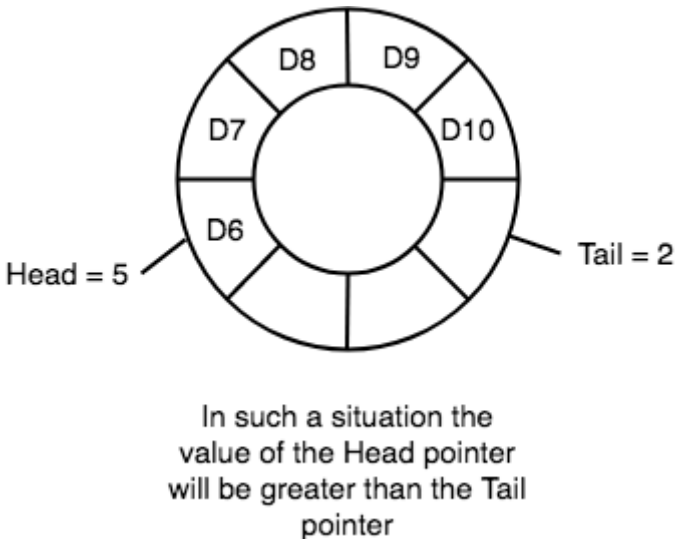
- In a circular queue, data is not actually removed from the queue. Only the `head` pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between `head` and `tail`, hence the data left outside is not a part of the queue anymore, hence removed.



- The `head` and the `tail` pointer will get reinitialised to **0** every time they reach the end of the queue.



- Also, the `head` and the `tail` pointers can cross each other. In other words, `head` pointer can be greater than the `tail`. Sounds odd? This will happen when we dequeue the queue a couple of times and the `tail` pointer gets reinitialised upon reaching the end of the queue.



Implementation:

Below we have the implementation of a circular queue:

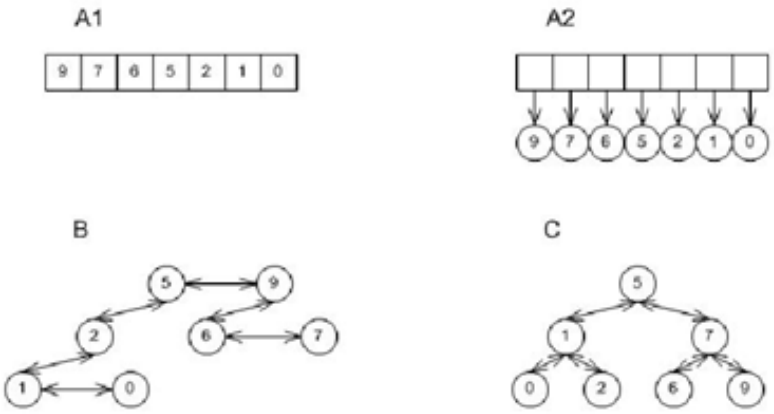
1. Initialize the queue, with size of the queue defined (maxSize), and head and tail pointers.
2. enqueue: Check if the number of elements is equal to maxSize - 1:
    - If **Yes**, then return **Queue is full**.
    - If **No**, then add the new data element to the location of tail pointer and increment the tail pointer.
3. dequeue: Check if the number of elements in the queue is zero:
    - If **Yes**, then return **Queue is empty**.
    - If **No**, then increment the head pointer.
4. Finding the size:
    - If, **tail >= head**, size = (tail - head) + 1
    - But if, **head > tail**, then size = maxSize - (head - tail) + 1

8. Explain Priority Queue with an example.

PRIORITY QUEUES

Priority queues *are a kind of queue in which the elements are dequeued in priority order.*

A priority queue is a collection of elements where each element has an associated priority. Elements are added and removed from the list in a manner such that the element with the highest (or lowest) priority is always the next to be removed. When using a heap mechanism, a priority queue is quite efficient for this type of operation.



- Each element has a priority, an element of a totally ordered set (usually a number).
- More important things come out first, even if they were added later.
- Three types of priority. Low priority [10], Normal Priority [5] and High Priority [1].
- There is no (fast) operation to find out whether an arbitrary element is in the queue.

ALGORITHM:

Priority Queue - Algorithms - Adjust

Adjust(i)

left = 2i, right = 2i + 1

if left <= H.size and H[left] > H[i]

```
    then largest = left
    else largest = i
if right <= H.size and H[right] > H[largest]
    then largest = right
if largest != i
    then swap H[largest] with H[i]
        Adjust(largest)
```

Explanation:

Adjust works recursively to guarantee the heap property. It compares the current node with its children finding which, if either, has a greater priority. If one of them does, it will swap array locations with the largest child. Adjust is then run again on the current node in its new location.

Priority Queue - Algorithms - Insert

```
Insert(Key)
        H.size = H.size + 1
        i = H.size
        while i > 1 and H[i/2] < Key
                H[i] = H[i/2]
                i = i/2
        end while
        H[i] = key
```

Explanation

The insert algorithm works by first inserting the new element (key) at the end of the array. This element is then compared with its parent for highest priority. If the parent has a lower priority, the two elements are swapped. This process is repeated until the new element has found its place.

Priority Queue - Algorithms - Extract

```
Extract()
    Max = H[1] H[1] = H[H.size]
    H.size = H.size – 1
     Adjust(1) Return
    Max
```

Explanation

Extract works by removing and returning the first array element, the one of highest priority, and then promoting the last array element to the first. Adjust is then run on the first element so that the heap property is maintained.

Run Time Complexity

Θ(lg *n*) time for Insert and worst case for Extract (where *n* is number of elements)

Θ(lg *n*) average time for Insert

Can construct a Heap from a list in Θ(lg *n*) where a Binary Search Tree takes Θ(*n* lg *n*)

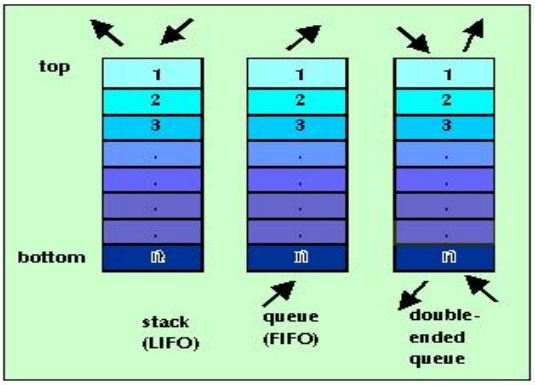
Space Requirements

All operations are done in place - space requirement at any given time is the number of elements, *n*.

9. Write an algorithm for a doubly ended queue for insertion and deletion.

Double Ended Queue

A deque *(short for double-ended queue) is an abstract data structure for which elements can be added to or removed from the front or back(both end). This differs from a normal queue, where elements can only be added to one end and removed from the other. Both queues and stacks can be considered specializations of deques, and can be implemented using deques.*



double-ended queue

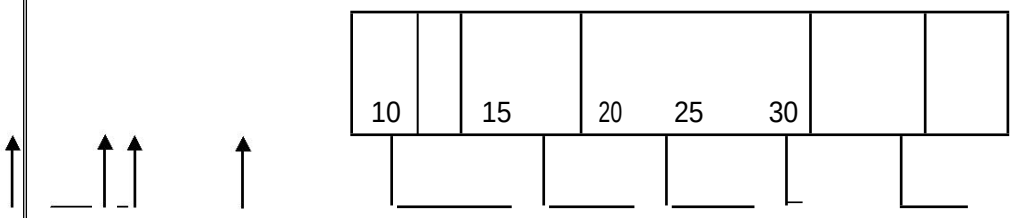
INSERT NEW ELEMENT INTO DEQUEUE:

AT THE END OF THE DEQUEUE:

- *First check the dequeue is full or not.*
- *We have to check whether the element is first to be added or inserted if it is true assign front and rear is 0.*
- *Insert a new element*

AT THE BEGINNING OF THE DEQUEUE:
First check the dequeue is full or not.

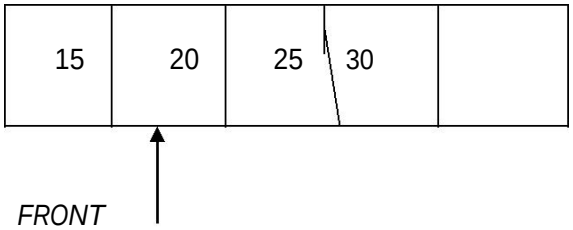
- *We have to check whether the element is first to be inserted if it is true then perform the following steps*
  1. *shift the elements from left to right*
  2. *so that we need the number of elements present in the dequeue ,the position of the last element ie the position of the rear*
  3. *now insert any element in zeroth position*



DELETING THE ELEMENT FROM DEQUEUE:

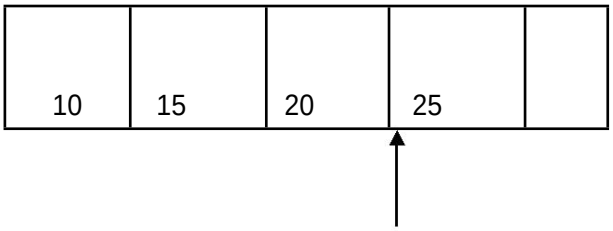
AT THE BEGINNING OF DEQUEUE:

- *delete the first element from the front position of the dequeue*
- *the index of next elem*
- *.ent is stored in front pointer.*
- Increment the front pointer by one .



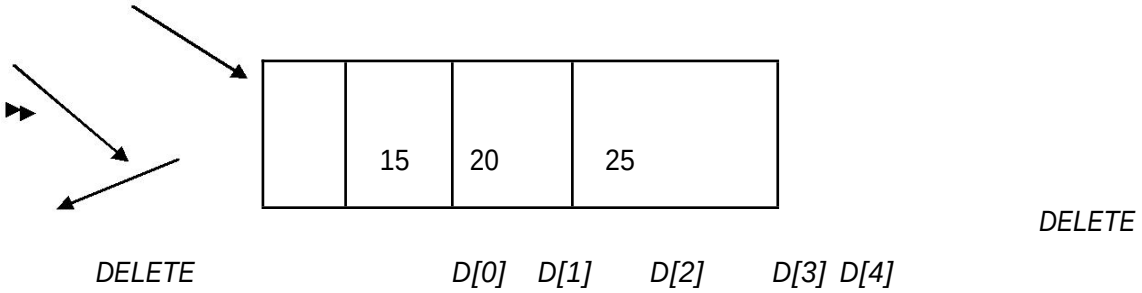
AT THE END OF DEQUEUE:

- The last element is deleted .
- The index of next element is stored in rear pointer
- Decrement the rear pointer by one.



INPUT RESTRICTED DEQUEUE:

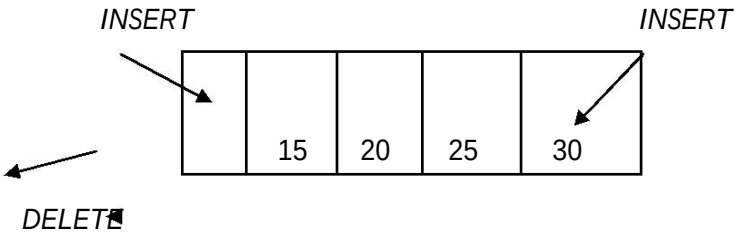
It means we can insert the element only at one end and delete the elements at both ends.



- *The above diagram shows the input restricted dequeue.*

OUTPUT RESTRICTED DEQUEUE:

It means we can insert the elements in both ends and delete the elements at only one end.



The above diagram shows the output restricted dequeue.

10. Write an algorithm for peek(), isFull() and isEmpty() operations of queue.

The supportive functions are required to make the a queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
Being procedure isfull
        If rear equals to MAZSIZE
                return true
        else
                return false
        endif
end procedure
```

isempty()

Algorithm of isempty() function −

Algorithm

```
Being procedure isfull
        If front is less than MIN or front is greater than rear
                return true
        else
                return false
        endif
end procedure
```

11. Difference between Stack and queue.

STACKS	QUEUES
Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.

STACKS	QUEUES
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.
Stack is used in solving problems works on recursion.	Queue is used in solving problems having sequential processing.

12. Comparison between Linear Queue and Circular Queue.

LINEAR QUEUE	CIRCULAR QUEUE
Organizes the data elements and instructions in a sequential order one after the other.	Arranges the data in the circular pattern where the last element is connected to the first element.
Tasks are executed in order they were placed before (FIFO).	Order of executing a task may change.
The new element is added from the rear end and removed from the front.	Insertion and deletion can be done at any position.

LINEAR QUEUE	CIRCULAR QUEUE
The linear queue is an ordered list in which data elements are organized in the sequential order	In contrast, circular queue stores the data in the circular fashion.
Linear queue follows the FIFO order for executing the task (the element added in the first position is going to be deleted in the first position)	Conversely, in the circular queue, the order of operations performed on an element may change.
Inefficient	Works better than the linear queue.

13. Write note on Applications of Stack.

The Stack is Last In First Out (LIFO) data structure. This data structure has some important applications in different aspect. These are like below −

- Expression Handling −
  - Infix to Postfix or Infix to Prefix Conversion −

    The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions. These expressions are not so much familiar to the infix expression, but they have some great advantages also. We do not need to maintain operator ordering, and parenthesis.
  - Postfix or Prefix Evaluation −

    After converting into prefix or postfix notations, we have to evaluate the expression to get the result. For that purpose, also we need the help of stack data structure.
- Backtracking Procedure −

  Backtracking is one of the algorithm designing technique. For that purpose, we dive into some way, if that way is not efficient, we come back to the previous state and go into some other paths. To get back from current state, we need to store the previous state. For that purpose, we need stack. Some examples of backtracking is finding the solution for Knight Tour problem or N-Queen Problem etc.

- Another great use of stack is during the function call and return process. When we call a function from one other function, that function call statement may not be the first statement. After calling the function, we also have to come back from the function area to the place, where we have left our control. So we want to resume our task, not restart. For that reason, we store the address of the program counter into the stack, then go to the function body to execute it. After completion of the execution, it pops out the address from stack and assign it into the program counter to resume the task again.

14. Write an example program for Stack using array.

```
// C program for array implementation of stack
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a stack

 struct Stack {
    int top;
    unsigned capacity;
    int* array;
 };

 // function to create a stack of given capacity. It initializes size of
 // stack as 0
 struct Stack* createStack(unsigned capacity)
 {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
 }

 // Stack is full when top is equal to the last index
 int isFull(struct Stack* stack)
 {
    return stack->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 to stack.  It increases top by 1
 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 from stack.  It decreases top by 1
 int pop(struct Stack* stack)
 {
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
 }

 // Function to return the top from stack without removing it
 int peek(struct Stack* stack)
 {
    if (isEmpty(stack))
        return INT_MIN;
```

```
    return stack->array[stack->top];
}

// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));

    return 0;
}
```

Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 popped from stack
```

15. What is Stack? Write the time complexities of stack operations and applications of Stack.

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Applications of stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver
- In Graph Algorithms like Topological Sorting and Strongly Connected Components

10 MARKS

1. **What is stack? Write the algorithm with an example program.**

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Operations:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

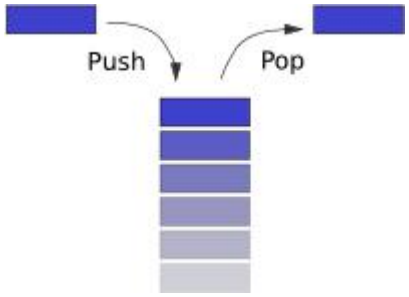


Fig: Simple representation of a Stack

Example for Stack:

Converting a decimal number into a binary number:

To solve this problem, we use a stack. We make use of the *LIFO* property of the stack. Initially we *push* the binary digit formed into the stack, instead of printing it directly. After the entire digit has been converted into the binary form, we *pop* one digit at a time from the stack and print it. Therefore, we get the decimal number is converted into its proper binary form.

Algorithm:

```
      1. Create a stack
      2. Enter a decimal number which has to be converted into its
equivalent binary form.
      3. iteration1 (while number > 0)
            3.1 digit = number % 2
            3.2 Push digit into the stack
            3.3 If the stack is full
                  3.3.1 Print an error
                  3.3.2 Stop the algorithm
            3.4 End the if condition
            3.5 Divide the number by
      4. End iteration1
      5. iteration2 (while stack is not empty
            5.1 Pop digit from the stack
            5.2 Print the digit
      6. End iteration2
      7. STOP
```

```
// C program for array implementation of stack
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a stack

 struct Stack {
    int top;
    unsigned capacity;
```

```
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
    return stack->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 to stack.  It increases top by 1
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 from stack.  It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);
```

```
    printf("%d popped from stack\n", pop(stack));

    return 0;
}
```

Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 popped from stack
```

2. Explain both push and pop operation on stack.

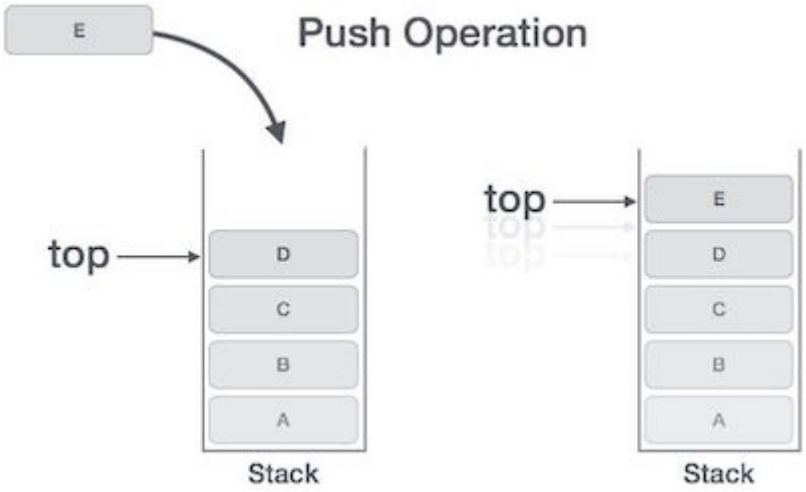
Stack operations may involve initializing the stack, using it and then de-initalizing it. Appart from these basic stuffs, a stack is used for the following two primary operations –

- **push() –** Pushing (storing) an element on the stack.
- **pop() –** Removing (accessing) an element from the stack.

Push Operation:

The process of putting a new data element onto stack is known as Push Operation. Push operation involves a series of steps –

Step 1 - Checks if the stack is full.
Step 2 - If the stack is full, produces an error and exit.
Step 3 - If the stack is not full, increments **top** to point next empty space.
Step 4 - Adds data element to the stack location, where top is pointing.
Step 5 - Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation:

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

        if stack is full
                return null
        endif

        top ← top + 1
        stack[top] ← data

end procedure
```

Example:

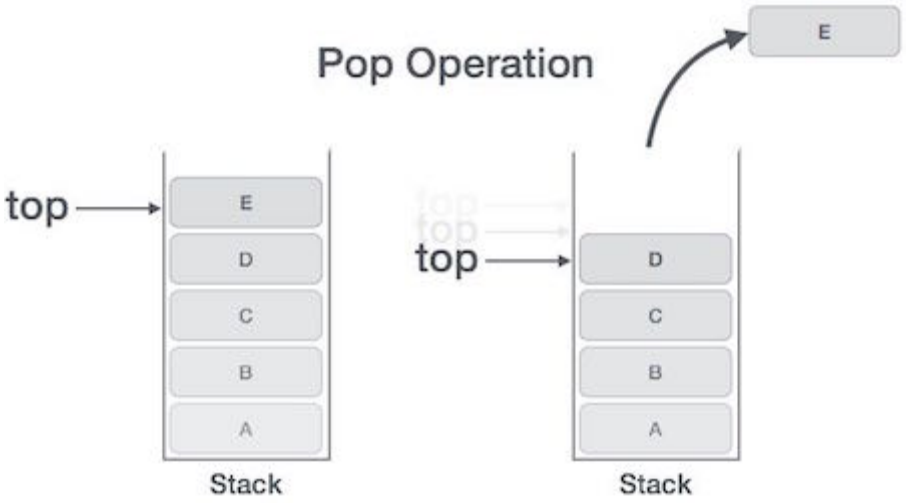
```
void push(int data) {
      if(!isFull()){
            top += 1;
            stack[top] = data;
      } else{
            printf("Could not insert, Stack  is full. \n");
      }
}
```

POP Operation:

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation:

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack
      if stack is empty
            return null
      end if

      data ← stack[top]
```

```
top ← top – 1
return data
```

Implementation of this algorithm in C, is as follows –

Example:

```
int pop(int data){
     if(!isempty()){
          data = stack[top];
          top = top – 1;
          return data;
     }else{
          printf("Could not retrieve, Stack is empty.\n");
     } }
```

3. How the fundamental operations of a queue can be implemented and explain the types of queue in detail.

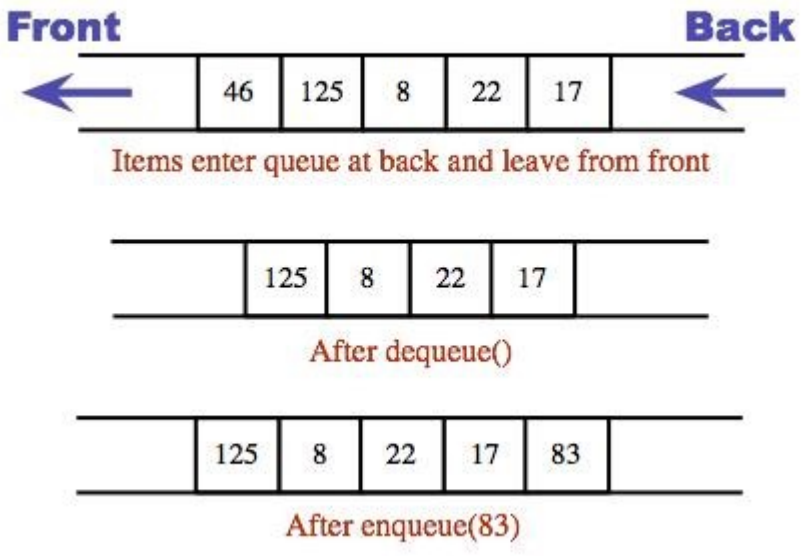
QUEUE:

A queue is an ordered collection of items from which items may be deleted at one end called the front and the items may be inserted at the other end called rear of the queue.

PRINCIPLE:

The first element inserted into a queue is the first element to be removed. Queue is called First In First Out (FIFO) list.

BASIC OPERATIONS INVOLVED IN A QUEUE:



1.Create a queue

2.Check whether a queue is empty or full

3.Add an item at the rear end

4.Remove an item at the

front end

5.Read the front of the

queue

6.Print the entire queue

INSERTION OPERATION

An attempt to push an item onto a queue, when the queue is full, causes an overflow.

3. Check whether the queue is full before attempting to insert another element.

4. Increment the rear pointer & 3. Insert the element at the rear pointer of the queue.

ALGORITHM:

Rear – Rear end pointer, Q – Queue, N – Total number of elements & Item – The element to be inserted

4. if(Rear=N)

[Overflow?] Then

Call QUEUE_FULL

Return

5. Rear<-Rear+1 [Increment rear pointer]

6. Q[Rear]<-Item [Insert element]

End INSERT

DELETION OPERATION:

An attempt to remove an element from the queue when the queue is empty causes an underflow.

Deletion operation involves:

4. Check whether the queue is empty.

5. Increment the front pointer.

6. Remove the element.

ALGORITHM:

Q – Queue, Front – Front end pointer , Rear – Rear end pointer & Item – The element to be deleted.

1.if(Front=Rear) [Underflow?]

Then Call QUEUE_EMPTY

4. Front<-Front+1 [Incrementation]

5. Item<-Q [Front] [Delete element]

Thus queue is a dynamic structure that is constantly allowed to grow and shrink and thus changes its size, when implemented using linked list.

There are four different types of queue in data structure.

- Simple queue
- Circular queue
- Priority queue
- Double Ended queue or Dequeue

Simple Queue:

In a simple queue, insertion takes palce at the rear and removal occurs at the front. It strictly

follows FIFO rule.

Queue Specifications

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations.

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty.
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

Circular Queue:

In a circular queue, the last element points to the first element making a circular link.

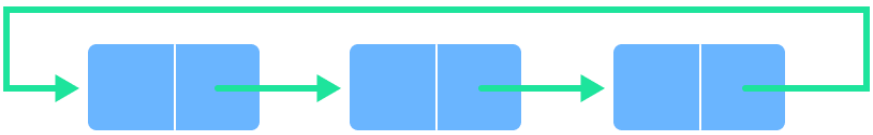


Fig: Circular Queue

The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty then, an element can be inserted in the first position. This action is not possible in a simple queue.

Priority Queue:

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

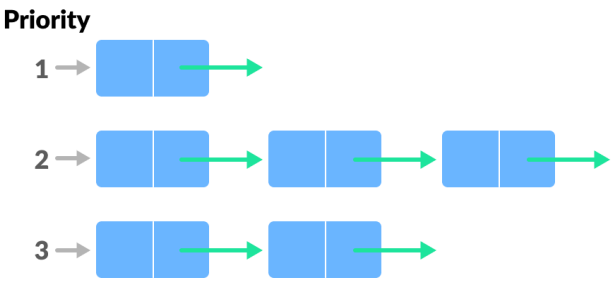


Fig: Priority Queue

Insertion occurs based on the arrival of the values and removal occurs based on priority.

Double Ended queue or Dequeue:

Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).

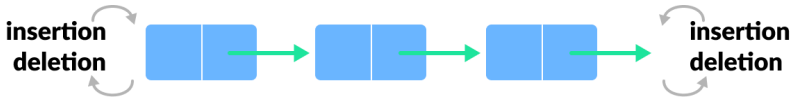
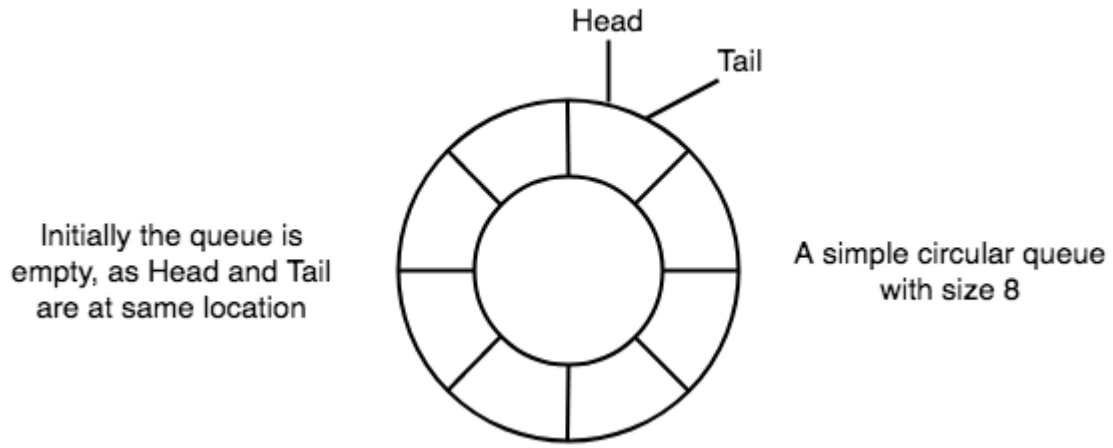


Fig: Dequeue

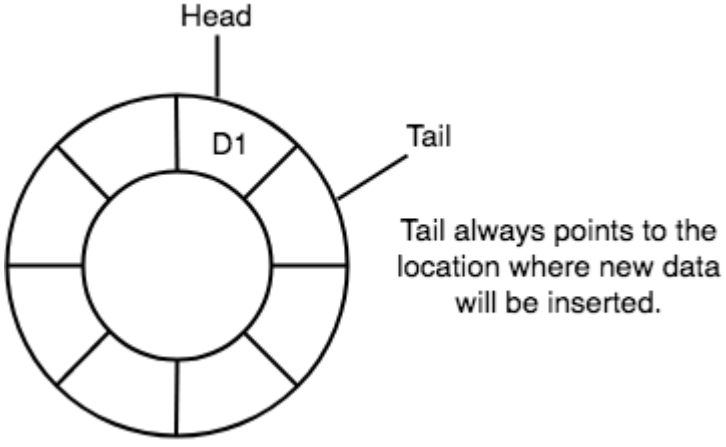
4.Explain Circular queue and Priority queue in detail.

Circular Queue is also a linear data structure, which follows the principle of **FIFO** (First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

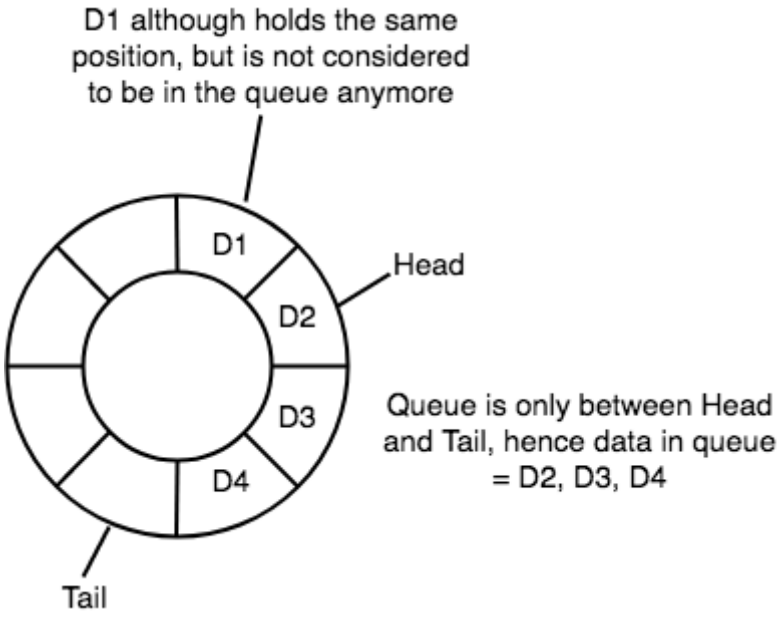
- In case of a circular queue, `head` pointer will always point to the front of the queue, and `tail` pointer will always point to the end of the queue.
- Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.



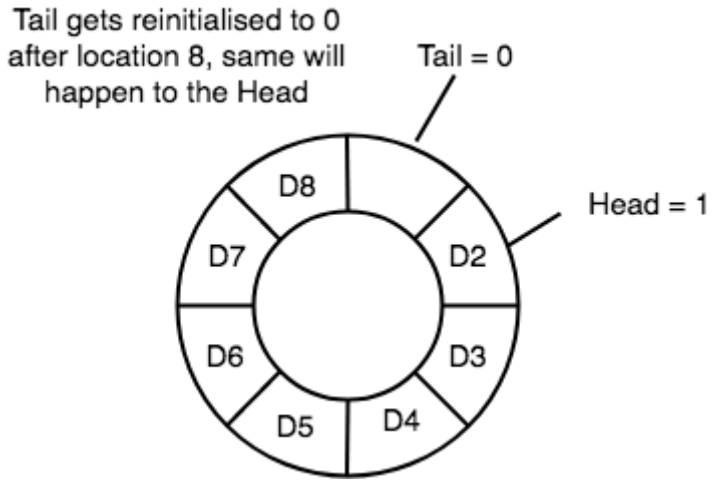
- New data is always added to the location pointed by the `tail` pointer, and once the data is added, `tail` pointer is incremented to point to the next available location.



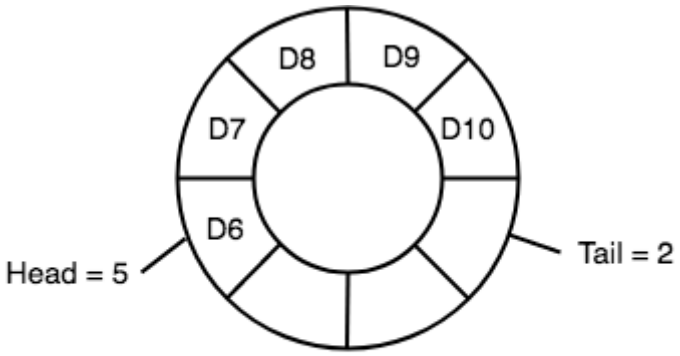
- In a circular queue, data is not actually removed from the queue. Only the `head` pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between `head` and `tail`, hence the data left outside is not a part of the queue anymore, hence removed.



- The `head` and the `tail` pointer will get reinitialised to **0** every time they reach the end of the queue.



- Also, the `head` and the `tail` pointers can cross each other. In other words, `head` pointer can be greater than the `tail`. Sounds odd? This will happen when we dequeue the queue a couple of times and the `tail` pointer gets reinitialised upon reaching the end of the queue.



In such a situation the value of the Head pointer will be greater than the Tail pointer

Implementation:

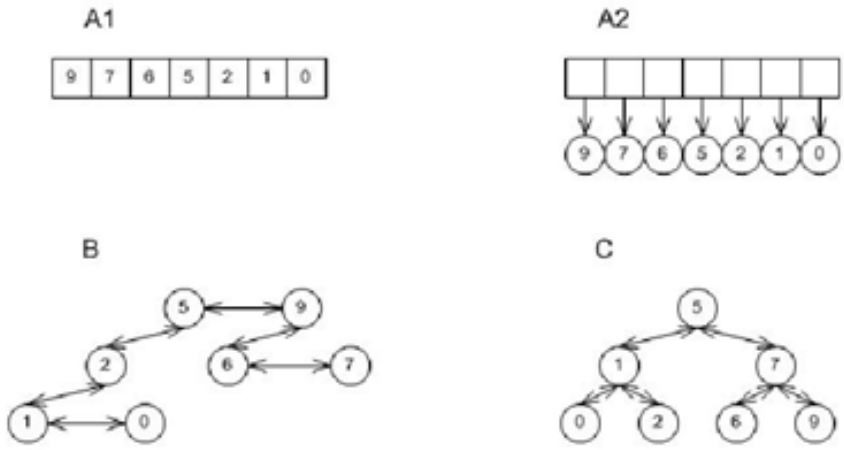
Below we have the implementation of a circular queue:

5. Initialize the queue, with size of the queue defined (maxSize), and head and tail pointers.
6. enqueue: Check if the number of elements is equal to maxSize - 1:
    - If **Yes**, then return **Queue is full**.
    - If **No**, then add the new data element to the location of tail pointer and increment the tail pointer.
7. dequeue: Check if the number of elements in the queue is zero:
    - If **Yes**, then return **Queue is empty**.
    - If **No**, then increment the head pointer.
8. Finding the size:
    - If, **tail >= head**, size = (tail - head) + 1
    - But if, **head > tail**, then size = maxSize - (head - tail) + 1

PRIORITY QUEUES

Priority queues *are a kind of queue in which the elements are dequeued in priority order.*

A priority queue is a collection of elements where each element has an associated priority. Elements are added and removed from the list in a manner such that the element with the highest (or lowest) priority is always the next to be removed. When using a heap mechanism, a priority queue is quite efficient for this type of operation.



- Each element has a priority, an element of a totally ordered set (usually a number).
- More important things come out first, even if they were added later.
- Three types of priority. Low priority [10], Normal Priority [5] and High Priority [1].
- There is no (fast) operation to find out whether an arbitrary element is in the queue.

ALGORITHM:

Priority Queue - Algorithms - Adjust

```
Adjust(i)
left = 2i, right = 2i + 1

if left <= H.size and H[left] > H[i]
    then largest = left
    else largest = i
if right <= H.size and H[right] > H[largest]
    then largest = right
if largest != i
    then swap H[largest] with H[i]
        Adjust(largest)
```

Explanation:

Adjust works recursively to guarantee the heap property. It compares the current node with its children finding which, if either, has a greater priority. If one of them does, it will swap array locations with the largest child. Adjust is then run again on the current node in its new location.

Priority Queue - Algorithms - Insert

```
Insert(Key)
        H.size = H.size + 1
        i = H.size
        while i > 1 and H[i/2] < Key
                H[i] = H[i/2]
                i = i/2
        end while
        H[i] = key
```

Explanation

The insert algorithm works by first inserting the new element (key) at the end of the array. This element is then compared with its parent for highest priority. If the parent has a lower priority, the two elements are swapped. This process is repeated until the new element has found its place.

Priority Queue - Algorithms - Extract

```
Extract()
    Max = H[1] H[1] = H[H.size]
```

H.size = H.size – 1

Adjust(1) Return

Max

Explanation

Extract works by removing and returning the first array element, the one of highest priority, and then promoting the last array element to the first. Adjust is then run on the first element so that the heap property is maintained.

Run Time Complexity

Θ(lg *n*) time for Insert and worst case for Extract (where *n* is number of elements)

Θ(lg *n*) average time for Insert

Can construct a Heap from a list in Θ(lg *n*) where a Binary Search Tree takes Θ(*n* lg *n*)

Space Requirements

All operations are done in place - space requirement at any given time is the number of elements, *n*.

5.Explain simple queue and double ended queue in detail.

Simple Queue:

In a simple queue, insertion takes palce at the rear and removal occurs at the front. It strictly follows FIFO rule.

Queue Specifications

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations.

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty.
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

Double Ended Queue

A deque *(short for double-ended queue) is an abstract data structure for which elements can be added to or removed from the front or back(both end). This differs from a normal queue, where*

elements can only be added to one end and removed from the other. Both queues and stacks can be considered specializations of deques, and can be implemented using deques.

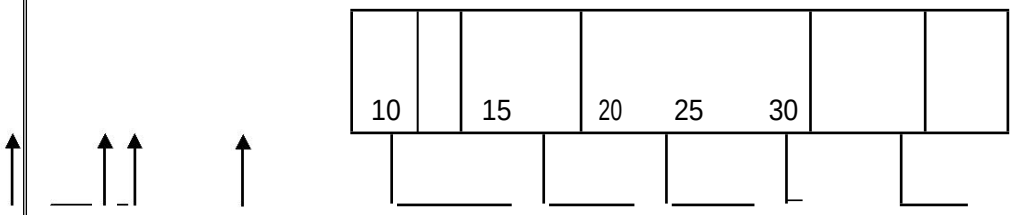
INSERT NEW ELEMENT INTO DEQUEUE:

AT THE END OF THE DEQUEUE:

- *First check the dequeue is full or not.*
- *We have to check whether the element is first to be added or inserted if it is true assign front and rear is 0.*
- *Insert a new element*

AT THE BEGINNING OF THE DEQUEUE:
First check the dequeue is full or not.

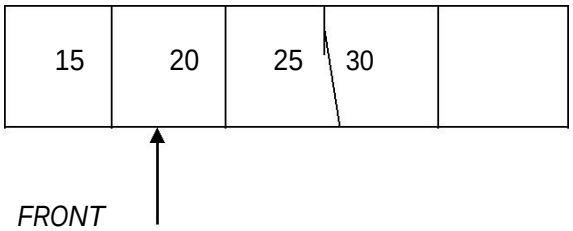
- *We have to check whether the element is first to be inserted if it is true then perform the following steps*
  1. *shift the elements from left to right*
  2. *so that we need the number of elements present in the dequeue ,the position of the last element ie the position of the rear*
  3. *now insert any element in zeroth position*



DELETING THE ELEMENT FROM DEQUEUE:

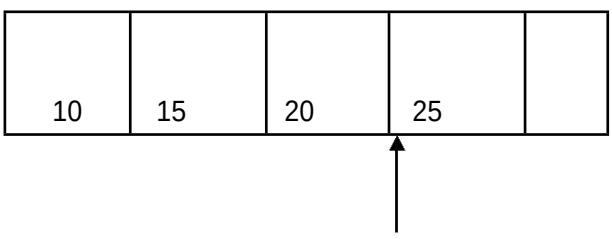
AT THE BEGINNING OF DEQUEUE:

- *delete the first element from the front position of the dequeue*
- *the index of next elem*
- *.ent is stored in front pointer.*
- Increment the front pointer by one .



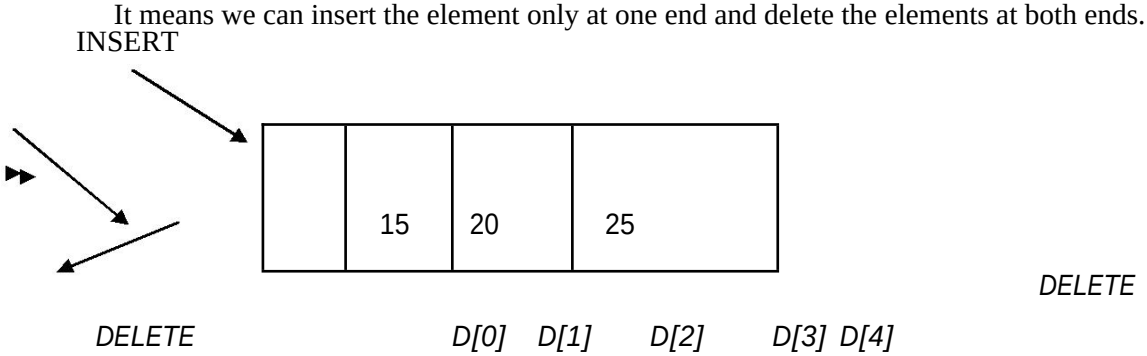
AT THE END OF DEQUEUE:

- The last element is deleted .
- The index of next element is stored in rear pointer
- Decrement the rear pointer by one.



INPUT RESTRICTED DEQUEUE:

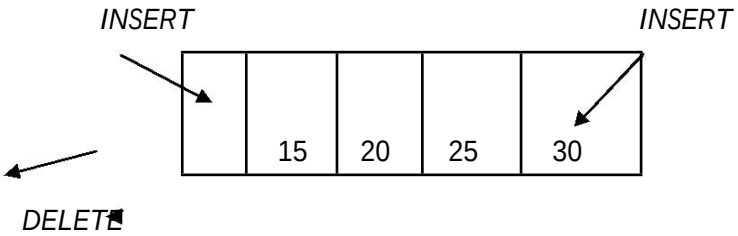
It means we can insert the element only at one end and delete the elements at both ends.



➢ *The above diagram shows the input restricted dequeue.*

OUTPUT RESTRICTED DEQUEUE:

It means we can insert the elements in both ends and delete the elements at only one end.



The above diagram shows the output restricted dequeue.

6. What is stack? Write its time complexities along with its application in detail.

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

The Stack is Last In First Out (LIFO) data structure. This data structure has some important applications in different aspect. These are like below −

- Expression Handling −
    - Infix to Postfix or Infix to Prefix Conversion −

      The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions. These expressions are not so much familiar to the infix expression, but they have some great advantages also. We do not need to maintain operator ordering, and parenthesis.

    - Postfix or Prefix Evaluation −

After converting into prefix or postfix notations, we have to evaluate the expression to get the result. For that purpose, also we need the help of stack data structure.

- Backtracking Procedure −

  Backtracking is one of the algorithm designing technique. For that purpose, we dive into some way, if that way is not efficient, we come back to the previous state and go into some other paths. To get back from current state, we need to store the previous state. For that purpose, we need stack. Some examples of backtracking is finding the solution for Knight Tour problem or N-Queen Problem etc.

- Another great use of stack is during the function call and return process. When we call a function from one other function, that function call statement may not be the first statement. After calling the function, we also have to come back from the function area to the place, where we have left our control. So we want to resume our task, not restart. For that reason, we store the address of the program counter into the stack, then go to the function body to execute it. After completion of the execution, it pops out the address from stack and assign it into the program counter to resume the task again.

SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution & Accredited by NAAC with "A" Grade)
(An Autonomous Institution)
Madagadipet, Puducherry - 605 107

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject Name: **Data Structures**
Subject Code:

Prepared by:

Verified by: **Approved by:**

UNIT – III

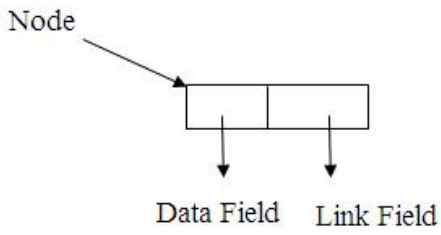
LINKED LIST OPERATIONS

Linked Lists: Singly linked lists: Representation in memory, Algorithms of several operations: Traversing, Searching, Insertion, Deletion in linked list; Linked representation of Stack and Queue. Doubly linked list: Operations. Circular Linked Lists: operations.

2 MARKS

1.What is Linked list? (Nov 11, Nov 13, Apr 15)

- Linked list is a dynamic and linear data structure.
- Linked list is an ordered collection of elements in which each element is referred as a node.
- Each node has two fields namely
  i. Data field or Information field and
  ii. Address field or Link field.



2.What are various fields in a Linked list?

Each node has two fields namely
i. Data field or Information field and
ii. Address field or Link field.

Data Field: Data field contains the actual data.

Address Field: Address field contains the address of another node.

3.Head Pointer?

- Head pointer is the pointer which always points the first node in the list.
- Head pointer holds the address of the first node.
- Using Head pointer only we can move from first node to last node.

4.Define External address, internal address and Null address?

External Address: External address is an address stored in head pointer which holds the address of the first node.

Internal Address: Internal address is an address stored in link field of the each node except the last node. **Null Address:** Null address is an address stored in link field of the last node indicates the end of the list.

5.What are the types of Linked List? (Apr 15)

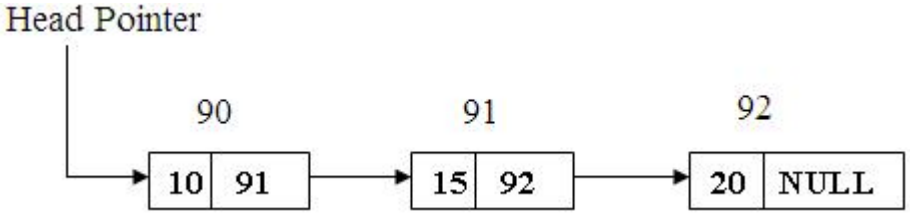
There are three types of Linked list. They are

i. Single Linked list,

ii. Double Linked list, and Circular Linked list.

6.What is Single linked list?

- In Single linked list, each node has one link to the next node.
- In Single linked list, we can move from only one direction from head pointer to Null pointer.
- Each node has two fields namely
  i. Data field or Information field and
  ii. Address field or Link field.
- It is otherwise called as Linear Linked list.
- Consider the following single linked list,



7.What is Double Linked list? (Apr 12, Apr 13) (NOV 15)

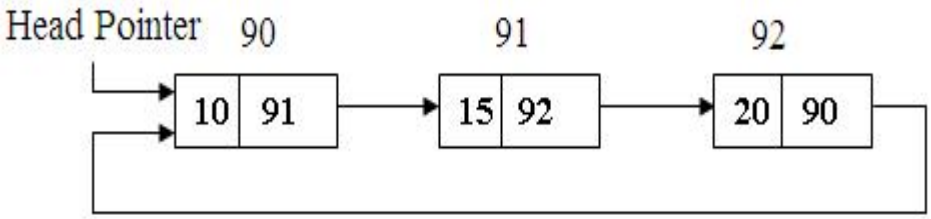
- Each node consist of three fields namely
  i. Previous address field or Backward link field,
  ii. Data field or Information field,
  iii. External address field or Forward link field.
- The previous address field holds the address of the previous node and the next address field holds the address of next node.
- In Double Linked list, we can move in both the direction from head pointer to Null address or vice versa.
- Consider the following linked list,

8.What is Circular Linked list? Mention its types.

- In Circular linked list, the last node is connected to the first node.
- In Circular Linked list, we can move from head pointer to Null address.
- There are two types of Circular linked list. They are
    i. Circular Single Linked list and
    ii. Circular Double Linked list.

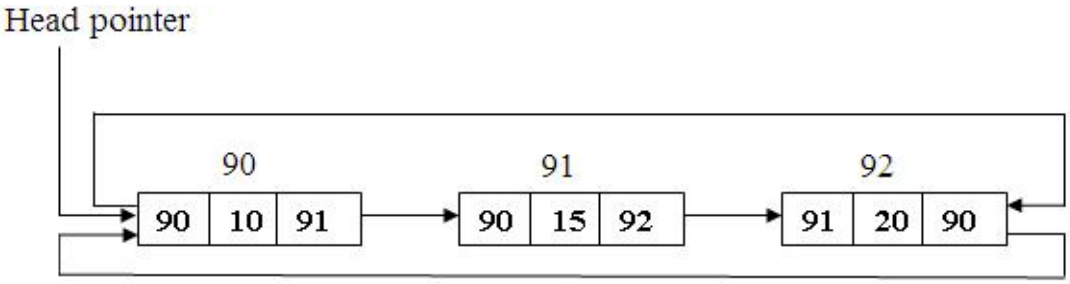
9.What is Circular Single Linked list?

- In Circular Single linked list, the last node is connected to the first node.
- In Circular Single linked list, we can move from only one direction from head pointer to Null pointer
- Consider the following circular single linked list,



10.What is Circular Double Linked list?

- In Circular Double linked list, the last node is connected to the first node.
- In Circular Double Linked list, we can move in both the direction from head pointer to Null address or vice versa. Consider the following circular single linked list,



11.In how many ways we can insert a node into a single linked list?

We can insert a new node into the list by following positions

i. Inserting a node as a first node,
ii. Inserting a node as a last node and
iii. Inserting a node as an intermediate node.

12. How can we insert a new node as a first node?

- If we want to insert a new node into a single linked list, first we need the empty node with empty data field and empty address field.
- Now store the new element into data field of the new node.
- To insert as a first node, the link field of the new node should replaced by the head pointer.
- Head pointer should replaced by the address of new node.

13.How can we insert a new node as a last node?

If we want to insert a new node into a single linked list, first we need the empty node with empty data field and empty address field.

- Now store the new element into data field of the new node.
- To insert as a last node, the link field of the last node should replaced by the address of new node.
- Link field of the new node should replaced by NULL.

14.How can we insert a new node as an intermediate node?

- If we want to insert a new node into a single linked list, first we need the empty node with empty data field and empty address field.
- Now store the new element into data field of the new node.
- To insert as an intermediate node, first we should know the address of the previous node. Then replace the link field of the previous node by the address of new node.
- Replace the link field of new node by the address of successor.

15.How does a stack-linked list differ from a linked list?

A stack linked list refers to a stack implemented using a linked list. That is to say, a linked list in which you can only add or remove elements to or from the top of the list. A stack-linked list accesses data last in, first out; a linked list accesses data first in, first out.

16.How can you search for data in a linked list?

The only way to search a linked list is with a linear search, because the only way a linked list's members can be accessed is sequentially. Sometimes it is quicker to take the data from a linked list and store it in a different data structure so that searches can be more efficient.

17.List the basic operations carried out in a linked list.

The basic operations carried out in a linked list include.

- Creation of list.
- Insertion of a node.
- Deletion of a node.
- Modification of a node.
- Traversal of the list.

18.List the operations other than the basic operations that carried out in a linked list.

The operations other than the basic operations that carried out in a linked list includes

- Searching an element in a list.
- Finding the predecessor element of a node.

- Finding the successor element of a node.
- Appending a linked list to another existing list.
- Splitting a linked list in to two lists.
- Arranging a linked list in ascending or descending order.

19.List out the advantages in using a linked list. (Nov 14)

The advantages in using a linked list are

- It is not necessary to specify the number if elements in a linked list during its declaration
- Linked list can grow and shrink in size del=pending upon the insertion and deletion that occurs in the list.
- Insertions and deletions at any place in a list can be handled easily and efficiently
- A linked list does not waste any memory space.

20.List out the disadvantages in using a linked list.

The advantages in using a linked list

- Searching a particular element in list is difficult and time consuming
- A linked list will use more storage space than an array to store the same number of elements.

21.List out the application of a linked list

Some of the important applications of linked lists are

- Manipulation of polynomials,
- Stacks and
- Queues.

22.State the difference between arrays and linked list.

Arrays	Linked List
Size of any arrays is fixed.	Size of a list is variable.
It is necessary to specify the number of elements during the declaration.	It is not necessary to specify the number of elements during the declaration.
It occupies less memory space than linked list for the same memory number of elements.	It occupies more memory space.

23.How will you search an element in a linked list by iterative approach?

Searching an element in a given linked list will happen sequentially starting from the head node of the list until the element has been found.

1) Initialize a node pointer, temp = head

2) Do following while temp is not NULL

- temp->data is equal to the data being searched, return true.
- temp = temp->next

3) Return false

24.How will you search an element in a linked list by iterative approach?

Searching an element in a given linked list will happen sequentially starting from the head node of the list until the element has been found.

1) If head is NULL, return false.

2) If head's data is same as the data to be searched, return true;

3) Else return search(head->next, search_data)

25.How will you insert an node in a linked list?

struct Node

{

int data;

struct Node *next;

};

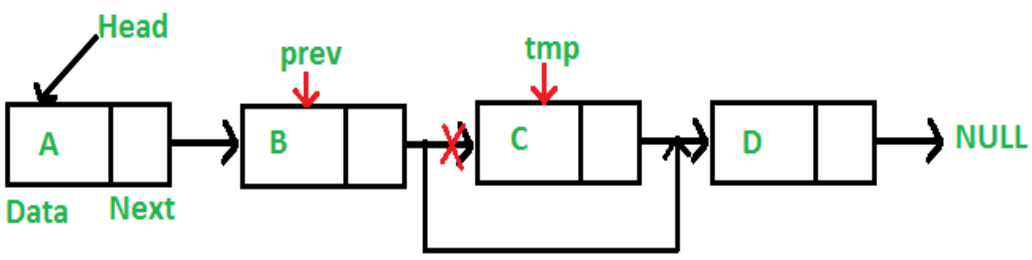
A node can be added in three ways

- At the front of the linked list
- After a given node.
- At the end of the linked list.

26.How will you insert an node in a linked list?

To delete a node from linked list, we need to do following steps.

1) Find previous node of the node to be deleted.
2) Change the next of previous node.
3) Free memory for the node to be deleted.



27.Mention the advantages of representing stacks using linked list than arrays.

The advantages of representing stacks using linked list than arrays are

- It is not necessary to specify the number of elements to be stored in a stack during its declaration.
- Insertions and deletions can be handled easily and efficiently.
- Linked list representation of stacks can grow & shrink in size without wasting the memory space, depending upon the insertion and deletion that occurs in the list.
- Multiple stacks can be represented efficiently using a chain for each stack.

28.Mention the representation of DLL.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

29.What are the operations performed in DLL?

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner

30. Advantages of circular linked list.

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

5 Marks

1. What is Linked List? What are its types?

Some demerits of array, leads us to use linked list to store the list of items. They are,

a. It is relatively expensive to insert and delete elements in an array.

b. Array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. *(For this reason, arrays are called **"dense lists"** and are said to be **"static"** data structures.)*

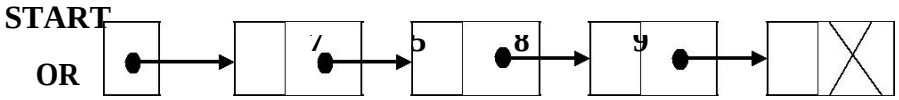
A ***linked list***, or ***one-way list***, is a linear collection of data elements, called ***nodes***, where the linear order is given by means of ***pointers***. That is, each node is divided into two parts:

- The first part contains the information of the element i.e. INFO or DATA.
- The second part contains the ***link field***, which contains the address of the next node in the list.
- The linked list consists of series of nodes, which are not necessarily adjacent in memory.
- A list is a ***dynamic data structure*** i.e. the number of nodes on a list may vary dramatically as elements are inserted and removed.

The pointer of the last node contains a special value, called the ***null*** pointer, which is any invalid address. This ***null pointer*** signals the end of list.

The list with no nodes on it is called the ***empty list*** or ***null list***.

Example*: The linked list with 4 nodes.*



Types of Linked Lists:

a) Singly Linked List

b) Circular Linked List

c) Two-way or doubly linked lists

d) Circular doubly linked lists

2. Write an advantage of linked list over Array?

- An array is the data structure that contains a collection of similar type data elements whereas the Linked list is considered as non-primitive data structure contains a collection of unordered linked elements known as nodes.
- In the array the elements belong to indexes, i.e., if you want to get into the fourth element you have to write the variable name with its index or location within the square bracket.
- In a linked list though, you have to start from the head and work your way through until you get to the fourth element.
- Accessing an element in an array is fast, while Linked list takes linear time, so it is quite a bit slower.
- The requirement of memory is less due to actual data being stored within the index in the array. As against, there is a need for more memory in Linked Lists due to storage of additional next and previous referencing elements.

- Arrays are of fixed size. In contrast, Linked lists are dynamic and flexible and can expand and contract its size.

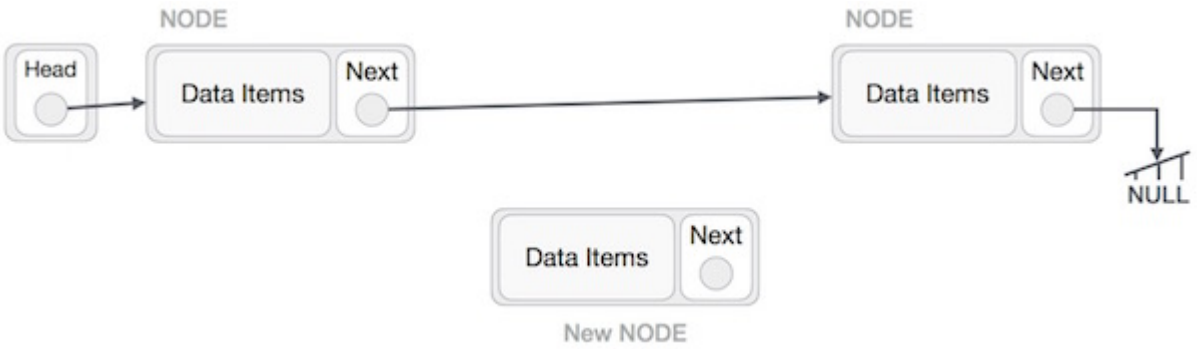
3. Explained any two operations of Linked List?

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

INSERTION OPERATION:

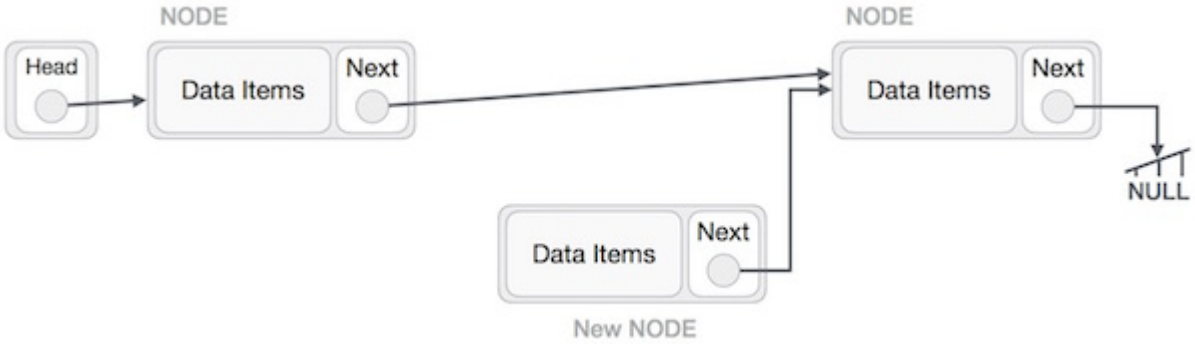
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

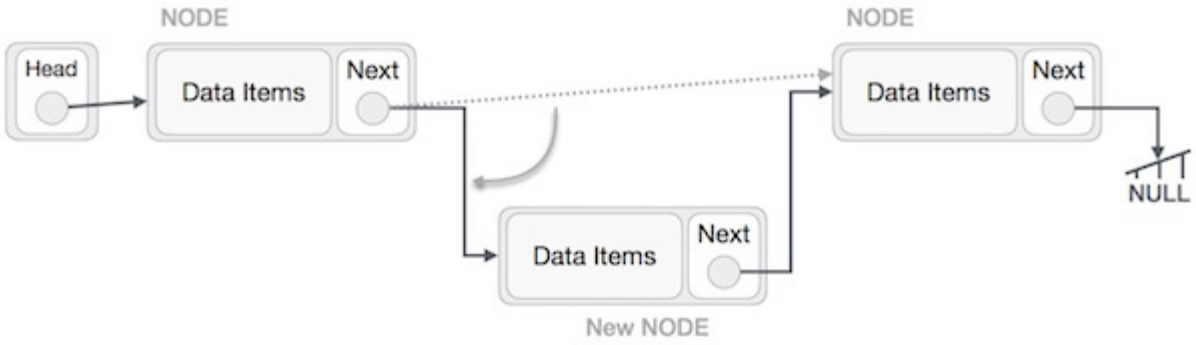
NewNode.next −> RightNode;

It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;

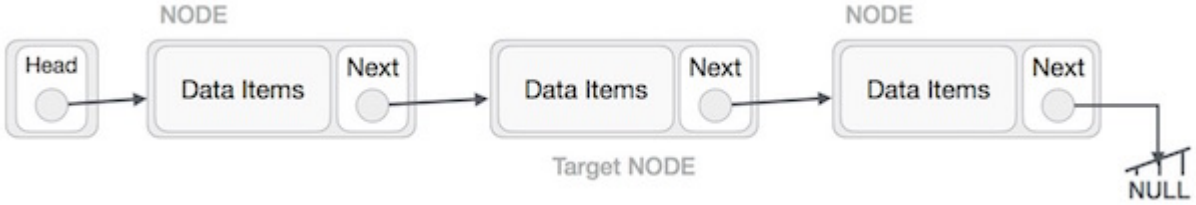


This will put the new node in the middle of the two. The new list should look like this −

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.'

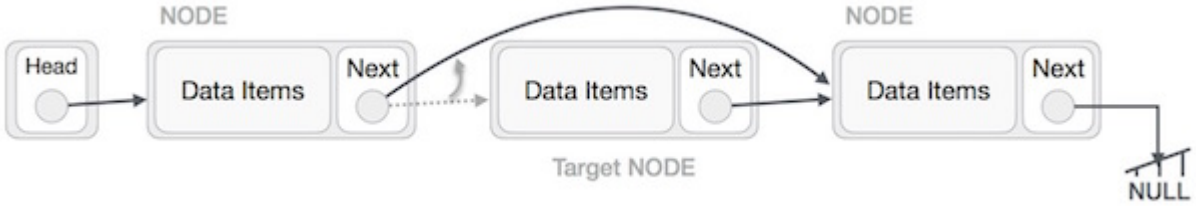
DELETE OPERATION:

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



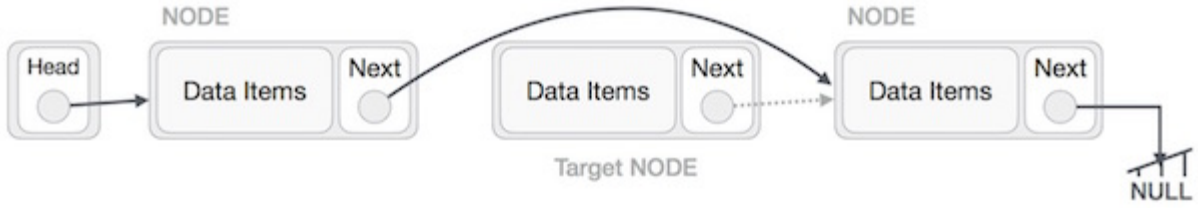
The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −> TargetNode.next;

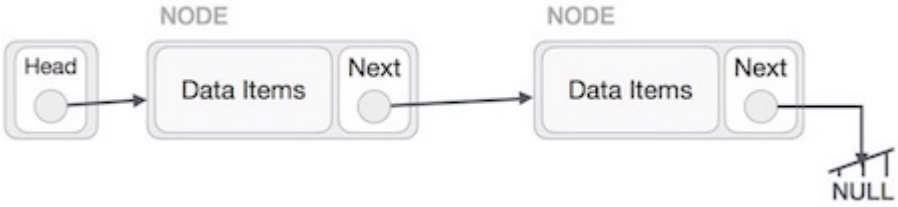


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;



We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

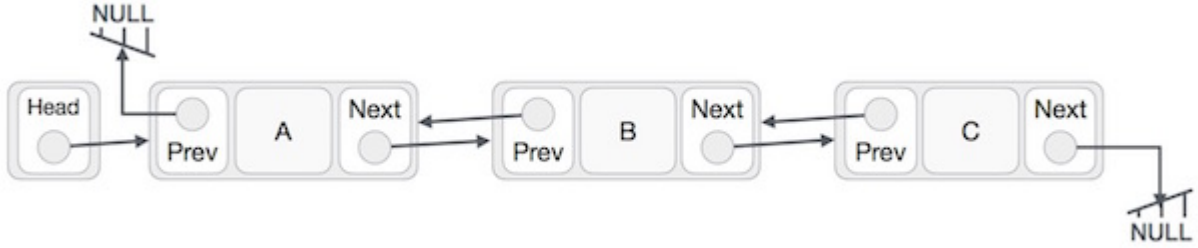


4. Explained Doubly Linked List?

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly 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 Next.
- **Prev** − Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



BASIC OPERATION:

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner

5. Explain insertion and deletion operations of doubly linked list?

INSERTION OPERATION:

```
//insert link at the first location
void insertFirst(int key, int data) {

  //create a link
  struct node *link = (struct node*) malloc(sizeof(struct node));
  link->key = key;
  link->data = data;

  if(isEmpty()) {
    //make it the last link
    last = link;
  } else {
    //update first prev link
    head->prev = link;
  }

  //point it to old first link
  link->next = head;

  //point first to new first link
  head = link;
}
```

DELETION OPERATION:

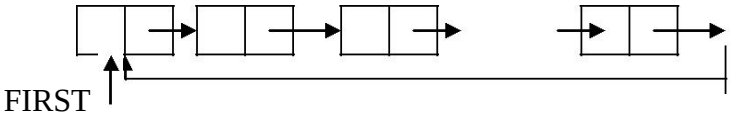
```
//delete first item
struct node* deleteFirst() {

  //save reference to first link
  struct node *tempLink = head;

  //if only one link
  if(head->next == NULL) {
    last = NULL;
  } else {
    head->next->prev = NULL;
  }

  head = head->next;

  //return the deleted link
  return tempLink;
}
```

6. Explain circular linked list?

In the singly linked linear list the last node consist of the NULL pointer. Slightly improvement in this type of linked list is accomplished by replacing the null pointer in the last node of a list with the address of its first node.such a list is called circularly linked linear list or simply a circular list

7.

STRUCTURE OF A CIRCULAR LIST:



ADVANTAGE OF CIRCULAR LIST OVER SINGLY LINKED LISTS*:*

1. Accessibility of a node from a given node, every node is easily accessible i.e., all node be reached by merely chaining through the list

2. Deletion operation: In addition to the address of x the node to be deleted from a singly list, it is also necessary to give address of the first node of the list in order to the predecessor of X.

Such a requirement does not exist for a circular list, since the search for the predecessor of node X can be initiated from X itself.

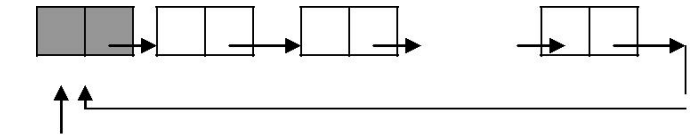
DISADVANTAGE OF A CIRCULAR LIST:

In processing a circular list, if we are not able to detect the end of the list, it is possible to get into an infinite loop.

This problem can be sowed i.e, the end of list can be detected by placing a special node which can be easily identified in the circular list .This special node is often called the list head of the circular list.

One more advantage of using this technique is that the list can never be empty which can be checked in the operation of singly linked list.

Representation of a circular list with a list head is shown below:



Here, variable HEAD denote the address of the list head .INFO field in the list head node is not used, which is shown by the shading the field.

An empty list is represented by having LINK(HEAD)=HEAD.

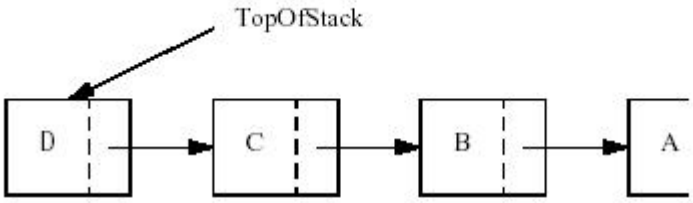
The algorithm for inserting a node at the head of a circular list with a list head consist of the following

steps:

NEW□NODE
INFO(NEW)□y
LINK(HEAD)□LINK(H
EAD)
LINK(HEAD)□NEW

7. Describe about Linked Stack.

The operation of adding an element to the front of a linked list is quite similar to that of pushing an element onto a stack. A stack can be accessed only through its top element, and a list can be accessed only from the pointer to its first element. Similar, the operation of removing the first element from a link list is analogous to popping a stack.



Linked list implementation of the stack

In both cases the only accessible item of collection is removed from that collection, and the next item becomes immediately accessible. The First node of the list is the top of the stack.

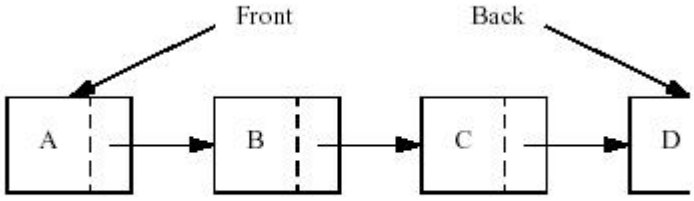
Available free memory spaces can be thought of as a finite pool of empty nodes existing initially. The most natural form from this pool to take that of a linked together by the next field in each node. This pool cannot be accessed by the programmer except through the memory allocation functions and free function. For eg: malloc function remove the first node from the list where as free return a mode to the front of the list. The list of available node is called the available list.

The advantage of the list implementation of the stack is that of all stacks being used by a program can share the same available list.

When any stack needs a node ,it can obtain it from the single variable list. When any stack no longer needs no node ,it returns the node to that same available list. As long as the total amount of spaces needed by all stack at any onetime is less than the amount of space initially available to them all ,each stack is available to grow and shrink to any size . No space has been preallocated to any single stack and stack is using the space that is does not need.

8. Describe about Linked Queue?

In queue, items are deleted from the front of a queue and inserted at the rear. Let a pointer to the first element of the list represent the front of the queue .Another pointer to the last element of the list represents the rear of the queue as shown in the following figure:



Linked list implementation of the queue

For this NodeQueue class, begin by declaring two instance variables itsFront and itsRear, each referring to a Node. In general, the queue's itsFront will always refer to the Node containing the first data value (if any) and the queue's itsRear will always refer to the Node containing the last data value (if any). An empty queue has null for itsFront, since there are no data values and so no Nodes at all.

9. Write an advantages, disadvantages and applications of Linked List?

Advantages:

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages:

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications:

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

10 Marks

Describe Doubly Linked List in detail?

A linked list is a sequence of data structures, which are connected together via links.

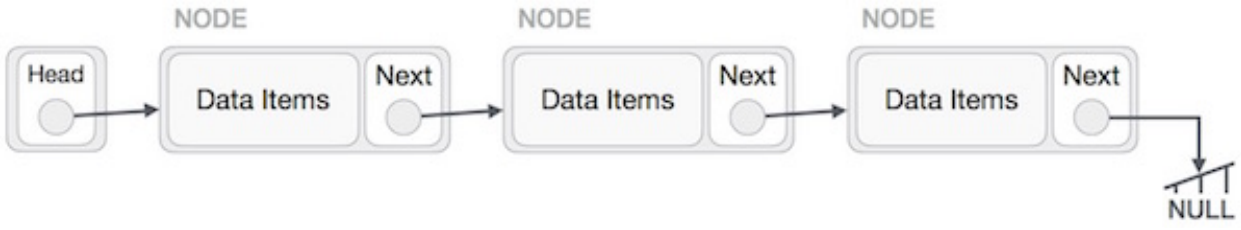
Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

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 Next.

Linked List – A Linked List contains the connection link to the first link called First

Linked List Representation



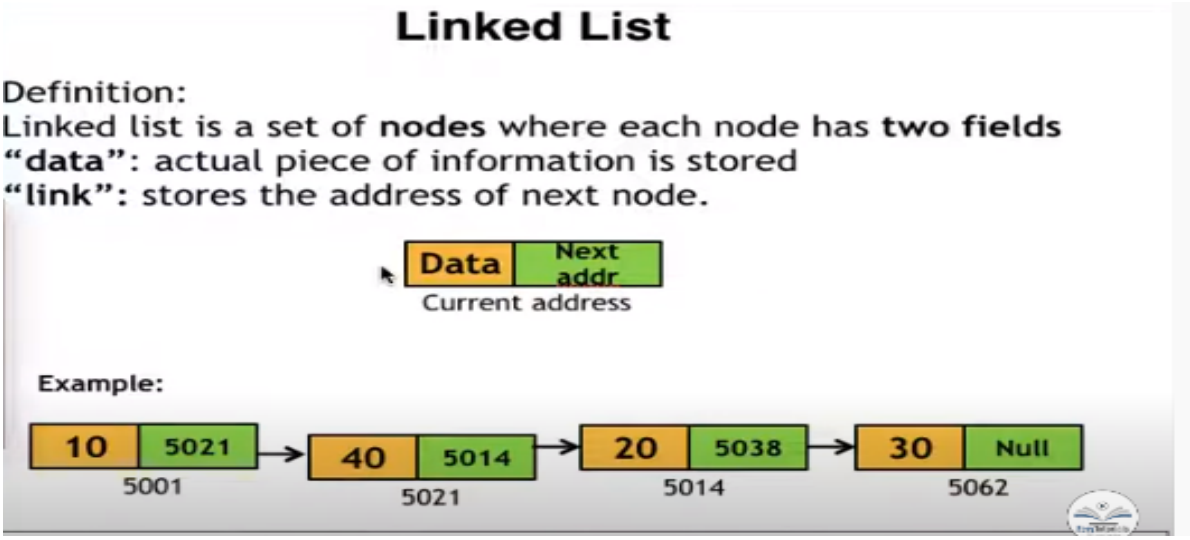
As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

What is 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 (tail).
- 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 access every other element in the list. The last node, also sometimes called the tail, points to *NULL* which helps us in determining when the list ends.



When to use Linked List?

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

Types of linked list

1. Singly Linear Linked list
2. Singly Circular linked list
3. Doubly Linear Linked list
4. Doubly circular linked list

Basic Operations

Following are the basic operations supported by a list.

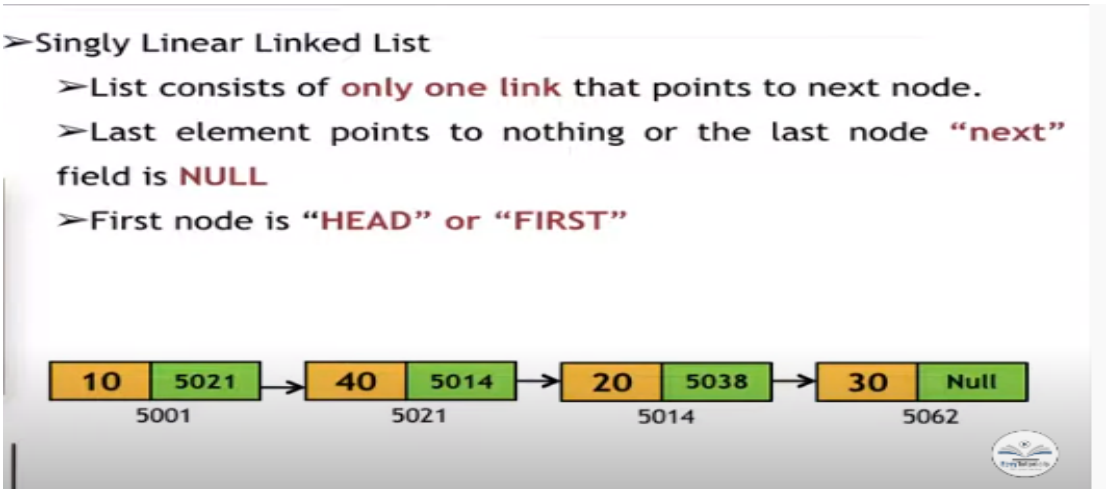
Insertion – Adds an element at the beginning of the list.

Deletion – Deletes an element at the beginning of the list.

Display – Displays the complete list.

Search – Searches an element using the given key.

Delete – Deletes an element using the given key.



Singly Linked List

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
   int data;
   struct node *next;
}*head=NULL,*p,*t;
int main()
{
   int ch;
   void insert_beg();
   void insert_end();
   int insert_pos();
```

```
void display();
void delete_beg();
void delete_end();
int delete_pos();
while(1)
{
    printf("\n\n---- Singly Linked List(SLL) Menu ----");
    printf("\n1.Insert\n2.Display\n3.Delete\n4.Exit\n\n");
    printf("Enter your choice(1-4):");
    scanf("%d",&ch);

    switch(ch)
    {
        case 1:
            printf("\n---- Insert Menu ----");
            printf("\n1.Insert at beginning\n2.Insert at end\n3.Insert at specified position\n4.Exit");
            printf("\n\nEnter your choice(1-4):");
            scanf("%d",&ch);

            switch(ch)
            {
                case 1: insert_beg();
                        break;
                case 2: insert_end();
                        break;
                case 3: insert_pos();
                        break;
                case 4: exit(0);
```

```
                default: printf("Wrong Choice!!");
            }
            break;
        case 2: display();
            break;
        case 3: printf("\n---- Delete Menu ----");
        printf("\n1.Delete from beginning\n2.Delete from end\n3.Delete from specified
       position\n4.Exit");
            printf("\n\nEnter your choice(1-4):");
            scanf("%d",&ch);
            switch(ch)
            {
                case 1: delete_beg();
                        break;
                case 2: delete_end();
                        break;
                case 3: delete_pos();
                        break;
                case 4: exit(0);
                default: printf("Wrong Choice!!");
            }
            break;
        case 4: exit(0);
            default: printf("Wrong Choice!!");
      }
   }
   return 0;
}
```

```
void insert_beg()
{
    int num;
    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    t->data=num;


    if(head==NULL)        //If list is empty
    {
        t->next=NULL;
        head=t;
    }
    else
    {
        t->next=head;
        head=t;
    }
}

void insert_end()
{
    int num;
    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
```

```
    t->data=num;
    t->next=NULL;

    if(head==NULL)        //If list is empty
    {
        head=t;
    }
    else
    {
        p=head;
        while(p->next!=NULL)
        p=p->next;
        p->next=t;
    }
}

int insert_pos()
{
    int pos,i,num;
    if(head==NULL)
    {
        printf("List is empty!!");
        return 0;
    }

    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
```

```
    printf("Enter position to insert:");
    scanf("%d",&pos);
    t->data=num;

    p=head;
    for(i=1;i<pos-1;i++)
    {
        if(p->next==NULL)
        {
            printf("There are less elements!!");
            return 0;
        }

        p=p->next;
    }

    t->next=p->next;
    p->next=t;
    return 0;
}

void display()
{
    if(head==NULL)
    {
        printf("List is empty!!");
    }
    else
```

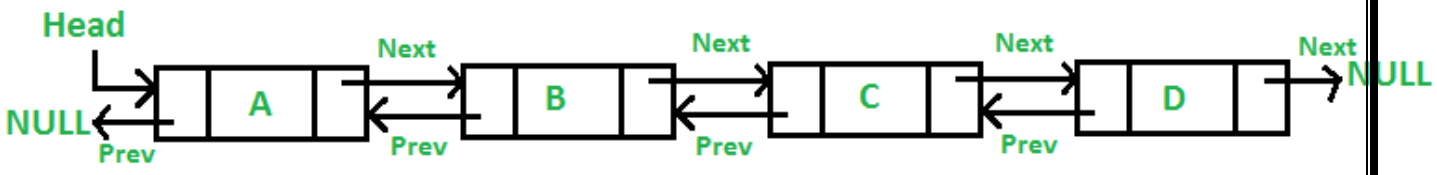
```
    {
        p=head;
        printf("The linked list is:\n");
        while(p!=NULL)
        {
            printf("%d->",p->data);
            p=p->next;
        }
    }
}

void delete_beg()
{
    if(head==NULL)
    {
        printf("The list is empty!!");
    }
    else
    {
        p=head;
        head=head->next;
        printf("Deleted element is %d",p->data);
        free(p);
    }
}

void delete_end()
{
```

```
    if(head==NULL)
    {
        printf("The list is empty!!");
    }
    else
    {
        p=head;
        while(p->next->next!=NULL)
        p=p->next;

        t=p->next;
        p->next=NULL;
        printf("Deleted element is %d",t->data);
        free(t);
    }
}

int delete_pos()
{
    int pos,i;

    if(head==NULL)
    {
        printf("List is empty!!");
        return 0;
    }

    printf("Enter position to delete:");
```

```
    scanf("%d",&pos);


    p=head;
    for(i=1;i<pos-1;i++)
    {
       if(p->next==NULL)
       {
          printf("There are less elements!!");
          return 0;
       }
       p=p->next;
    }

    t=p->next;
    p->next=t->next;
    printf("Deleted element is %d",t->data);
    free(t);

    return 0;
}
```

2. Describe Doubly Linked List in detail?

A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Following is representation of a DLL node in C language:

```
/* Node of a doubly linked list */

struct Node {

    int data;

    struct Node* next; // Pointer to next node in DLL

    struct Node* prev; // Pointer to previous node in DLL

};
```

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3) We can quickly insert a new node before a given node.
In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list

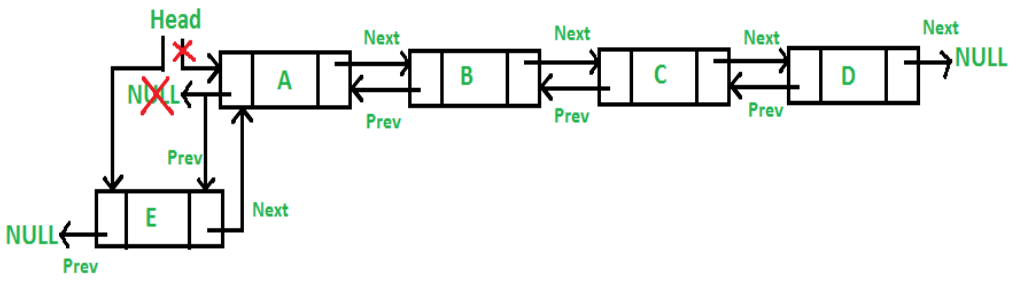
1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though (See this and this).
2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

Insertion

A node can be added in four ways
1) At the front of the DLL
2) After a given node.
3) At the end of the DLL
4) Before a given node.

Add a node at the front: (A 5 steps process)

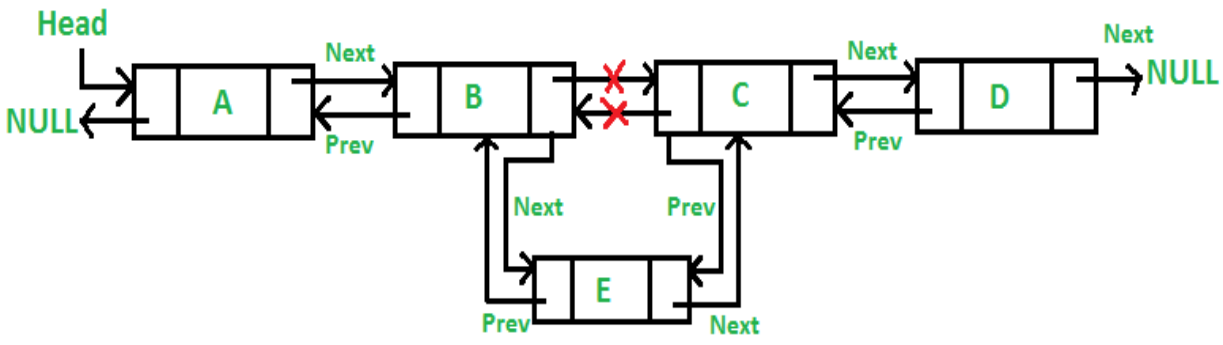
The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example if the given Linked List is 10152025 and we add an item 5 at the front, then the Linked List becomes 510152025. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node



```
/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    /* 2. put in the data */
    new_node->data = new_data;
    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

Add a node after a given node.: (A 7 steps process)

We are given pointer to a node as prev_node, and the new node is inserted after the given node.



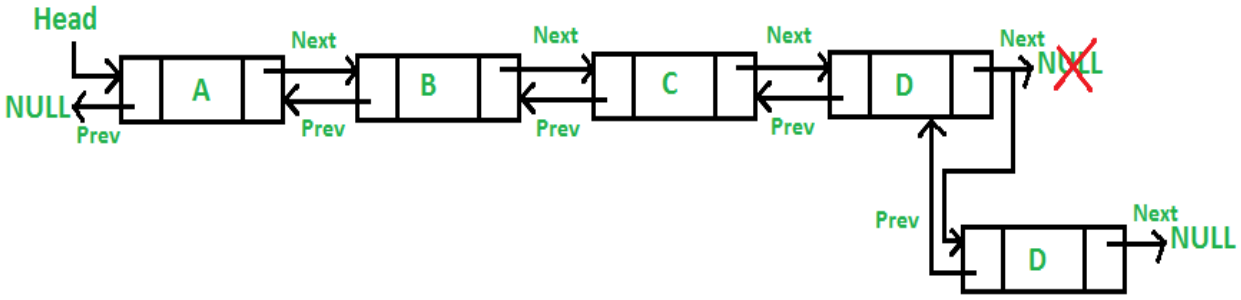

```
/* Given a node as prev_node, insert a new node after the given node */
```

```
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL) {
            printf("the given previous node cannot be NULL");
            return;
    }

    /* 2. allocate new node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. Make the next of prev_node as new_node */
    prev_node->next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node->prev = prev_node;

    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)
            new_node->next->prev = new_node;
}
```

Add a node at the end: (7 steps process)

The new node is always added after the last node of the given Linked List. For example if the given DLL is 510152025 and we add an item 30 at the end, then the DLL becomes 51015202530.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node



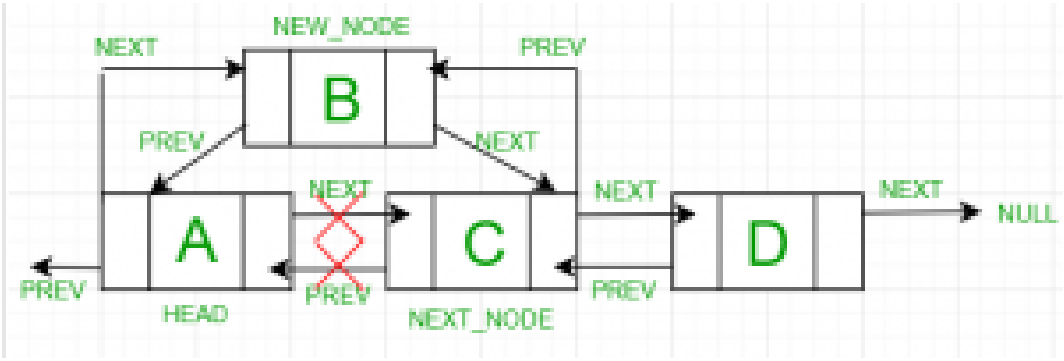

```
/* Given a reference (pointer to pointer) to the head

of a DLL and an int, appends a new node at the end */

void append(struct Node** head_ref, int new_data)

{

    /* 1. allocate node */

    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));


    struct Node* last = *head_ref; /* used in step 5*/


    /* 2. put in the data */

    new_node->data = new_data;


    /* 3. This new node is going to be the last node, so

          make next of it as NULL*/

    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new

          node as head */

    if (*head_ref == NULL) {

          new_node->prev = NULL;
```

```
            *head_ref = new_node;
            return;
        }

        /* 5. Else traverse till the last node */
        while (last->next != NULL)
            last = last->next;
        /* 6. Change the next of last node */
        last->next = new_node;


        /* 7. Make last node as previous of new node */
        new_node->prev = last;
        return;
    }
```

Add a node before a given node:

Steps
Let the pointer to this given node be next_node and the data of the new node to be added as new_data.

1. Check if the next_node is NULL or not. If it's NULL, return from the function because any new node can not be added before a NULL
2. Allocate memory for the new node, let it be called new_node
3. Set new_node->data = new_data
4. Set the previous pointer of this new_node as the previous node of the next_node, new_node->prev = next_node->prev
5. Set the previous pointer of the next_node as the new_node, next_node->prev = new_node
6. Set the next pointer of this new_node as the next_node, new_node->next = next_node;
7. If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node, new_node->prev->next = new_node
8. Else, if the prev of new_node is NULL, it will be the new head node. So, make (*head_ref) = new_node




```
// A complete working C program to demonstrate all
// insertion before a given node
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    new_node->data = new_data;

    new_node->next = (*head_ref);
    new_node->prev = NULL;
```

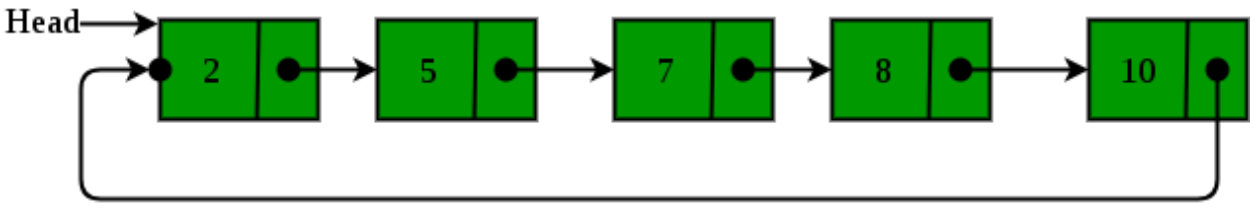
```
    if ((*head_ref) != NULL)
            (*head_ref)->prev = new_node;

    (*head_ref) = new_node;
}

/* Given a node as next_node, insert a new node before the given node */
void insertBefore(struct Node** head_ref, struct Node* next_node, int new_data)
{
    /*1. check if the given next_node is NULL */
    if (next_node == NULL) {
            printf("the given next node cannot be NULL");
            return;
    }

    /* 2. allocate new node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make prev of new node as prev of next_node */
    new_node->prev = next_node->prev;

    /* 5. Make the prev of next_node as new_node */
    next_node->prev = new_node;

    /* 6. Make next_node as next of new_node */
```

```
    new_node->next = next_node;

    /* 7. Change next of new_node's previous node */
    if (new_node->prev != NULL)
        new_node->prev->next = new_node;
    /* 8. If the prev of new_node is NULL, it will be
    the new head node */
    else
        (*head_ref) = new_node;
}

// This function prints contents of linked list starting from the given node
void printList(struct Node* node)
{
    struct Node* last;
    printf("\nTraversal in forward direction \n");
    while (node != NULL) {
        printf(" %d ", node->data);
        last = node;
        node = node->next;
    }

    printf("\nTraversal in reverse direction \n");
    while (last != NULL) {
        printf(" %d ", last->data);
        last = last->prev;
    }
}
```

```
/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    push(&head, 7);
    push(&head, 1);
    push(&head, 4);



    // Insert 8, before 1. So linked list becomes 4->8->1->7->NULL
    insertBefore(&head, head->next, 8);
    printf("Created DLL is: ");
    printList(head);
    getchar();
    return 0;
}
```

3. Describe about Circular Linked List in detail.

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

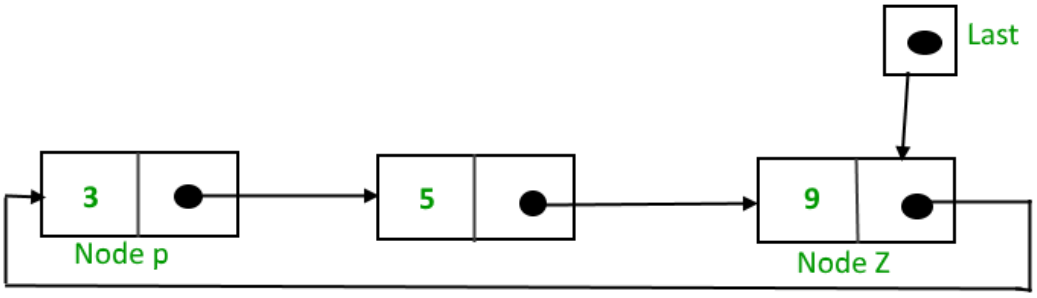
2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4) Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap

Implementation

To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last -> next will point to the first node.



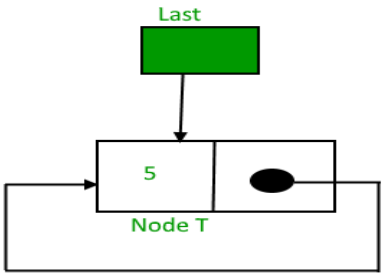
Insertion

A node can be added in three ways:

- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion in an empty List:

Initially when the list is empty, last pointer will be NULL.



After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.
Function to insert node in an empty List,

```
struct Node *addToEmpty(struct Node *last, int data)

{

    // This function is only for empty list

    if (last != NULL)

    return last;


    // Creating a node dynamically.

    struct Node *last =

            (struct Node*)malloc(sizeof(struct Node));


    // Assigning the data.

    last -> data = data;


    // Note : list was empty. We link single node

    // to itself.

    last -> next = last;


    return last;

}
```

Insertion at the beginning of the list
To Insert a node at the beginning of the list, follow these step:
1. Create a node, say T.
2. Make T -> next = last -> next.
3. last -> next = T.

Function to insert node in the beginning of the List,

```
struct Node *addBegin(struct Node *last, int data)

{
```

```
if (last == NULL)
    return addToEmpty(last, data);

// Creating a node dynamically.
struct Node *temp
        = (struct Node *)malloc(sizeof(struct Node));

// Assigning the data.
temp -> data = data;

// Adjusting the links.
temp -> next = last -> next;
last -> next = temp;

return last;
}
```

Insertion at the end of the list

To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T.

Function to insert node in the end of the List,

```
struct Node *addEnd(struct Node *last, int data)
{
if (last == NULL)
    return addToEmpty(last, data);

// Creating a node dynamically.
struct Node *temp =
```

```
        (struct Node *)malloc(sizeof(struct Node));


// Assigning the data.

temp -> data = data;


// Adjusting the links.

temp -> next = last -> next;

last -> next = temp;

last = temp;


return last;

}
```

Insertion in between the nodes

To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

Function to insert node in the end of the List,

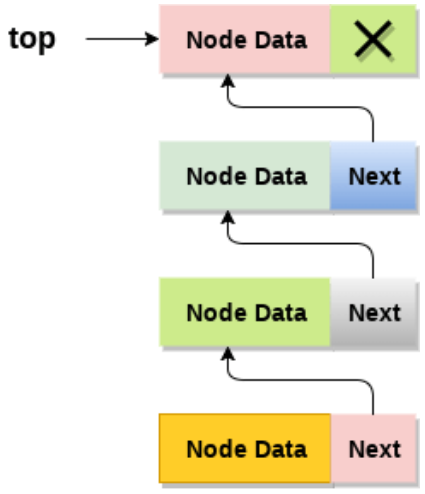
```
struct Node *addAfter(struct Node *last, int data, int item)

{

    if (last == NULL)

    return NULL;


    struct Node *temp, *p;

    p = last -> next;


    // Searching the item.

    do

    {
```

```
            if (p ->data == item)
            {
                    // Creating a node dynamically.
                    temp = (struct Node *)malloc(sizeof(struct Node));

                    // Assigning the data.
                    temp -> data = data;

                    // Adjusting the links.
                    temp -> next = p -> next;

                    // Adding newly allocated node after p.
                    p -> next = temp;
                    // Checking for the last node.
                    if (p == last)
                            last = temp;
                    return last;
            }
            p = p -> next;
    } while (p != last -> next);

    cout << item << " not present in the list." << endl;
    return last;
}
```

4. Describe about Linked Stack in detail.

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the 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.



Stack

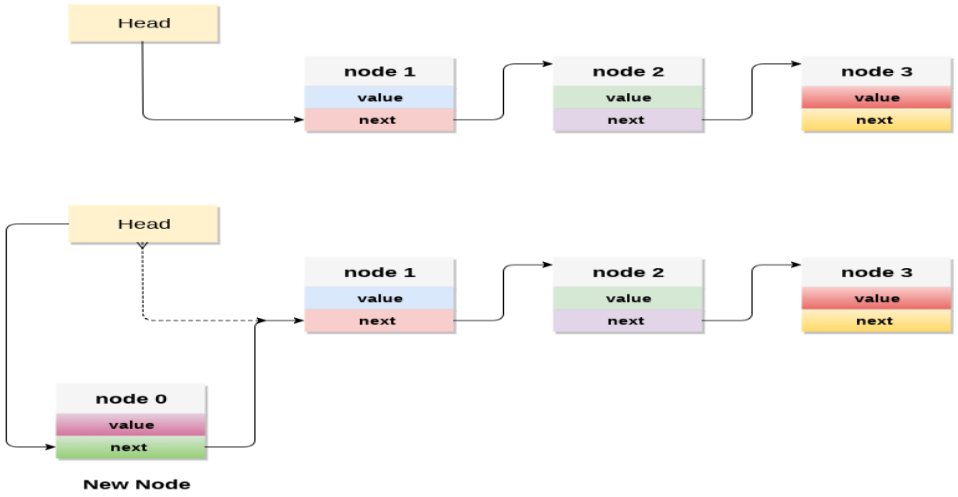
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the 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. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)




```
void push ()
{
   int val;
   struct node *ptr =(struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("not able to push the element");
   }
   else
   {
      printf("Enter the value");
      scanf("%d",&val);
      if(head==NULL)
      {
         ptr->val = val;
         ptr -> next = NULL;
         head=ptr;
      }
      else
      {
```

```
        ptr->val = val;
        ptr->next = head;
        head=ptr;

    }
    printf("Item pushed");
    }
}
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

- **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
- **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

```
void pop()
{
   int item;
   struct node *ptr;
   if (head == NULL)
   {
      printf("Underflow");
   }
   else
   {
      item = head->val;
      ptr = head;
```

```
        head = head->next;
        free(ptr);
        printf("Item popped");
    }
}
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

- Copy the head pointer into a temporary pointer.
- Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

```
void display()
{
   int i;
   struct node *ptr;
   ptr=head;
   if(ptr == NULL)
   {
      printf("Stack is empty\n");
   }
   else
   {
      printf("Printing Stack elements \n");
      while(ptr!=NULL)
      {
         printf("%d\n",ptr->val);
         ptr = ptr->next;
      }
   }
}
```

4. Describe about Linked Queue in details.

Due to the drawbacks of the array implementation of queue, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

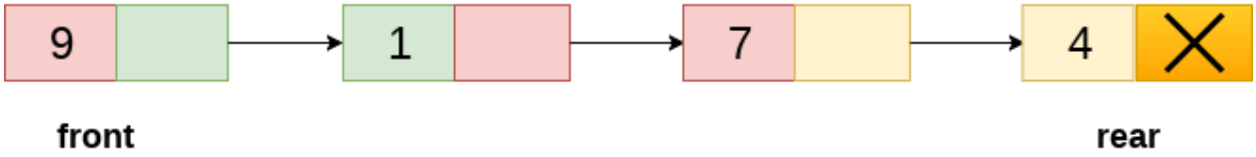
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

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

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

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

Firstly, allocate the memory for the new node ptr by using the following statement.

Ptr = (struct node *) malloc (sizeof(struct node));

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
    if(front == NULL)
    {
       front = ptr;
       rear = ptr;
       front -> next = NULL;
       rear -> next = NULL;
    }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

```
rear -> next = ptr;
       rear = ptr;
       rear->next = NULL;
```

Algorithm

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL
- **Step 3:** IF FRONT = NULL
  SET FRONT = REAR = PTR
  SET FRONT -> NEXT = REAR -> NEXT = NULL
  ELSE
  SET REAR -> NEXT = PTR
  SET REAR = PTR
  SET REAR -> NEXT = NULL
  [END OF IF]
- **Step 4:** END

```
void insert(struct node *ptr, int item; )
{
   ptr = (struct node *) malloc (sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW\n");
```

```
        return;
    }
    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 = NULL;
        }
    }
}
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
ptr = front;
        front = front -> next;
        free(ptr);
```

Algorithm

- **Step 1:** IF FRONT = NULL
  Write " Underflow "
  Go to Step 5
  [END OF IF]
- **Step 2:** SET PTR = FRONT
- **Step 3:** SET FRONT = FRONT -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** END

```
void delete (struct node *ptr)
{
   if(front == NULL)
   {
      printf("\nUNDERFLOW\n");
      return;
   }
   else
   {
      ptr = front;
      front = front -> next;
      free(ptr);
   }
}
```

5. Explain Single Linked List with its operations.

SINGLY / LINEAR LINKED LIST:

In a sequential representation, suppose that the items were implicitly ordered, that is, each item contained within itself the address of the next item, such an implicit ordering gives rise to a data structure known as a Linear linked list or Singly linked list which is shown in figure:

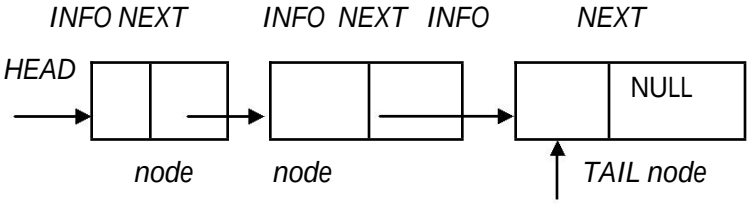
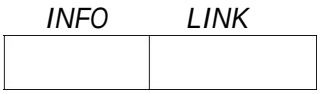


FIG:linear linked list.

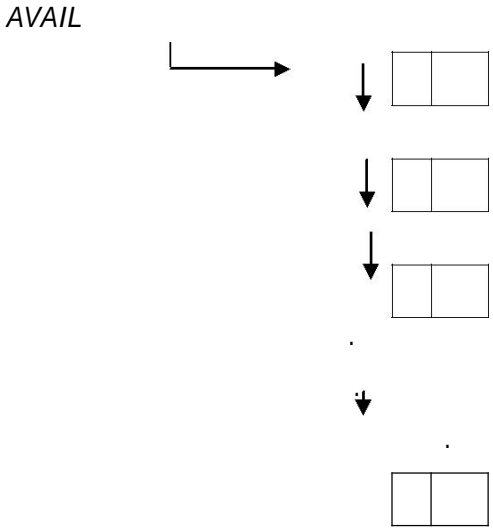
Each node consists of two fieds,

- *information field called INFO that holds the actual element on the list and*
- *a pointer pointing to next element of the list called LINK ,ie. It holds the address of the next element.*

The name of a typical element is denoted by NODE. Pictorially, the node NODE



Structure is given as follows It is assumed that an available area of storage for this node structure consists of a stack of available nodes, as shown below:



Here, the pointer variable AVAIL contains the address of the top node in the stack. The head and tail are pointers pointing to first and last element of the list repectively. For an empty list the head and tail have the value NIL. When the list has one element, the head and tail point to the same.

OPERATIONS*:*

INSERTION IN A LIST:

Inserting a new item,say 'x' ,into the list has three situations:

1. *Insertion at the front of the list*
2. *Insertion in the middle of the list or in the order*
3. *Insertion at the end of the list*

INSERTION AT FRONT:

Algorithms for placing the new item at the beginning of a linked list:

1. *Obtain space for new node*
2. *Assign data to the item field of new node*
3. *Set the next field of the new node to point to the start of the list*
4. *Change the head pointer to point to the new node.*

Function *INSERT(X, FIRST)*

Variables used:

X ← *New element to be inserted*

FIRST ← *Pointer to the first element whose node contains INFO and LINK fields.*

AVAIL ← *Pointer to the top element of the availability stack*

NEW ← *Temporary pointer variable.*

1. [Underflow?] If

AVAIL = NULL

Then Write('AVAILABILITY STACK UNDERFLOW')

Return(FIRST)

2. [Obtain address of next free node]

NEW ← *AVAIL*

3. [Remove free node from availability stack]

AVAIL ← *LINK(AVAIL)*

4. [Initialize fields of new node and its link to the list]

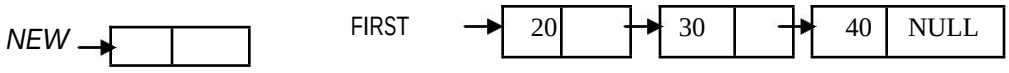
INFO(NEW) ← *X*

LINK(NEW) ← *FIRST*

5. [Return address of new node]

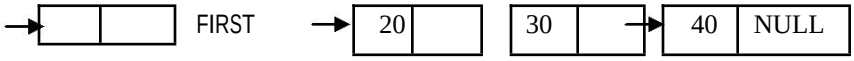
Return(NEW)

Step 1:

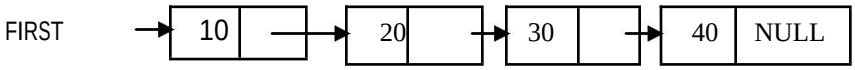


Step 2:

NEW 10



Step 3:



Algorithm for inserting the new node x between the two existing nodes, say N1 and N2(or in order):

1. Set space for new node x

2. Assign value to the item field of x

3. Search for predecessor node n1 of x

4.Set the next field of x to point to link of node n1(node N2)

5.Set the next field of N1 to point to x.

Function *INSORD(X, FIRST)*

Variables used:

X ← *new element*

FIRST ← *Pointer to the first element whose node contains INFO and LINK fields.*

AVAIL ← *pointer to the top element of the availability stack*

NEW,SAVE ← *Temporary pointer variables*

1. *[Underflow?]*

   If AVAIL = NULL

   Then Write('AVAILABILITY STACK UNDERFLOW')

   Return(FIRST)

2. *[Obtain address of next free node]*

   NEW ← AVAIL

3. *[Remove free node from availability stack] AVAIL ← LINK(AVAIL)*
4. *[Copy information contents into new node] INFO(NEW) ← X*
5. *[Is the list empty?]*

   If FIRST = NULL

   Then LINK(NEW) ← FIRST
6. *[Does the new node precede all others in the list?]*

   If INFO(NEW) ≤ INFO(FIRST)

   Then LINK(NEW) ← FIRST

   Return(NEW)
7. *[Initialize temporary pointer]*

   SAVE ← FIRST
8. *[Search for predecessor of new node]*

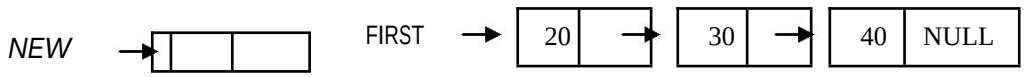
   Repeat while LINK(SAVE) ≠ NULL and INFO(LINK(SAVE))≤ INFO(NEW)

   SAVE ← LINK(SAVE)
9. *[Set link fields of new node and its predecessor]*

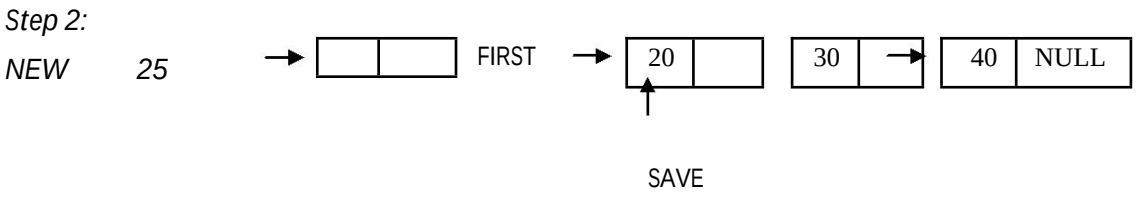
   LINK(NEW) ←
   LINK(SAVE)
   LINK(SAVE) ← NEW
10. *[Return first node pointer]*

    Return(FIRST)

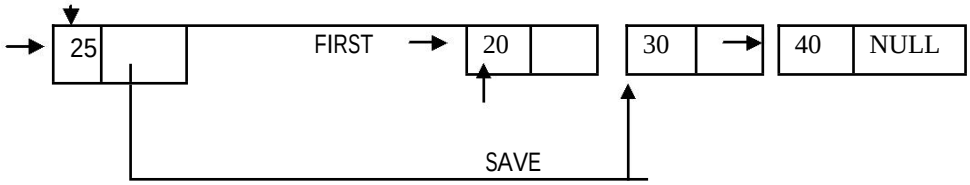
Step 1:



Step 2:



Step 3:



Step 4:

SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE

Algorithms for inserting an item at the end of the list:

1.Set space for new

node x 2.Assign value

to the item field of x

3.Set the next field of x

to NULL

4.Set the next field of N2 to point to x

Function *INSEND(X,FIRST)*

Variables used:

X ← *new element*

FIRST ← *Pointer to the first element whose node contains INFO and LINK fields.*

AVAIL ← *pointer to the top element of the availability stack*

NEW,SAVE ← *Temporary pointer variables*

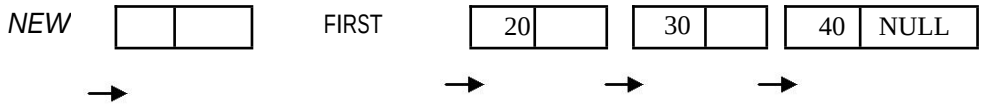
1. *[Underflow?] If AVAIL = NULL*

   Then Write('AVAILABILITY STACK UNDERFLOW')

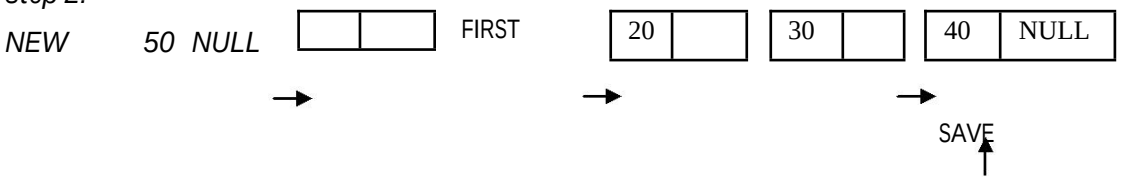
   Return(FIRST)

2. *[Obtain address of next free node]*
   NEW ← AVAIL
3. *[Remove free node from availability stack] AVAIL ← LINK(AVAIL)*
4. *[Initialize fields of new node]*
   INFO(NEW) ← X
   LINK(NEW) ← NULL
5. *[Is the list empty?]*
   If FIRST = NULL
   Then Return(NEW)
6. *[Initiate search for the last node] SAVE ← FIRST*
7. *[Search for end of list]*
   Repeat while LINK(SAVE) ≠ NULL
   SAVE ← LINK(SAVE)
8. *[Set LINK field of last node to NEW] LINK(SAVE) ← NEW*
9. *[Return first node pointer]*
   Return(FIRST)

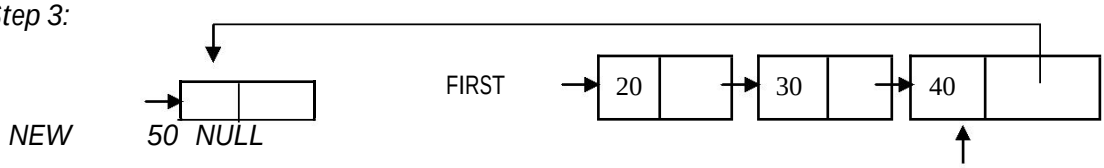
Step 1:



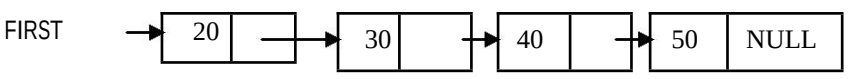
Step 2:



Step 3:



Step 4:



Note that no data is physically moved, as we had seen in array implementation. Only the pointers are readjusted.

DELETING AN ITEM FROM A LIST:

Deleting a node from the list requires only one pointer value to be changed, there we have situations:

1.Deleting the first item

2.Deleting the last item

3.Deleting between two nodes in the middle of the list.

Algorithms for deleting the first item:

1.If the element x to be deleted is at first store next field of x in some other variable y.

2.Free the space occupied by x

3.Change the head pointer to point to the address in y.

Algorithm for deleting the last item:

1.Set the next field of the node previous to the node x which is to be deleted as NULL

2.Free the space occupied by x

Algorithm for deleting x between two nodes N1 and N2 in the middle of the list:

1.Set the next field of the node N1 previous to x to point to the successor field N2 of the node x.

2.Free the space occupied by x.

Procedure *DELETE(X, FIRST)*

Variables used:

X ← *New element to be inserted*

FIRST ← *Pointer to the first element whose node contains INFO and LINK fields.*

TEMP ← *To find the desired node*

PRED ← *keeps track of the predecessor of TEMP*

1. [Empty list?]

If FIRST = NULL

Then Write('UNDERFLOW")

Return

2. [Initialize search for X] TEMP ← *FIRST*

3. [Find X]

Repeat thru step 5 while TEMP ≠ X and LINK(TEMP) ≠ NULL

4. [Update predecessor marker] PRED ← *TEMP*

5. [Move to next node]

TEMP ← LINK(TEMP)

6. [End of the list]

If TEMP ≠ X

Then Write('NODE NOT FOUND")

Return

7. [Delete X]

If X = FIRST (Is X the first node?)

Then FIRST ← LINK(FIRST)

Else LINK(PRED) ← LINK(X)

8. [Return node to availability area]

LINK(X) ← AVAIL

AVAIL ← X

Return

SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution & Accredited by NAAC with "A" Grade)
(An Autonomous Institution)
Madagadipet, Puducherry - 605 107

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject Name: **Data Structures**
Subject Code:

Prepared by:

Verified by: **Approved by:**

UNIT – IV

TREES

Trees: Basic Tree Terminologies, Different types of Trees: Binary Tree, Threaded Binary Tree, Binary Search Tree, Binary Tree Traversals, AVL Tree. Introduction to B-Tree and B+ Tree.

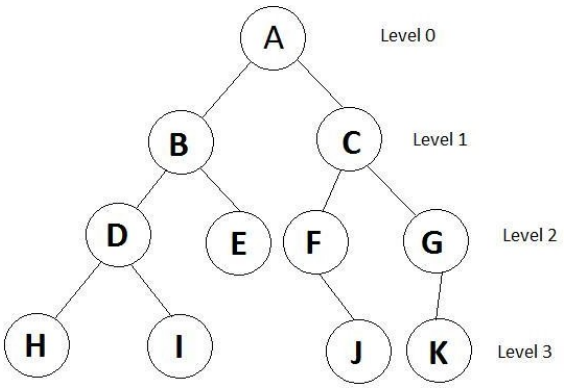
1. Define tree?

   Tree is a non linear data structure. There is no linear relation between the data items. It can be defined as finite set of more than one node.

   There is a special node designated as root node.

   The remaining nodes are partitioned into sub tree of a tree.

   The below diagram shows the tree

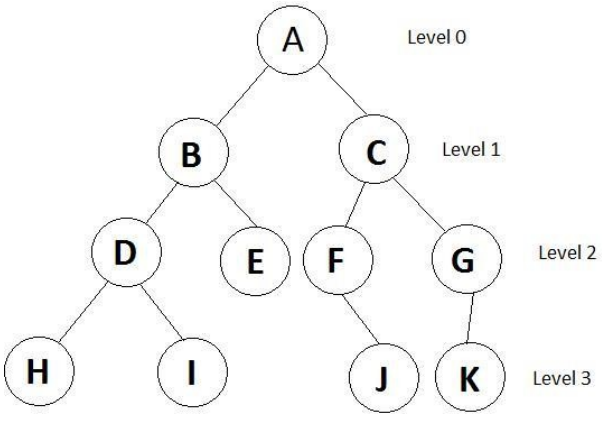


2. **Define path in tree?**

   The path in a tree is referred as the nodes in which the successive nodes are connected by the edge in a tree. For example: the path from A to l

   A-B, B-D, D-I.

3. **Define terminal nodes in a tree?**
   A node that has no children is called as a terminal node. It is also referred as a leaf node. These nodes have degree has zero.
4. **Define non-terminal nodes in a tree?**
   All intermediate nodes that traverse the given tree from its root node to the terminal nodes are referred as non-terminal nodes.
5. **Define branch, siblings & ancestors?**
   Branch or edge of a tree is called as the link or connection between two nodes. The nodes having the same parent are called siblings. The ancestor of a node is referred as all nodes along the path of root node to the path node.
6. **State the properties of the tree?**
   Any node can be the root of the tree.
   If the root is identified; then that tree is called as the rooted node.
   If it is not identified; then that tree is called as the free tree.
   Every node, expect the root node has a unique parent.
7. **Define degree?**
   The degree of a node is referred as the number of sub-trees of a particular node.
8. **Define a binary tree?**
   A binary tree is tree, which has nodes either empty or not more than two child nodes each of which may be a leaf node.



9. **Define a full binary tree?**
   A full binary tree is a tree in which all the leaves are on the same level and every non-leaf node has exactly two children.
10. **Define complete binary tree?**

A complete binary tree is a tree in which every non-leaf node has exactly two children not necessarily to be on the same level.

11. State the properties of binary tree?

The maximum number of nodes on level n of the binary tree is 2^{n-1} where n>=1 The maximum number of nodes in a binary tree of height n is 2^{n-1} where n>=1. For any non empty tree $n_l = n_d + 1$ where n_l is the number of leaf nodes and n_d is the number of nodes of degree 2.

12. What are the different ways of representing binary tree?

- Linear representation using arrays
- Linked representation using pointers

13. What is meant by binary tree traversal?

Traversing a binary tree means moving through all the nodes in the binary tree visiting each node in the tree only once.

14. What are the different types of binary tree traversal?

- Preorder
- Inorder
- Postorder

15. What are the tasks performed during preorder tree traversal?

Process the root node.

Traverse the left subtree

Traverse the right subtree

16. What are the tasks performed during inorder tree traversal?

Traverse the left subtree

Process the root node.

Traverse the right subtree

17. What are the tasks performed during postorder tree traversal?

Traverse the left subtree

Traverse the right subtree

Process the root node.

18. State the merits and demerits of linear representation of binary tree?

Merits:

Storage method is easy and can be easily implemented in arrays.

When the location of a parent /child node is known other one can be determined easily. It requires static memory allocation so it is easily implemented in all programming language

Demerits:

Insertions and deletions in a node, taker an excessive amount of processing time due to data movement up and down the array.

19. State the merits and demerits of linked representation of binary tree?

Merits:

Insertions and deletions in a node, involves no data movement except the re arrangement of pointers, hence less processing time.

Demerits:

Given a node structure, it is difficult to determine its parent node.

Memory spaces are wasted for storing null pointers for the nodes, which have one or no subtrees. It requires dynamic memory allocation, which is not possible in some programming languages.

20. What do you mean by general tree?

General tree is a tree with nodes having any number of children

21. What is the length of the path in a tree?

The path length of a tree is the sum of the levels of all the tree's nodes. The path length of a tree with N nodes is the sum of the path lengths of the subtrees of its root plus N-1.

22. Define B+ tree indexing?

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves. The primary values of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, file systems.

23. What are the applications of binary trees?
    1. Binary search tree
    2. Binary tries
    3. Hash trees
    4. Heaps
    5. T-tree
    6. Syntax tree
    7. Huffman coding tree
24. What is the use of indexing technique?
    Indexing is an auxiliary data structure which speeds up the record retrieval.
25. Compare B- tree and B+ tree?
    a. A B+ tree is the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is the same length. B+ trees are good for searches but cause some overhead issues in wasted space.
    b. B-trees are similar to B+ trees but it allows search-key values to appear only once, eliminates redundant storage of search keys. It is possible sometimes to find search-key value before reaching the leaf node. Implementation is harder than B+ trees.

26. Differentiate binary tree and binary search tree?
    a. Binary is a specialized form of tree with two child (left child and right child). It is simply representation of data in tree structure.
    b. Binary search tree is a special type of binary tree that follows the following condition:
        i. Left child is smaller than its parent node
        ii. Right child is greater than its parent node
27. Give some of the applications of B tree?

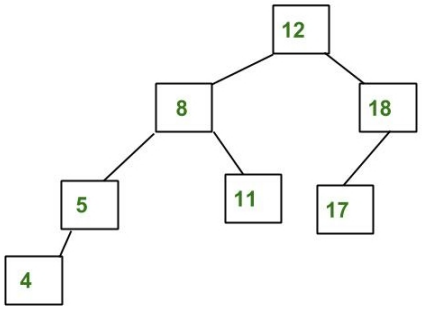
B tree is optimized for systems that read and write large blocks of data. It is commonly used in databases and file systems.

B tree also optimizes costly disk accesses that are of concern when dealing with large data sets.

28. Define AVL tree?

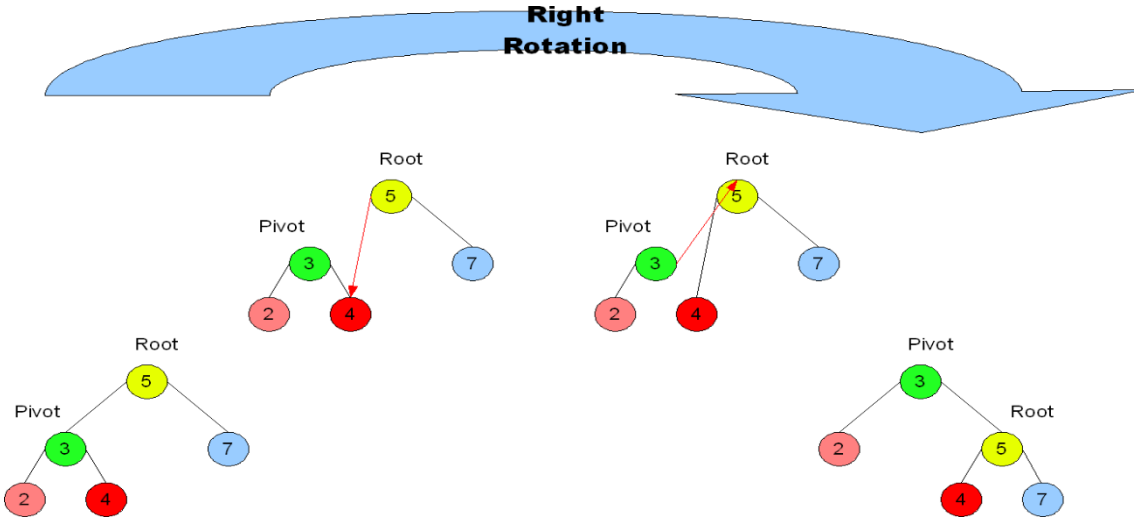
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

29. Perform RR rotation in AVL tree with an example.



5 MARKS

1. EXPLAIN THE TREE TERMINOLOGIES ROOT, EDGE, PARENT.

- In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. A tree data structure can be defined as follows...

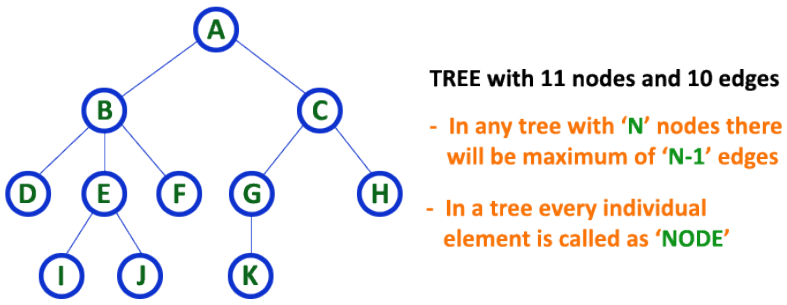
Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

- A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively

- In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure. In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

Example

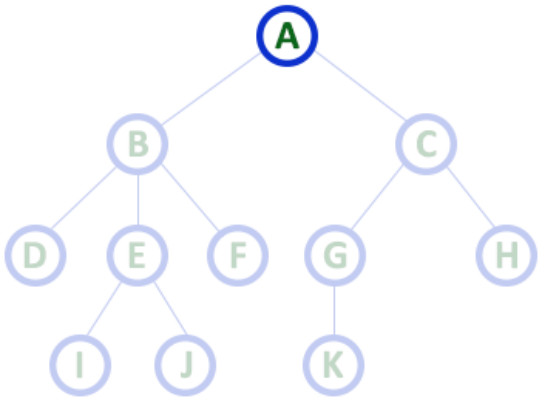


Terminology

In a tree data structure, we use the following terminology...

1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

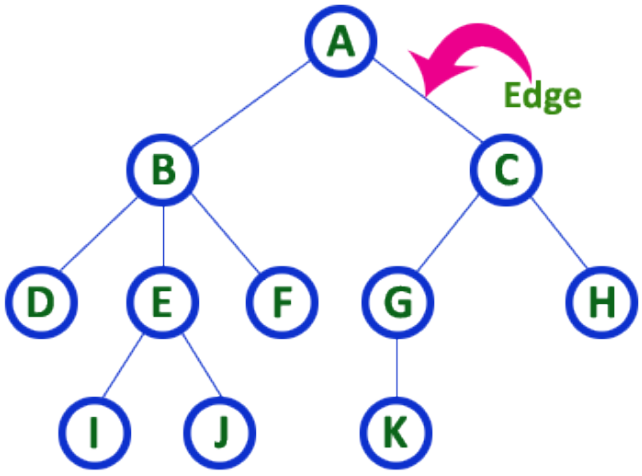


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

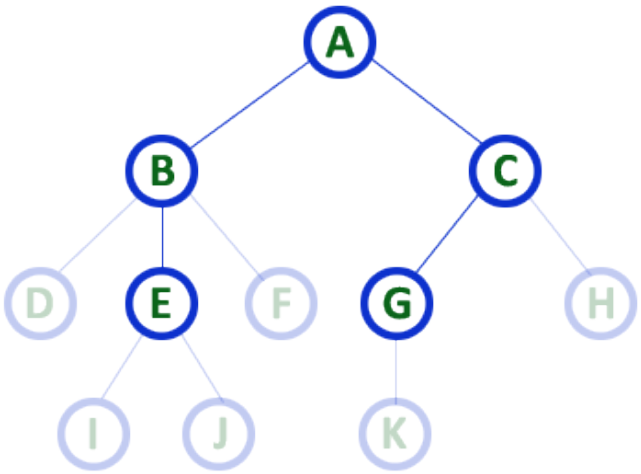
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".



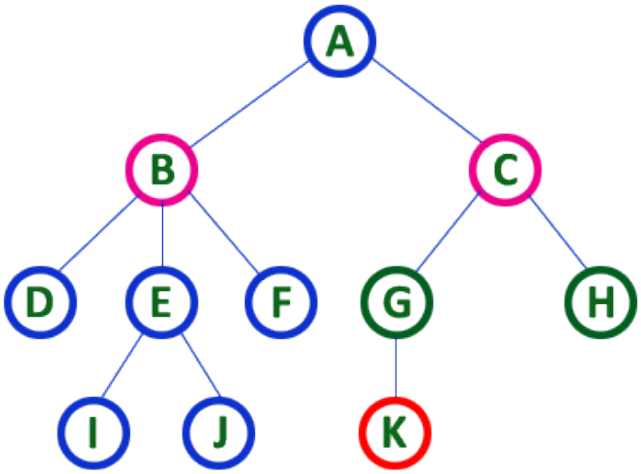
Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

2. EXPLAIN THE TREE TERMINOLOGIES LEAF, SIBLING, CHILD.

1. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are Children of A

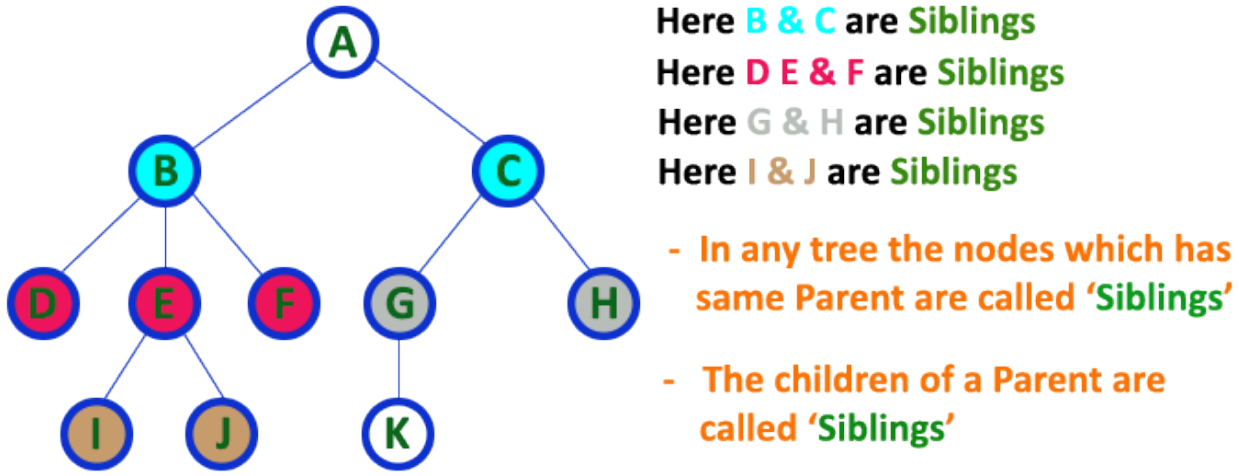
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

2. Siblings

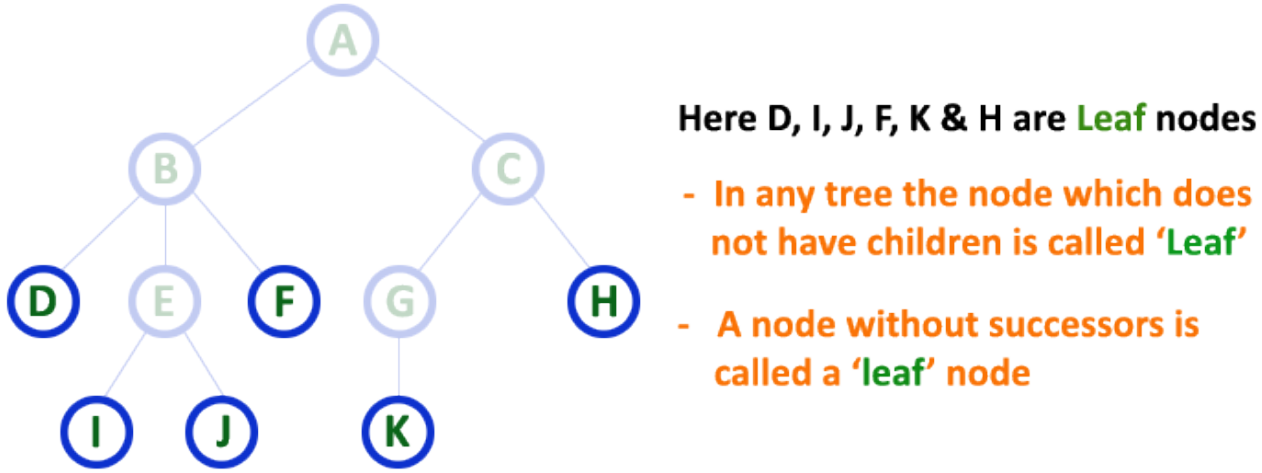
In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.



3. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.

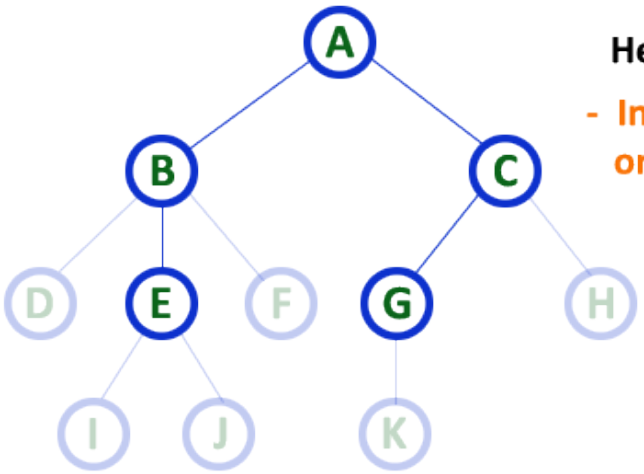


3. EXPLAIN THE TREE TERMINOLOGIE INTERNAL NODES, DEGREE, LEVEL.

1. Internal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



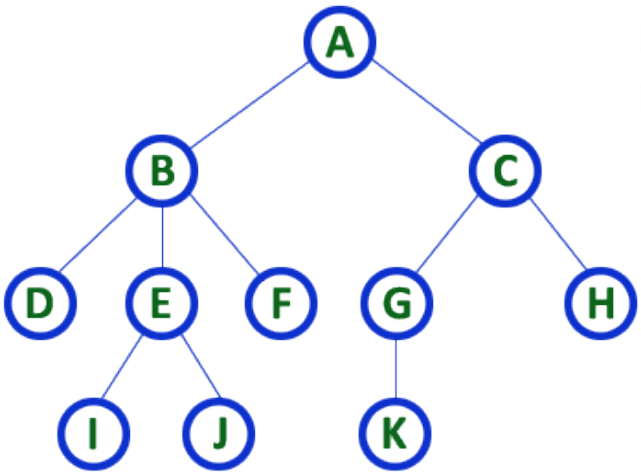
Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node

- Every non-leaf node is called as 'Internal' node

2. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



Here Degree of B is 3

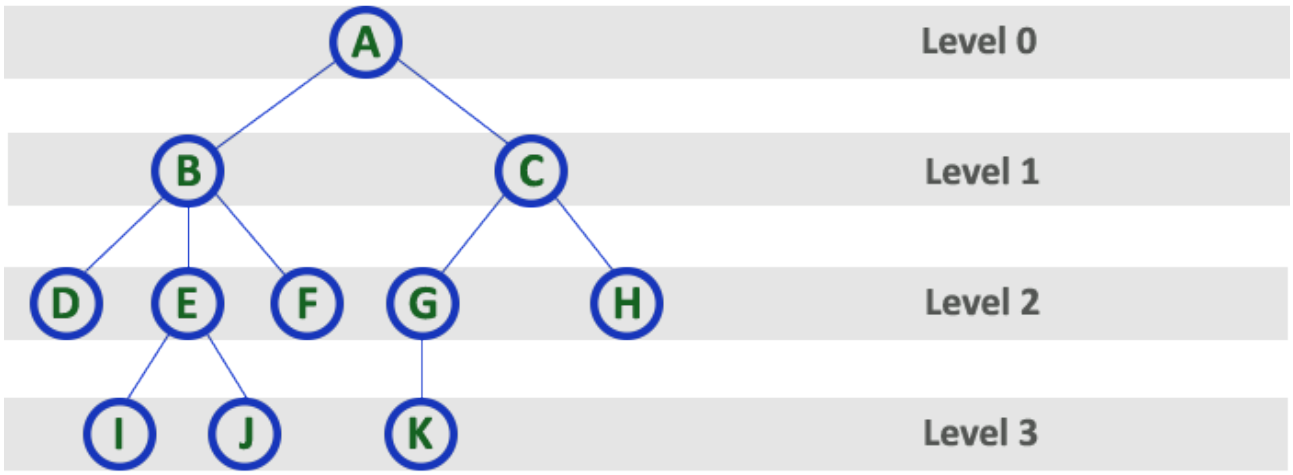
Here Degree of A is 2

Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

3. Level

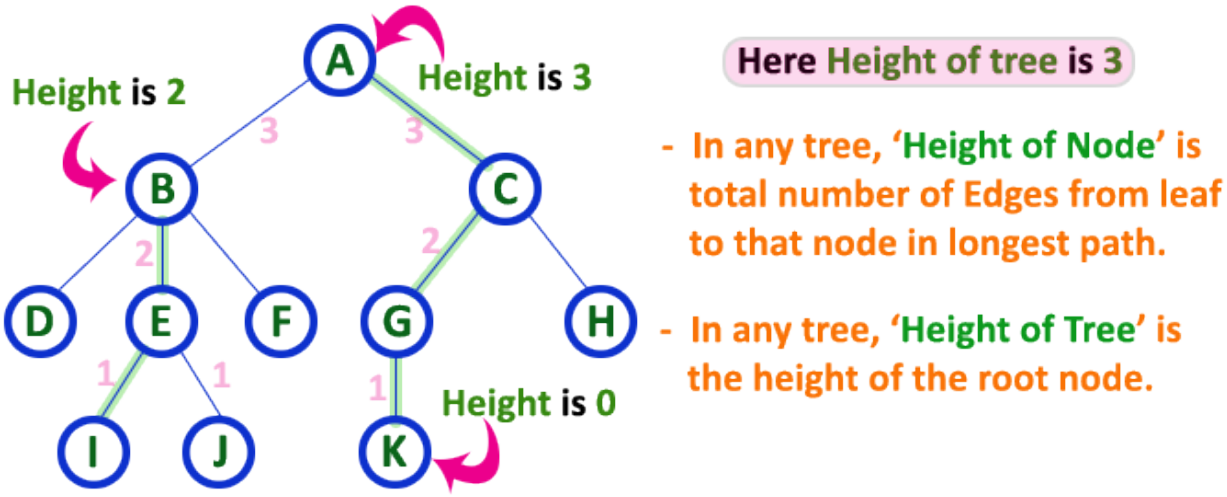
In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



4. EXPLAIN THE TREE TERMINOLOGIES HEIGHT, DEPTH, PATH, SUB TREE.

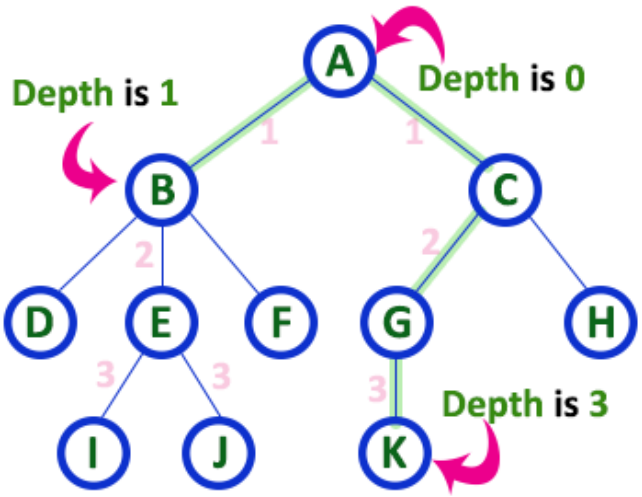
1. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**



2. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

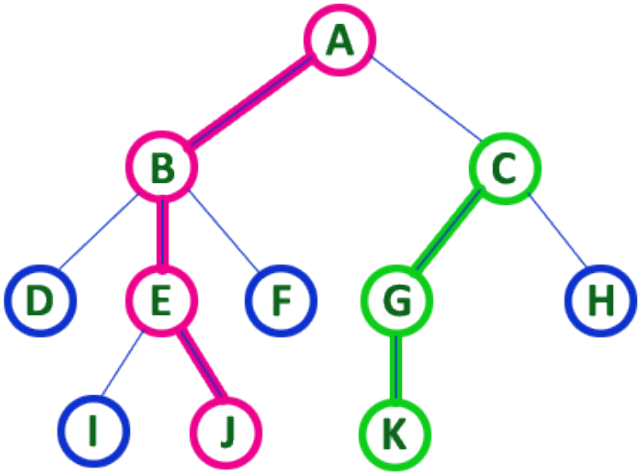


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

3. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



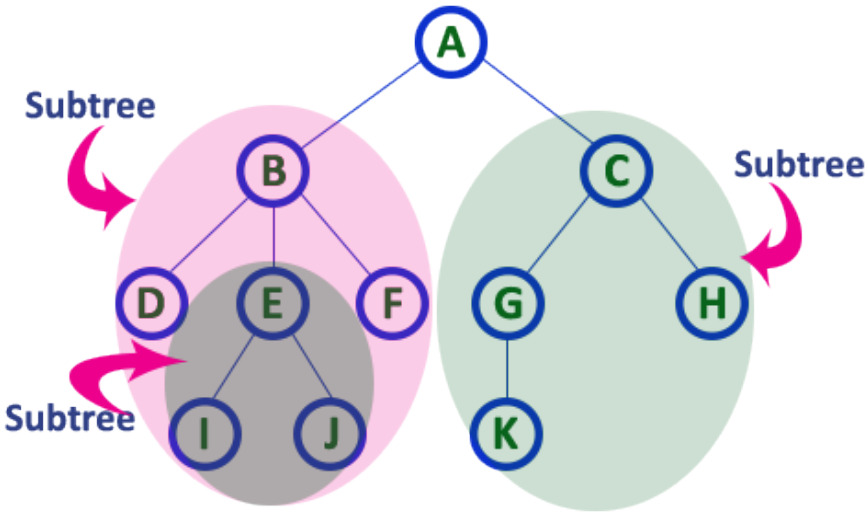
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

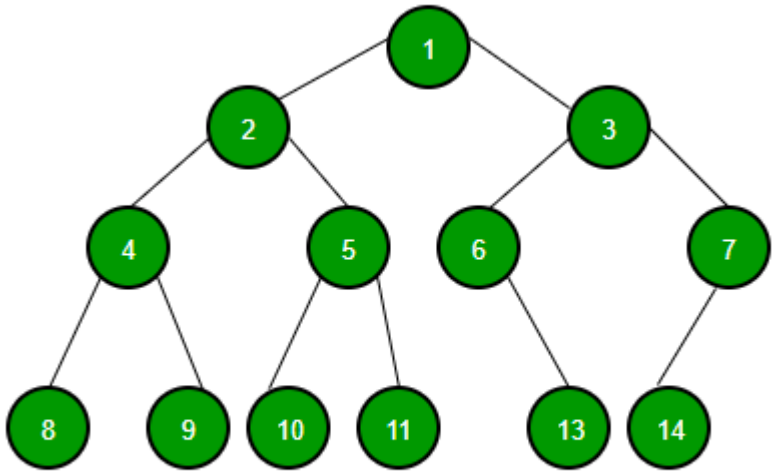
4. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



5. **WHAT IS BINARY TREE AND EXPLAIN THE STRICTLY BINARY TREE.**

- A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



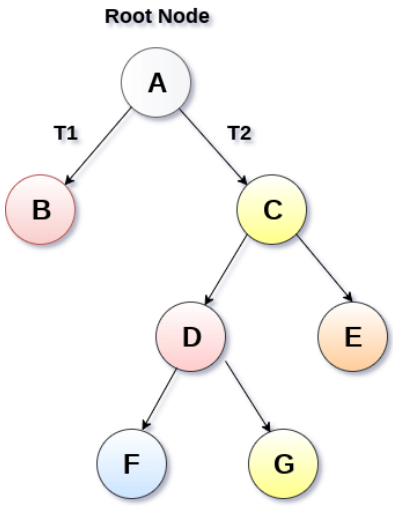
A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child

Strictly Binary Tree

In Strictly Binary Tree, every non-leaf node contain non-empty left and right sub-trees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves, will have (2n - 1) nodes.

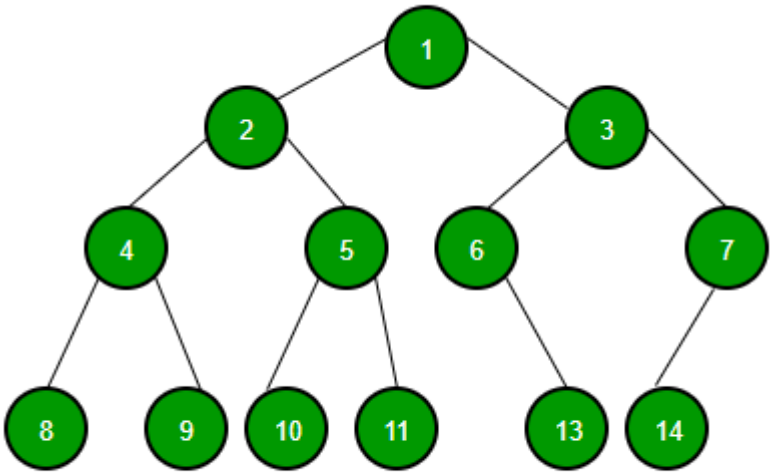
A strictly binary tree is shown in the following figure.



Strictly Binary Tree

6. **WHAT IS BINARY TREE AND EXPLAIN THE COMPLETE BINARY TREE.**

- A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

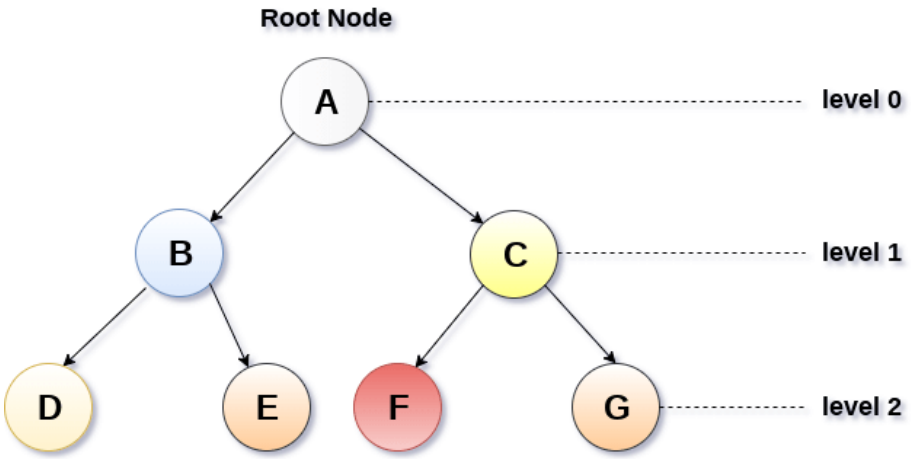


A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child

Complete Binary Tree

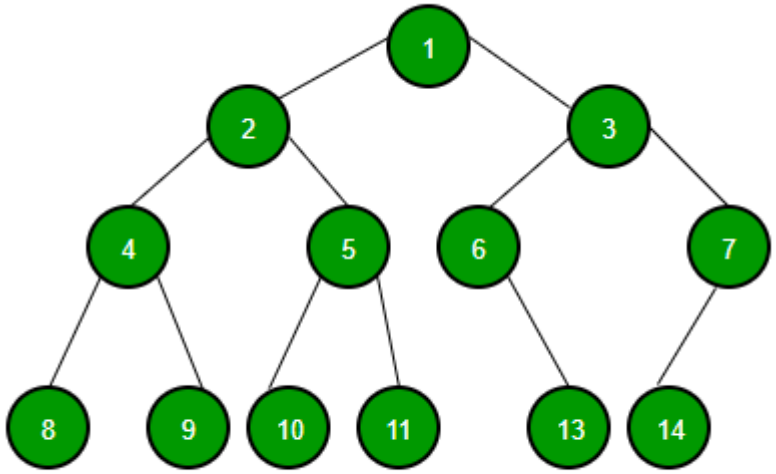
A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d. A complete binary tree is a binary tree that contains exactly 2^l nodes at each level between level 0 and d. The total number of nodes in a complete binary tree with depth d is 2^{d+1}-1 where leaf nodes are 2^{d} while non-leaf nodes are 2^{d}-1.



Complete Binary Tree

7. WHAT IS BINARY TREE AND EXPLAIN THE EXTENDED BINARY TREE

- A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



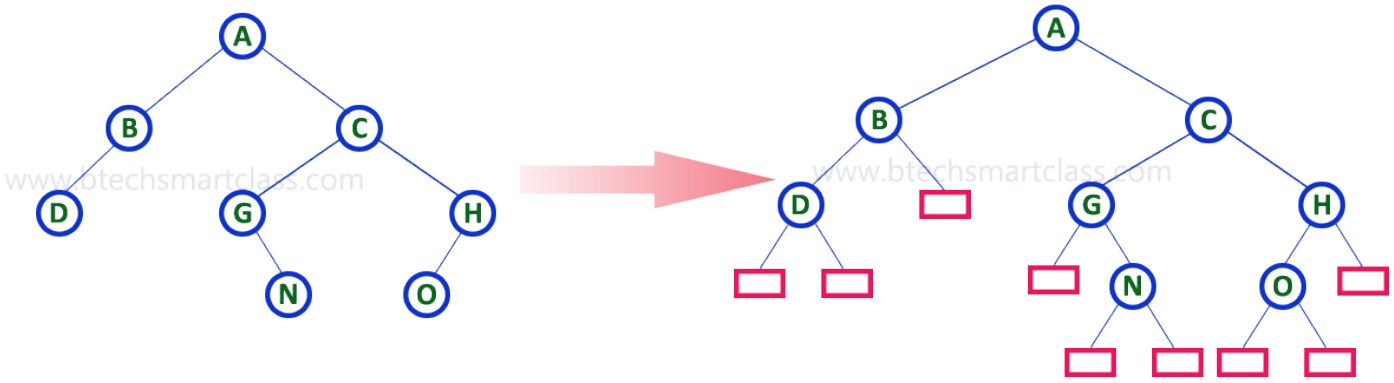
A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child

Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



8. EXPLAIN THE BINARY TREE TRAVERSALS.

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

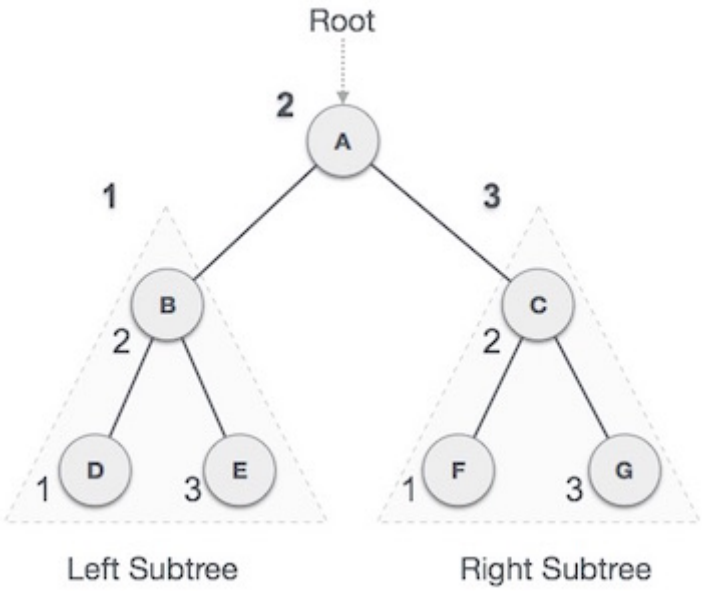
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

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 subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed −

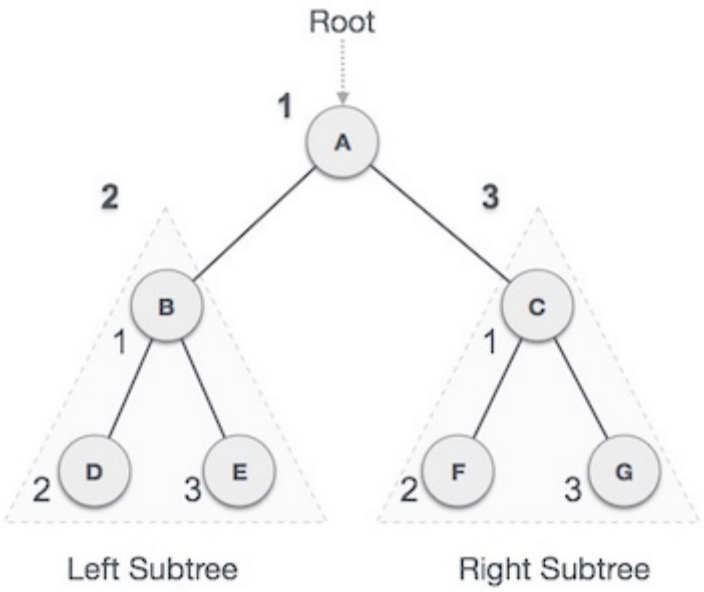
Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

Until all nodes are traversed −

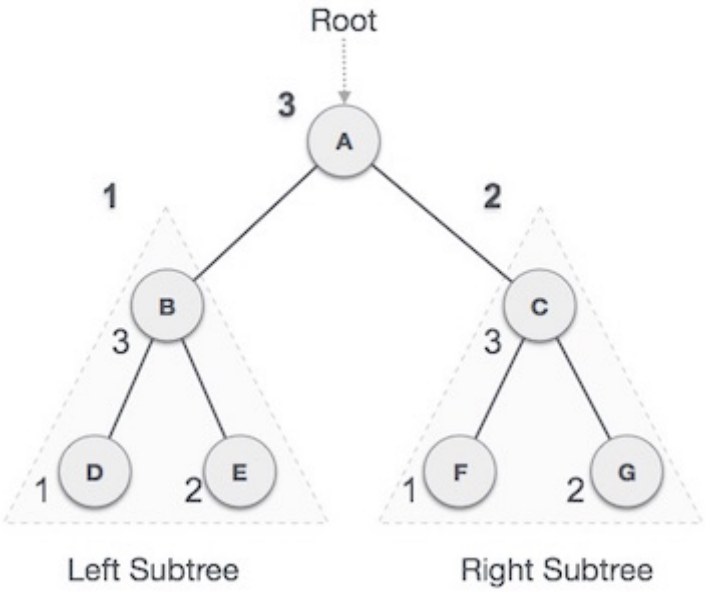
Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

9. EXPLAIN B TREE.

- B-Tree is a self-balancing search tree. In most of the other self-balancing search trees , it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory.
- When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses.
- Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size.
- Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.
  Time Complexity of B-Tree:

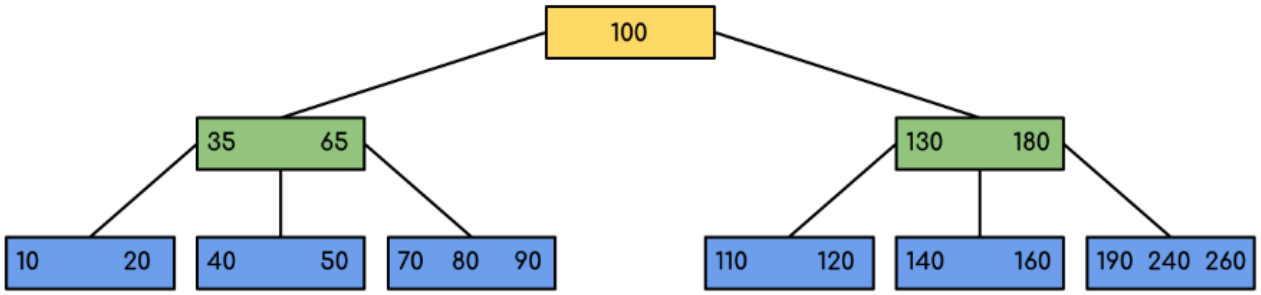
SR. NO.	ALGORITHM	TIME COMPLEXITY
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

“n” is the total number of elements in the B-tree.

Properties of B-Tree:

1. All leaves are at the same level.
2. A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.
3. Every node except root must contain at least (ceiling)([t-1]/2) keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most t – 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 sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).

Following is an example of B-Tree of minimum order 5. Note that in practical B-Trees, the value of the minimum order is much more than 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 and have keys one less than the number of their children.

10. EXPLAIN THE B+ TREE.

- In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of B-tree used for indexing, however is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique, greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record.
- B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. Moreover, the leaf nodes are linked to provide ordered access to the records. The leaf nodes, therefore form the first level of index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

From the above discussion it is apparent that a B+ tree, unlike a B-tree has two orders, ‘a’ and ‘b’, one for the internal nodes and the other for the external (or leaf) nodes.

The structure of the internal nodes of a B+ tree of order ‘a’ is as follows:

1. Each internal node is of the form :
   $<P_1, K_1, P_2, K_2, ….., P_{c-1}, K_{c-1}, P_c>$
   where c <= a and each **P_i is a tree pointer (i.e points to another node of the tree)** and, each **K_i is a key value** (see diagram-I for reference).

2. Every internal node has : $K_1 < K_2 < …. < K_{c-1}$
3. For each search field values ‘X’ in the sub-tree pointed at by P_i, the following condition holds :
   $K_{i-1} < X <= K_i$, for $1 < i < c$ and,
   $K_{i-1} < X$, for $i = c$
   (See diagram I for reference)
4. Each internal nodes has at most ‘a’ tree pointers.
5. The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.
6. If any internal node has ‘c’ pointers, c <= a, then it has 'c – 1' key values.

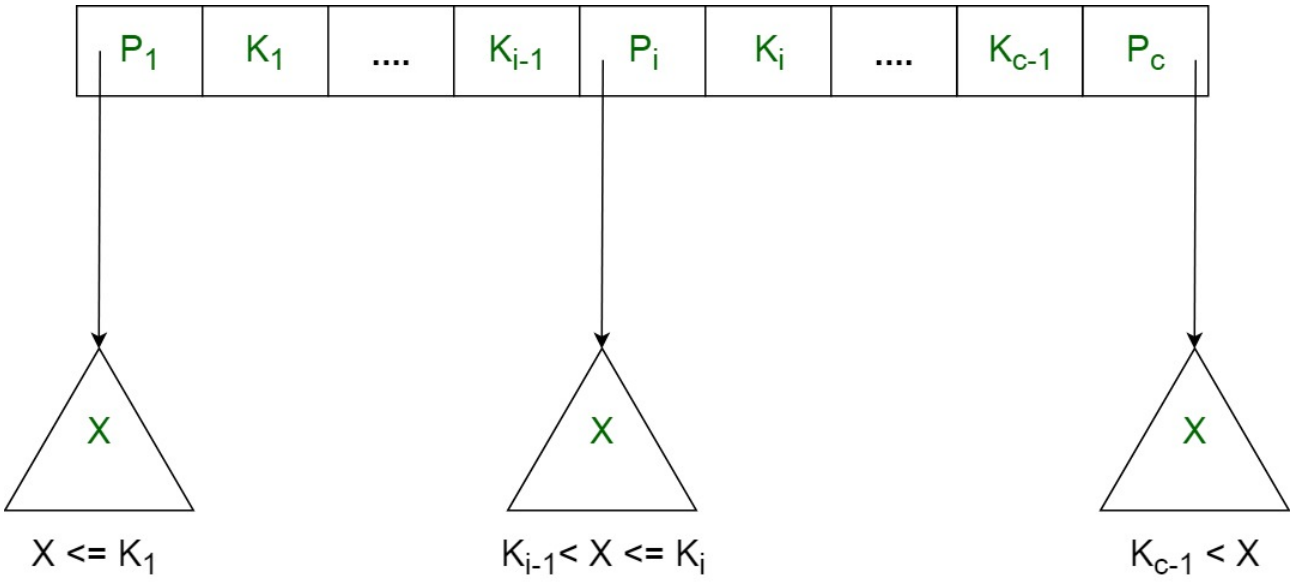


Diagram-I

The structure of the leaf nodes of a B+ tree of order 'b' is as follows:

1. Each leaf node is of the form :
   $<<K_1, D_1>, <K_2, D_2>, ….., <K_{c-1}, D_{c-1}>, P_{next}>$
   where c <= b and each **D_i is a data pointer (i.e points to actual record in the disk whose key value is K_i or to a disk file block containing that record)** and, each **K_i is a key value** and, **P_{next} points to next leaf node in the B+ tree** (see diagram II for reference).
2. Every leaf node has : $K_1 < K_2 < …. < K_{c-1}$, c <= b
3. Each leaf node has at least \ceil(b/2) values.
4. All leaf nodes are at same level.

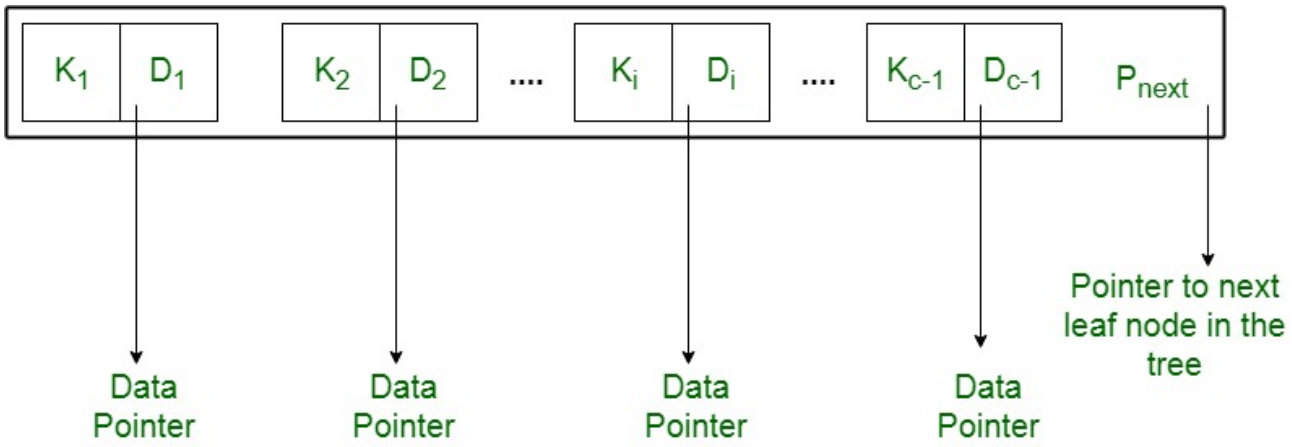
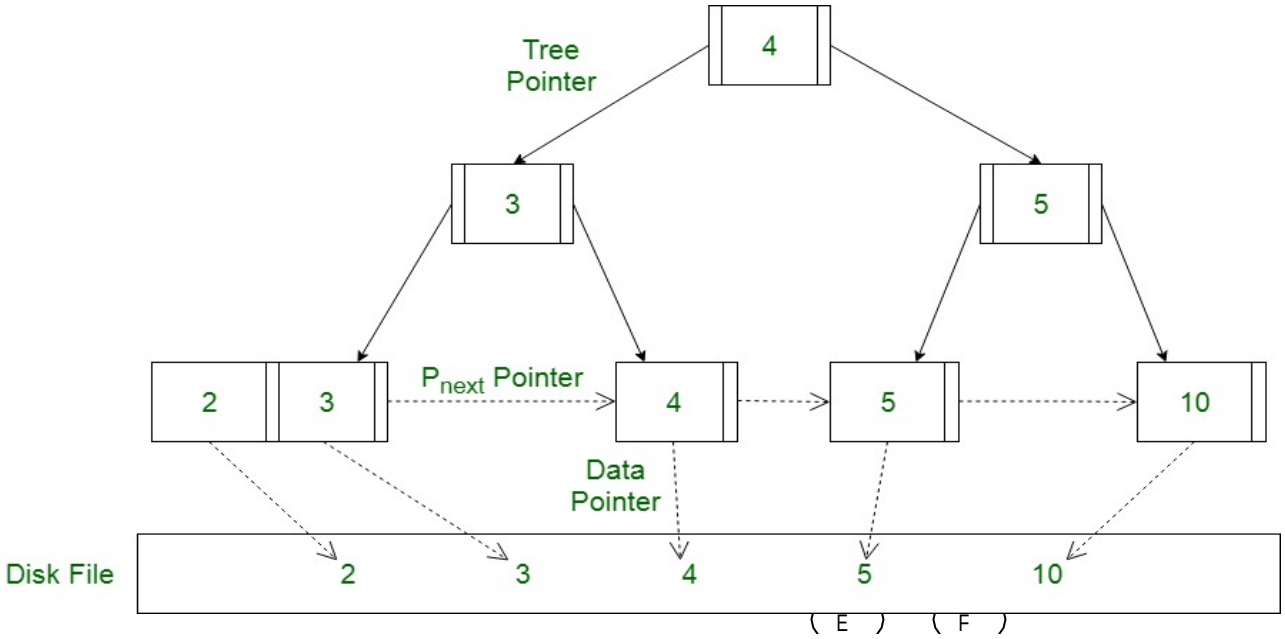


Diagram-II

Using the P_{next} pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving ordered access to the records stored in the disk.

A Diagram of B+ Tree –



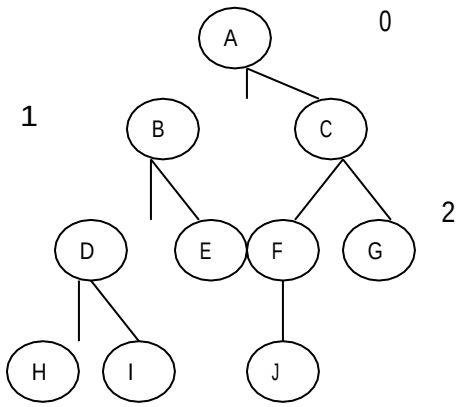
10 Marks

1. Explain about Basic Tree Terminology with need diagrams.

Tree: is defined as a finite set of one or more nodes such that

- ✓ There is one specially designated node called ROOT.
- ✓ The remaining nodes are partitioned into a collection of sub-trees of the root each of which is also a tree.

Example *LEVEL*



NODE: stands for item of information.

The nodes of a tree have a parent-child relationship. The root does not have a parent; but each one of the other nodes has a parent node associated to it. A node may or may not have children is called a leaf node or terminal nodes.

A line from a parent to a child node is called a branch. If a tree has n nodes, one of which is the root there would be n-1 branches.

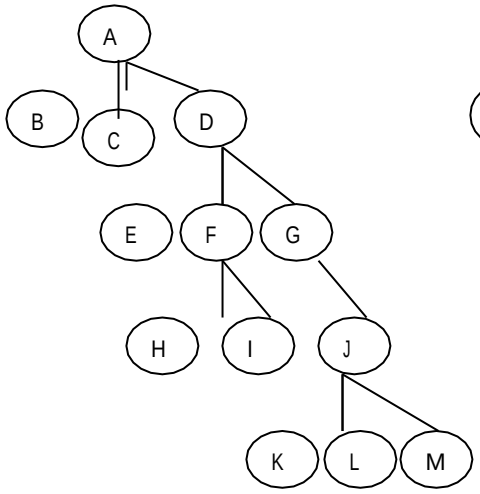
The number of sub-trees of a node is called its degree. The degree of A is 2, F is 1 and J is zero. The leaf node is having degree zero and other nodes are referred as non-terminals.

The degree of a tree is the maximum degree of the nodes in the tree.

Nodes with the same parent are called siblings. Here D & E are all siblings. H & I are also siblings.

The level of a node is defined by initially letting the root be at level zero. If a node is al level l, there its children are at level l+1.

The height or depth of a tree is defined to be the maximum level of any node in the tree. A set of trees is called forest; if we remove the root of a tree we get a forest. In the above fig, if we remove A, we get a forest with three trees.



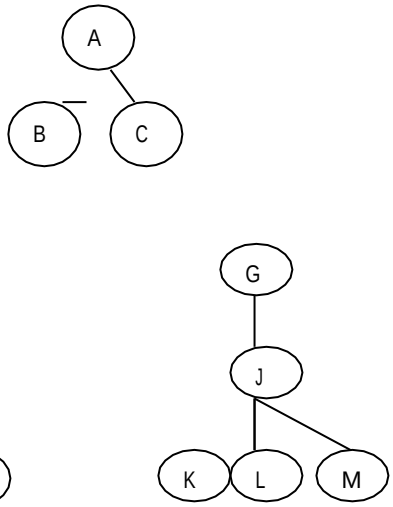


FIG: Tree *FIG: Forest (Sub-trees)*

Properties of a Tree

1. Any node can be the root of the tree and each node in a tree has the property that there is exactly one path connecting that node with every other node in the tree.

2. The tree in which the root is identified is called a rooted tree; a tree in which the root is not identified is called a free tree.

3. Each node, except the root, has a unique parent.

Binary Trees

A binary tree is a tree, which is, either empty or consists of a root node and two disjoint binary trees called the left sub-tree and right sub-tree.

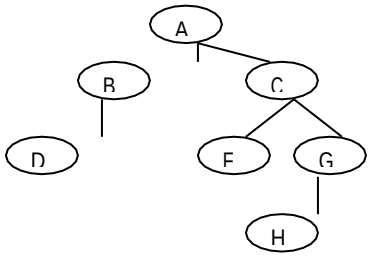


FIG: Binary Tree

In a binary tree, no node can have more than two children. So every binary tree is a tree, not every tree is a binary tree.

A complete binary tree is a binary tree in which all interval nodes have degree and all leaves are at the same level.

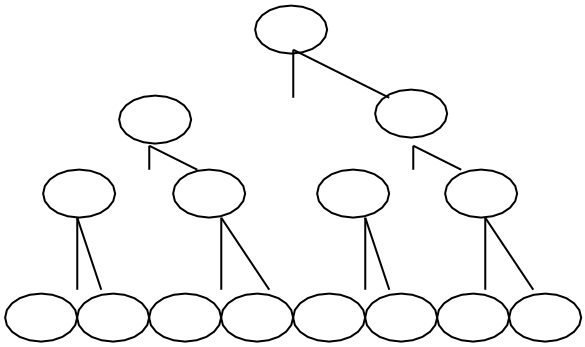


FIG: Complete binary tree

If every non-leaf node in a binary tree has non-empty left and right sub-trees, the tree is termed as strictly binary tree.

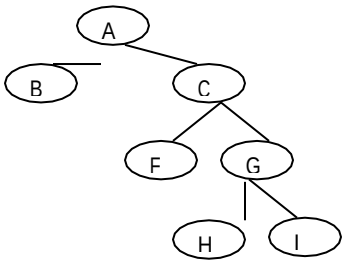


FIG: Strictly binary tree

If A is the root of a binary tree and B is the root of its left or right sub-tree, then A is said to be the parent of B and B is said to be the left or right child of A.

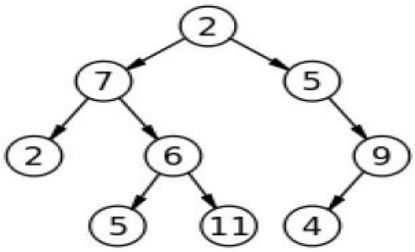
Node n1 is an ancestor of node n2, if n1 is either the parent of n2 or the parent of some ancestor of n2. Here n2 is a descendant of n1, a node n2 is a left descendant of node n1 if n2 is either the left child of n1 or a descendant of the left child of n1.

A right descendant may be similarly defined the number of nodes at level i is 2^i. For a complete binary tree with k levels contains 2i nodes.

2. Write briefly about the node representation of Binary trees.

Binary Tree:

A binary tree is a tree in which each node has atmost 2 childrens. (i.e) a node can have 0 child or 1 child or 2 child. A node must not have more than 2 childrens.



1. Linear Representation Of A Binary Tree

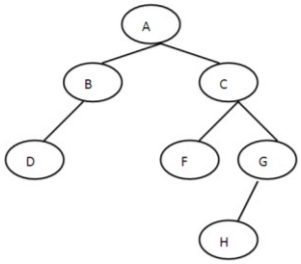
The linear representation method of implementing a binary tree uses a one-dimensional array of size ((2^d+1)-1) where d is the depth of the tree.

Once the size of the array has been determined the following method is used to represent the tree.

1. Store the root in 1st location of the array.
2. If a node is in location n of the array store its left child at location 2n and its right child at (2n+1).

In c, arrays start at position 0; therefore instead of numbering the trees nodes from 1 to n, we number them from 0 to n-1. The two child of a node at position P are in positions 2P+1 and 2P+2.

The following figure illustrates arrays that represent the almost complete binary trees.



0	1	2	3	4	5	6	7	8
A	B	C	D	E	F	G	H	

We can extend this array representation of almost complete binary trees to an array representation of binary trees generally.

The following fig (A) illustrates binary tree and fig (B) illustrates the almost complete binary tree of fig (a). Finally fig(C) illustrates the array implementation of the almost complete binary tree.

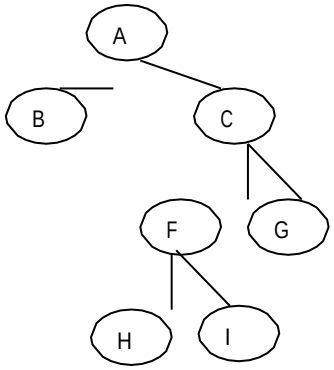


Fig (A) Binary tree

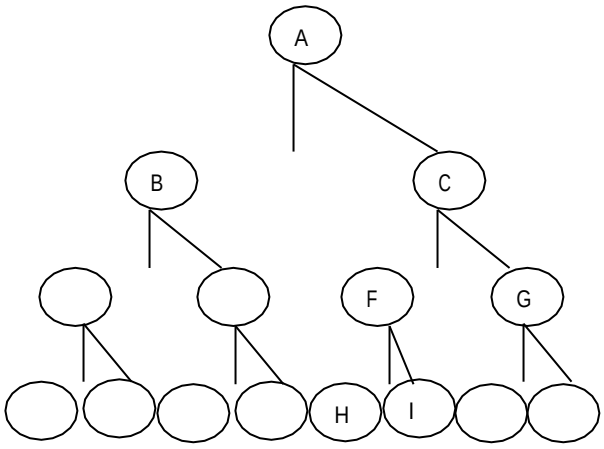


Fig (B) Almost a complete binary tree

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C			F	G					H	I

Fig (C) Array Representation

Advantages

1. Given a child node, its parent node can be determined immediately. If a child node is at location N in the

array, then its parent is at location N/2.

2. It can be implemented easily in languages in which only static memory allocation is directly available.

Disadvantages

1. Insertion or deletion of a node causes considerable data movement up and down the array, using an excessive amount of processing time.
2. Wastage of memory due to partially filled trees.

2. Linked List Representation

Linked lists most commonly represent binary trees. Each node can be considered as having 3 elementary fields : a data field, left pointer, pointing to left sub-tree and right pointer pointing to the right sub-tree.

The following figure is an example of linked storage representation of a binary tree.

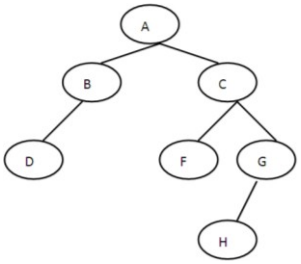


FIG: Binary tree

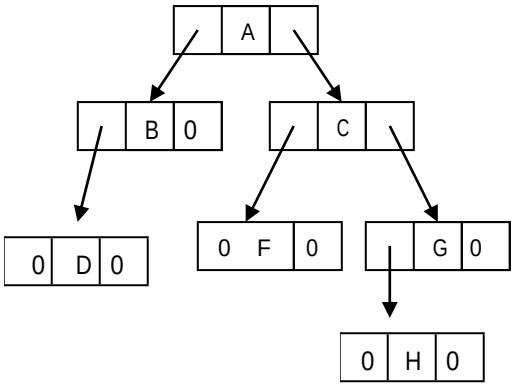


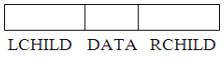
FIG: Linked representation of a binary tree

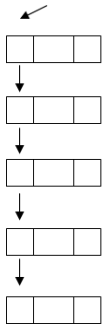
disadvantages

- Wasted memory space is well pointers.
- Given a node, it is difficult to determined to parent.
- Its implementation algorithm is more difficult in languages that do not offer dynamic storage techniques.

Linked Representation: -

The problems of sequential representation can be easily overcome through the use of a linked representation. Each node will have three fields LCHILD, DATA and RCHILD as represented below





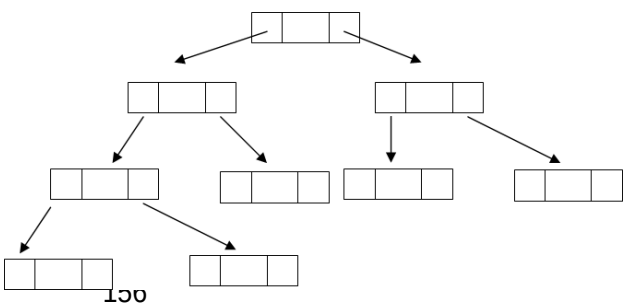


Fig (a) *Fig (b)*

In most applications it is adequate. But this structure make it difficult to determine the parent of a node since this leads only to the forward movement of the links.

Using the linked implementation,

we may declare, stuct nodetype

```
{
int info;
struct nodetype *left; struct nodetype *right; struct
nodetype *father;
};
typedef struct nodetype *NODEPTR;
```

This representation is called dynamic node representation. Under this representation,

info(p) would be implemented by reference p→info,

left(p) would be implemented by reference p→left,

right(p) would be implemented by reference p→right,

father(p) would be implemented by reference p→father.

Array Representation:

Each node contains info, left, right and father fields. The left, right and father fields of a node point to the node's left son, right son and father respectively.

Using the array implementation, we may declare,

```
#define NUMNODES 100 struct nodetype
{
int info; int left; int right; int father;
};
struct nodetype node[NUMNODES];
```

This representation is called linked array representation.

Under this representation,

info(p) would be implemented by reference node[p].info,

left(p) would be implemented by reference node[p].left,

right(p) would be implemented by reference node[p].right,

father(p) would be implemented by reference node[p].father respectively.

The operations,

isleft(p) can be implemented in terms of the operation left(p)

Example: -

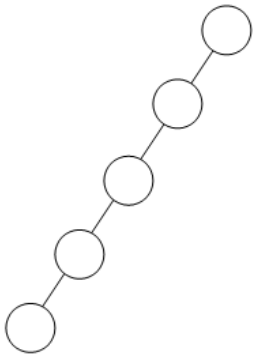


Fig (a)

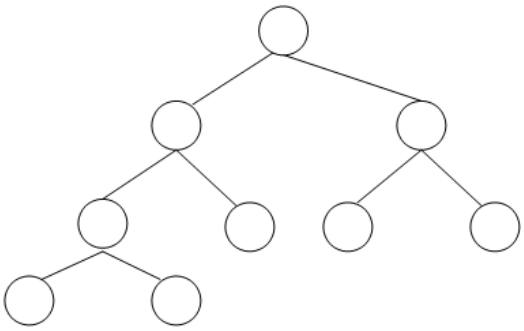
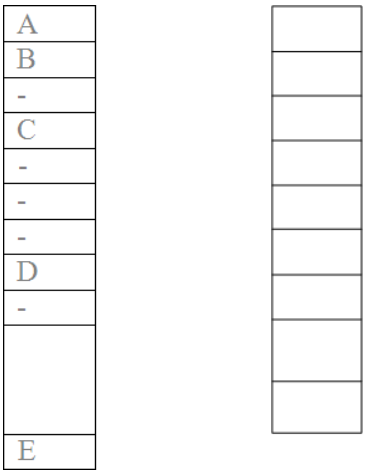


Fig (b)

isright(p) can be implemented in terms of the operation right(p)

The above trees can be represented in memory sequentially as follows



The above representation appears to be good for complete binary trees and wasteful for many other binary trees. In addition, the insertion or deletion of nodes from the middle of a tree requires the insertion of many nodes to reflect the change in level number of these nodes.

3. What is Binary Tree? Explain its operation in detail with example. (Nov 2012, April 2011, April 2012,April 2013)

Binary Tree:

A binary tree is a tree in which each node has atmost 2 childrens. (i.e) a node can have 0 child or 1 child or 2 child. A node must not have more than 2 childrens.

Eg:

BINARY SEARCH TREE OPERATIONS:

The basic operation on a binary search tree(BST) include, creating a BST, inserting an element into BST, deleting an element from BST, searching an element and printing element of BST in ascending

order.

The ADT specification of a BST:

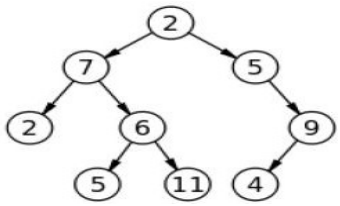
ADT BST

{

Create BST() : Create an empty BST;

Insert(elt) : Insert elt into the BST;

Search(elt,x) : Search for the presence of element elt and



Set x=elt, return true if elt is found, else return false.

FindMin() : Find minimum element;

FindMax() : Find maximum element;

Ordered Output() : Output elements of BST in ascending order; Delete(elt,x) : Delete elt and set x = elt;

}

Inserting an element into Binary Search Tree

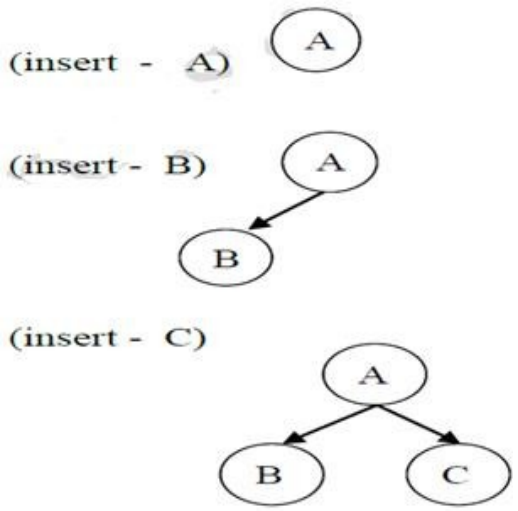
```
Algorithm InsertBST(int elt, NODE *T)
[ elt is the element to be inserted and T is the pointer to the root of the
 tree] If (T= = NULL) then
    Create a one-node tree and
return Else if (elt<key) then
    InsertBST(elt,
T→lchild) Else


    if(elt>key) then
    InsertBST(elt, T→rchild)
Else
    " element is
already exist return T
End
```

C coding to Insert element into a BST

```
struct node
    {
    int info;
    struct node *lchild; struct node *rchild;
    };
typedef stuct node NODE;
NODE *InsertBST(int elt, NODE *T)
    {
    if(T = = NULL)
      {
      T = (NODE *)malloc(sizeof(NODE)); if ( T = = NULL)
        printf ( " No memory error"); else
        {
        t→info = elt; t→lchild = NULL;
        t→rchild = NULL;
        }
      }
    else if ( elt < T→info)
      t→lchild = InsertBST(elt, t→lchild); else if ( elt > T→info)
      t→rchild = InsertBST( elt, t→rchild); return T;
    }
```

Inserting an External node:

The first node that we add is an external node'node-A'. To add a new node'node-B' after 'node-A', 'node-A' assigns the newly inserted node 'node-B' as one of its child and the added new node'node-B' assigns 'node-A' as its parent.



Searching an element in BST

Searching an element in BST is similar to insertion operation, but they only return the pointer to the node that contains the key value or if element is not, a NULL is return;

Searching start from the root of the tree;

If the search key value is less than that in root, then the search is left subtree;

If the search key value is greater than that in root, then the search is right subtree;

This searching should continue till the node with the search key value or null pointer(end of the branch) is reached.

In case null pointer(null left/righ chile) is reached, it is an indication of the absence of the node.

```
NODE * SearchBST(int elt, NODE *T)
{
if(T = = NULL)
  return NULL;

 if ( elt < T→info)
  return SearchBST(elt,
t→lchild); else if ( elt >
T→info)
  return SearchBST( elt, t→rchild); else
  return T;
}
```

Finding Minimum Element in a BST

Minimum element lies as the left most node in the left most branch starting from the root. To reach the node with minimum value, we need to traverse the tree from root along the left branch till we get a node with a null / empty leftsubtree.

Algorithm FindMin(NODE * T)

1. If Tree is null then return NULL;
2. if lchild(Tree) is null then return tree
   else
   return FindMin(T→lchild)
3. End

```
NODE * FindMin( NODE *T )
    {
    if(T = = NULL)
      return NULL;

     if(T→lchild==NULL) return tree;
    else
      return FindMin(Tree→lchild);
    }
```

Finding Maximum Element in a BST

Maximum element lies as the right most node in the right most branch starting from the root. To reach the node with maximum value, we need to traverse the tree from root along the right branch till we get a node with a null / empty rightsubtree.

Algorithm FindMax(NODE * T)

1. If Tree is null
   then return
   NULL;
2. if rchild(Tree) is null
   then return tree
   else
   return FindMax(T→rchild)
3. End

```
NODE * FindMax( NODE *T )
    {
    if(T = = NULL)
      return NULL;
     if(T→rchild==
      NULL) return
      tree;
    else
```

return FindMax(Tree→rchild); }

DELETING AN ELEMENT IN A BST

The node to be deleted can fall into any one of the following categories;

1. Node may not have any children (ie, it is a leafnode)
2. Node may have only on child (either left / right child)
3. Node may have two children (both left and

right) Algorithm DeleteBST(int elt, NODE * T)

1. If Tree is null then

print "Element is not found"

2. If elt is less than info(Tree) then

locate element in left subtree and

delete it else if elt is greater than

info(Tree) then locate element in

right subtree and delete it

else if (both left and right child are not NULL) then /* node with two

children */ begin

Locate minimum element in the right

subtree Replace elt by this value

Delete min element in right subtree and move the remaining tree as its right childend else

if leftsubtree is Null then

/* has only right subtree or both subtree

Null */ replace node by its rchild

else

if right subtree is Null then replace node by its left child\

end

free memory allocated to min node end

return Tree End

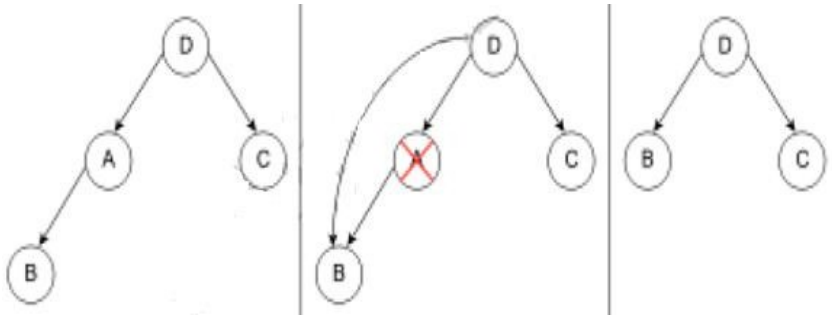
```
NODE *DeleteBST(int elt, NODE * T)
{
NODE * minElt;

if(T = = NULL)
    printf("element not found \n"); else if ( elt <T→info)
```

```
    T→lchild = DeleteBST(elt,
T→lchild); else if ( elt >T→info)
    T→rchild = DeleteBST(elt, T→rchild); else
    if(T→lchild && T→ rchild)
      {
      minElt = FindMin(T→rchild); T→info = minElt→info;
      T→rchild = DeleteBST(T→info, T→rchild);
      }
    else
      {
      minElt = Tree;
      if (T→lchild = = NULL) T = T→rchild;
      else if (T→rchild = = NULL) T = T→lchild;
      Free (minElt);
      }
return T;
}
```

Deletion:

Deletion is the process whereby a node is deleted from the tree. Only certain nodes in a binary tree can be deleted easily.

(i.e) the node with 0 or 1 children can be deleted, but, the node with 2 childrens cannot be deleted easily



4. Explain Tree Traversal techniques with its implementation and example. (April 2012) Traversals of a *Binary Tree:*

Traversing a tree means processing the tree such that each node is visited only one.

Traversal of a binary tree is useful in many applications. For example, in searching for particulars nodes compilers commonly build binary trees in the process of scanning, parsing, generating code and evaluation of arithmetic expression.

Let T be a binary tree, there are a number of a different ways to proceed. The methods differ

primarily in the order in which they visit the nodes. The three different traversals of T are Inorder, Post order and Preorder traversals.

In C, each node is defined as a structure of the following

```
form: struct node
  {
  int info;
  struct node
  *lchild; struct
  node *rchild;
  }
typedef struct node NODE;
```

I. *Inorder Traversal:*

It follows the general strategy of Left-Root-Right. In this traversal, if T is not empty, we first traverse (in order) the left sub-tree;

Then visit the root node of T and then traverse the right sub-tree. Consider the binary tree given in the following figure.

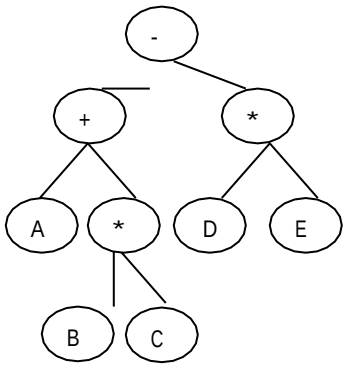


FIG: Expression Tree

This is an example of an expression tree for (A+B*C)-(D*E).

A binary tree can be used to represent arithmetic expressions if the node value can be either operators or operand values and are such that

- Each operator node has exactly two branches.
- Each operand node has no branches; such trees are called expression trees.

Tree, T at the start is rooted at '-';

- Since left(T) is not empty; current T becomes rooted at '+';
- Since left(T) is not empty; current T becomes rooted at 'A';
- Since left (T) is empty; we visit root i.e. A.
- We access T root i.e. '+'.

- We now perform in-order traversal of right (T) current T becomes rooted at '*'.
- Since left(T) is not empty; current T becomes rooted at 'B' since left(T) is empty; we visit it to root i.e. B;

cluck for right (T) which is empty, therefore we move back to parent tree. We visit its root i.e. '*'.

Now Inorder traversal of right (T) is performed; which would give us
'C'. We visit T's root i.e., 'D' and perform in-order traversal of right (T); which would give us '* and E'.
Therefore the complete listing is
A+B*C-D*E
We may note that expression is in infix notation. The in-order traversal produces a left expression then prints out the operator at root and then a right expression.

Steps :

1. Traverse left subtree in inorder
4. Process root node
5. Traverse right subtree in inorder

Algorithm

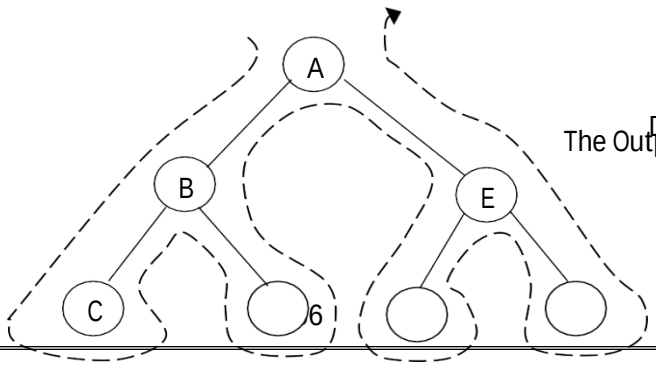
```
Algorithm inoder traversal (Bin-Tree T)
{
Begin
If ( not empty (T) ) then
Inorder_traversal ( left subtree ( T ) )
Print ( info ( T ) )  / * process node */
Inorder_traversal ( right subtree ( T ) )
E
```

C Coding

```
void inorder_traversal ( NODE * T)
{
if( T ! = NULL)
inorder_traversal(T->lchild);
        printf("%d \t", T->info); Begin
inorder_traversal(T->rchild);
}
}
```

II. *Postorder Traversal:*

In this traversal we first traverse Left(T) ; then traverse Right(T) ; and finally visit root. It is a Left-



The Output is : C → B → D →A →E → F

Right- Root strategy i.e.,

Traverse the left sub-tree in

Postorder Traverse the right sub-

tree in Postorder Visit the root

For example, a Postorder traversal of the fig: Expression tree

would be ABC*+DE*-

We can see that it is the prefix notation of the expression (A+(B*C))-

(D*E). Tree, T, at the start is rooted at '-' ;

- Since left (T) is not empty ; current T becomes rooted at '+';
- Since left (T) is not empty ; current T becomes rooted at 'A';
- Since left (T) is empty ; we visit right (T) '*'
- Since Right (T) is empty ; we visit root i.e. A
- We access right T's root it *
- Since left (T) is not empty ; we visit right (T)
- Since right (T) is empty we visit root i.e., B
- Since right (T) is not empty ; current T becomes rooted at 'C'
- Since left (T) & right (T) is empty , we visit root i.e. 'C'
- Visit root i.e. '*'
- Visit root i.e., '+'
- Since right (T) is not empty ; current T becomes rooted at '*'
- Since left (T) is not empty ; current T becomes rooted at 'D'
- Since left (T) and right (T) are empty we visit root i.e., 'D'
- Since right (T) is not empty ; current T becomes rooted at 'E'.
- Since left (T) & right (T) are empty ; we visit root i.e., 'E'
- Visit root is '*'
- Visit root is '-'

Therefore, the complete

listing is ABC*+DE*-

Steps : 1. Traverse left subtree in postorder

2. Traverse right subtree in postorder

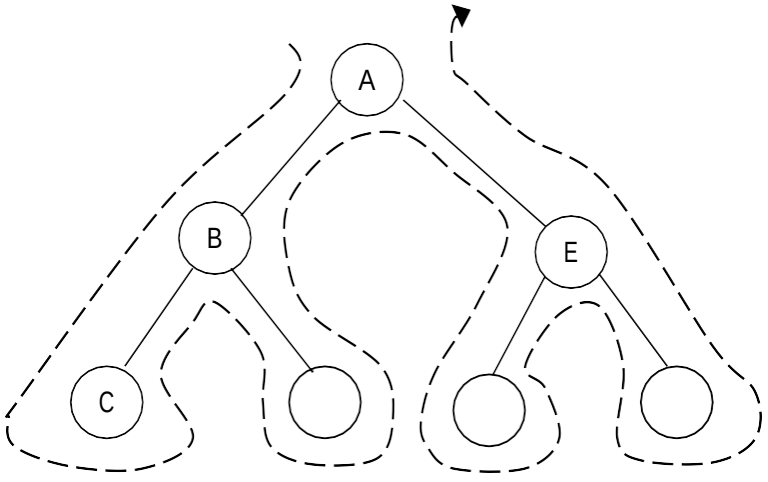
3. process root node

Algorithm

```
Postorder Traversal
                                        {
Algorithm postorder traversal (Bin-Tree T)
                                      {
Begin
If ( not empty (T) ) then
Begin
    Postorder_traversal ( left subtree ( T ) )
    Postorder_traversal ( right subtree( T))      }
    Print ( Info ( T ) ) / * process node */
End
End
```

C function

```
void postorder_traversal ( NODE * T)

  if( T ! = NULL)

  {
   postorder_traversal(T->lchild);
          postorder_traversal(T->rchile);
   printf("%d \t", T->info);
  }
```



The Output is : C → D → B → F → E → A

III. *PREORDER TRAVERSAL:*

In this traversal, we first visit the root; then traverse left (T) and finally right (T). It is a Root-Left-Right strategy i.e.

Visit the root

Traverse the left sub-tree in preorder

Traverse the right sub tree in preorder

For example, a preorder traversal of the fig: Expression tree would be

-+A*BC*DE

We can see that it is a prefix notation of the expression (A+(B*C))-(D*E)

Tree, T, at the start is rooted at '-';

- *since it visits the root first , it is displayed i.e. '-' is displayed*
- *Next left traversal is done. Here the current T becomes rooted at '+'*
- *Since left(T) is not empty; current T becomes rooted at A*
- *Since left (T) is empty we visit right (T) i.e. '*' and T is rooted here.*
- *Since left (T) is not empty , current T gets rooted at B*
- *Since left (T) is empty, we visit right (T) i.e. 'C' and T is at present rooted here*
- *Since left (T) is empty and no right (T) we move to the root '*'*
- *Visit root '+'*
- *Now the left sub tree is complete so we move to right sub tree i.e. T is rooted at '*'*
- *Since left (T) is not empty , current T becomes rooted at D*
- *Since left (T) is empty, T gets rooted at right (T) i.e. E.*

*Thus the complete listing is -+A*BC*DE*

Steps : 1. Process root node

2. Traverse left subtree in preorder
3. Traverse right subtree in preorder

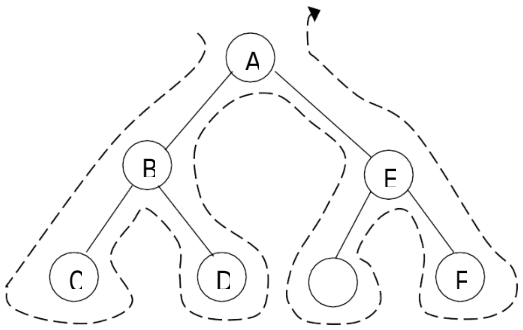
Algorithm

Algorithm preorder traversal (Bin-Tree T)

Begin
If (not empty (T)) then
Begin
Print (info (T)) / * process node * /
Preoder traversal (left subtree (T))
Inorder traversal (right subtree (T))
End
End

C function

```
void preorder_traversal ( NODE * T)
{
  if( T ! = NULL)
    {
  printf("%d \t", T->info);
          preorder_traversal(T->lchild);
  preorder_traversal(T->rchild);
     }
 }
```



Output is : A →B → C → D → E → F

5. Explain in detail about Application of Tress with examples. (Nov 2012)

I. Set Representation

In this section we study the use of trees in the representation of sets. We shall assume that the elements of the sets are the numbers 1, 2, 3...n. These numbers might, in practice, be indices into symbol tables where the actual names of the elements are stored. We shall assume that the sets being represented are pair wise disjoint: i.e. if Si and Sj, I != j, are two sets then there is no element which is in both Si and Sj. For example if we have 10 elements numbered from 1 through 10, they may be partitioned into three disjoint sets S1 = {1,7,8,9}; S2 = {2, 3, 10}; S3 = {3, 4,6}. The operations we wish to perform on these sets are

- Disjoint set union...if Si and Sj are two disjoint sets, then their union on Si U Sj
  = {all elements x such is in Si or Sj)

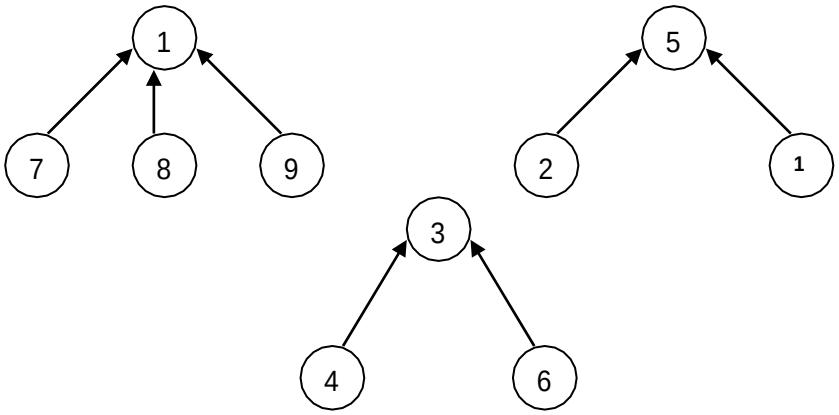
Thus,

S1 U S2 = {1, 7, 8, 9, 2, 5, 10}

Since we have assumed that all sets are disjoint. Following the union of Si and Sj we can assume that the sets Si and Sj no longer exist independently i.e., they are replaced by Si U Sj in the collection of sets.

- Find(i)...find the set containing element i. Thus, 4 is in set S3 and 9 is in set S1.

The sets will be represented by trees. One possible representation for the sets S1, S2 and S3 is is:



In presenting the UNION and FIND algorithms we shall ignore the actual set names and just identify sets by the roots of the trees representing them. This will simplify the discussion. The transition to set names is easy. If we determine that element I is in a tree with root j and I has a pointer to entry k in the set name table, then the set name is just(NAME(k)). If we wish to union sets Si and Sj then we wish to union the trees with roots POINTERS(Si) and POINTERS(Sj).

As we shall see in many application the set name is just the element at the root. The operation of FIND(i) now becomes; determine the root of the tree containing element i. UNION(I, j) requires two trees with roots I and j to be joined. We shall assume that the nodes in the trees are numbered 1 through n so that the node index corresponds to the element index. Thus element 6 is represented by the node with index 6. Consequently, each node needs only one field: The PARENT field to link to its parent. Root nodes have a PARENT field of zero. Based on the above discussion our first attempt arriving at UNION, FIND algorithms would result in the algorithms U and F below.

Algorithm for FIND / UNION operation

```
Procedure U(I, j)
    //replace the disjoint sets with roots I and j, i!=j with their union//
PARENT(i) ← j
End U

Procedure F(i)
    //Find the root j of the tree containing element i//
j← i
whilePARENT(j)>0do
      j← PARENT(j)
end
return(j)
end F
```

We can do much better if care is taken to avoid the creation of degenerate trees. In order to accomplish this we shall use a Weighting Rule for UNION(I, j). If the number of nodes in tree I is less than the number of nodes in tree j, then make j the parent of i, otherwise make i the parent of j. using this rule on the sequence of set unions given before we obtain the trees on next page remember that the arqument of UNION must both be roots. The time required to process all the n finds is only O(m) since in tis case the maximum level of any node is 2. This, however, is not the worst case.

The maximum level for any node is [log n]+1. First let us see how easy it is to implement the weighting rule. We need to know how many nodes there are in any tree. To do this easily, we maintain a count field in the toot of every tree. If I is a root node, then COUNT(i) = number of nodes in that tree. The count can be maintained in the PARENT field as a negative number. This is equivalent to using a one bit field to distinguish a count from a pointer. No confusion is created as for all other nodes the PARENT is positive.

```
Procedure UNION(i, j)
    //union sets with roots i and j1 != j using the weighting rule//
x ← PARENT(i) +PARENT(j)

if PARENT(i) > PARENT(j) then
    [PARENT(i) ← j
     PARENT(j) ← x]
Else
    [PARENT(j) ←1
     PARENT(i) ← x]
```

End UNION

Similarly modifying the FIND algorithm by using the Collapsing Rule:

If j is a node on the path from I to its root and PARENT(j) != root(i) then set PARENT(j) ← root(i). The new algorithm becomes:

```
Procedure FIND(i)
    //find the root of the tree containing element i//
j←i
while PARENT(j)>0 do
    j←PARENT(j)
end
k←i
while k != j do
    t ← PARENT(k)
    PARENT(k) ←j
    K←t
end
return(j)
end FIND
```

II. *Decision Trees*

Another application of trees is decision making. Consider the well-known eight coins problem. Given coins a,b,c,d,e,f,g,h. we are told that one is a counterfeit and has a different weight than the others. We want to do so using a minimum number of comparisons and at the same time determine whether the false coin is heavier or lighter than the rest. The tree below represents a set of decisions by which we can get the answer to our problem. This is why it is called a decision tree.

The use of capital H or L means that counterfeit coin is heavier or lighter. Let us trace through one possible sequence. If a+b+c < d+e+f, then we know that the false coin is present among the six and is neither g nor h. If on our next measurement we find that a+d < b+e, then by interchanging d and b we have is not the culprit, and (ii) that b or d is also not the culprit. If a+d was equal to b+e, the c or f would be the counterfeit coin. Knowing at this point that either a or e is the counterfeit, we compare a with a good coin say. If a = b then e is heavy, otherwise a must be light.

By looking at this tree we see that all possibilities are covered, since there are 8 coins which can be heavy or light and there are 16 terminal nodes. Every path requires exactly 3 comparisons.

ALGORITHM:

```
Procedure COMP(x,y,z)
    // x is compared against the standard coin z //
if x > z then print (x 'heavy')
else print (y 'light')
end COMP
Procedure EIGHTCOINS
    //eight weights are input; the different one is discovered using only 3 comparison//
read (a,b,c,d,e,f,g,h)
case
    : a+b+c = d+e+f : if g>h then call COMP(g,h,a)
            else call COMP(h,g,a)
    : a+b+c > d+e+f : case
                : a+d = b+e : call COMP(c,f,a)
                :a+d > b+e : call COMP(a,e,b)
                :a+d < b+e : call COMP (b,d,a)
            end
    : a+b+c < d+e+f : case
                : a+d = b+e : call COMP(f,c,a)
                : a+d > b+e : call COMP(d,b,a)
                : a+d < b+e : call COMP(e,a,b)
             end
  end
end EIGHTCOINS
```

III. *Game Tree*

A game tree is like a search tree in many ways ...

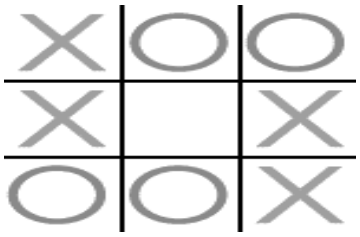
- Nodes are search states, with full details about a position
- characterize the arrangement of game pieces on the game board
- Edges between nodes correspond to moves
- leaf nodes correspond to a set of goals
- { win, lose, draw }
- Usually determined by a score for or against player
- at each node it is one or other player's turn to move
- A game tree is not like a search tree because you have an opponent!

Two-player games

The object of a search is to find a path from the starting state to a goal state

- In one-player games such as puzzle and logic problems you get to choose every move
- e.g. solving a maze
- In two-player games you alternate moves with another player
- competitive games
- each player has their own goal
- search technique must be different

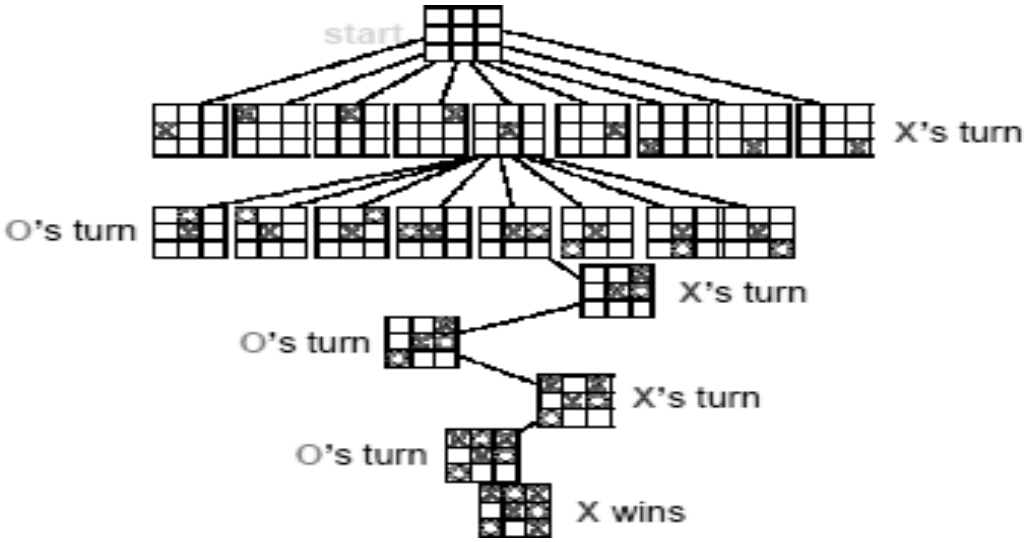
Eg. Tic-Tac-Toe



The first player is X and the second is O

- Object of game: get three of your symbol in a horizontal, vertical or diagonal row on a 3µ3 game board
- X always goes first
- Players alternate placing Xs and Os on the game board
- Game ends when a player has three in a row (a wins) or all nine squares are filled (a draw)

Partial game tree for Tic-Tac-Toe



6. Write the Linked list implementation of Polynomial addition.

Polynomials:

One classic example of an ordered list is a polynomial.

Definition:

A polynomial is the sum of terms where each term consists of variable, coefficient and exponent.

Various operations which can be performed on the polynomial are,

- Addition of two polynomials
- Multiplication of two polynomials
- Evaluation of polynomials

The Polynomial Expression can be represented using Linked list by the following illustration,

The typical node will look like this,

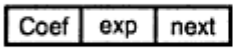
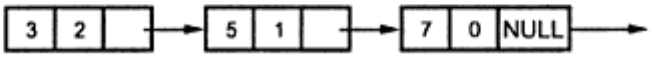


Fig. 3.13 Node of polynomial

For example : To represent $3x^2 + 5x + 7$ the link list will be,



Program for Addition of two polynomial using Linked List

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct list
{
    int coef;
    int pow;
    struct list *next;
};
struct list *p1=NULL,*p2=NULL,*p=NULL;

void generate(struct list *node)
{
    char c;
do
```

```
{
    printf("\n Enter a coefficient value:");
    scanf("%d",&node->coef);
    printf("\n Enter a power value:");
    scanf("%d", &node->pow);

    node->next=(struct list*)malloc(sizeof(struct list));
    node=node->next;
    node->next=NULL;

    printf("\n Want to continue? [Y/N]:");
    c=getch();
}
while(c=='y' || c=='Y');
}
void display(struct list *node)
{
    while(node->next!=NULL)
    {
        printf("%dx^%d",node->coef,node->pow);
        node=node->next;
        if(node->next!=NULL)
        {
                printf("+");
        }
    }
}
void add(struct list *p1,struct list *p2,struct list *p)
{
    while(p1->next && p2->next)
    {
        if(p1->pow > p2-> pow)
        {
                p-> pow = p1-> pow;
                p->coef= p1-> coef;
                p1= p1->next;
        }
        else if(p1->pow < p2->pow)
        {
                p->pow=p2->pow;
```

```
                p->coef=p2->coef;
                p2=p2->next;
        }
        else
        {
                p->pow=p1->pow;
                p->coef=p1->coef+p2->coef;
                p1=p1->next;
                p2=p2->next;
        }

        p->next=(struct list *)malloc(sizeof(struct list));
        p=p->next;
        p->next=NULL;
    }

    while(p1->next || p2->next)
    {
        if(p1->next)
        {
                p->pow=p1->pow;
                p->coef=p1->coef;
                p1=p1->next;
        }
        if(p2->next)
        {
                p->pow=p2->pow;
                p->coef=p2->coef;
                p2=p2->next;
        }
        p->next=(struct list *)malloc(sizeof(struct list));
        p=p->next;
        p->next=NULL;
    }
}
void main() // Main method
{
    char ch;
    clrscr();
    do
```

```
    {
        p1=(struct list *)malloc(sizeof(struct list));
        p2=(struct list *)malloc(sizeof(struct list));
        p=(struct list *)malloc(sizeof(struct list));

        printf("\n Enter First No:");
        generate (p1);
        printf("\n Enter second No:");
        generate (p2);

        printf("\n First No is:");
        display(p1);
        printf("\n Second No is:");
        display(p2);

        add(p1,p2,p);
        printf("\n Addition of the polynomial is:");
        display(p);

        getch();


    }
    while(ch=='y' || ch=='Y');

}
```

OUTPUT:

```
 Enter First No:
 Enter a coefficient value:3

 Enter a power value:3

 Want to continue? [Y/N]:
 Enter a coefficient value:
2

 Enter a power value:2
```

```
Want to continue? [Y/N]:
Enter a coefficient value:6

Enter a power value:1

Want to continue? [Y/N]:
Enter a coefficient value:7

Enter a power value:0


Want to continue? [Y/N]:
Enter second No:
Enter a coefficient value:2

Enter a power value:3

Want to continue? [Y/N]:
Enter a coefficient value:6

Enter a power value:2

Want to continue? [Y/N]:
Enter a coefficient value:7

Enter a power value:1

Want to continue? [Y/N]:
Enter a coefficient value:
4

Enter a power value:0

Want to continue? [Y/N]:n

First No is:          3x^3+2x^2+6x^1+7x^0
Second No is:         2x^3+6x^2+7x^1+4x^0
Addition of the polynomial is:    5x^3+8x^2+13x^1+11x^0
```

SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution & Accredited by NAAC with "A" Grade)
(An Autonomous Institution)
Madagadipet, Puducherry - 605 107

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject Name: **Data Structures**
Subject Code:

Prepared by:

Verified by: **Approved by:**

UNIT – V

Sorting, Hashing and Graphs

Sorting: Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Shell Sort and Radix Sort. Performance and Comparison among the sorting methods. Hashing: Hash Table, Hash Function and its characteristics. Graph: Basic Terminologies and Representations, Graph traversal algorithms

2 Marks

1. **Define Sorting**.

   A sorting algorithm is an algorithm that puts elements of a list either in ascending or descending order.

2. **What are the different types of sorting techniques? Give example.**

   There are 2 sorting techniques:

   - Internal sorting eg: sorting with main memory
   - External sorting eg: sorting with disk, tapes.

3. **Compare internal and external sorting**.

Internal sorting:

- It takes place in the main memory of a computer
- Internal sorting methods are applied for small collections of data (i.e) the entire collection of data to be sorted is small enough that the sorting can take place within main memory.

External sorting:

- External sorting methods are applied only when the number of data elements to be sorted is too large.
- These sorting methods involves as much external processing.

4. List out some of the sorting algorithms.

1. Insertion sort
2. Selection sort
3. Bubble sort
4. Shell sort
5. Merge sort
6. Heap sort
7. Quick sort
8. Radix Sort

5. Write down the limitations of Insertion sort.

- It is less efficient on list containing more number of elements.
- As the number of elements increases the performance of the program would be slow.
- Insertion sort needs a large number of element shifts.

6. What are the basic operations of Insertion sort?

We start with with empty list, „S" and unsorted list, „I" of „n" items. For (each item „X" of „I"

)

{

We are going to insert „X" into „S", in sorted order.

}

7. Name the sorting techniques which use Divide and Conquer strategy.

- Merge sort
- Quick sort

8. What is the best case and average case analysis for Quick sort?

- The total time in best case is : O(n logn)
- The total time in worst case is : $\Theta(n^2)$

9. What is the complexity of bubble sort?

- Best case performance: O(n)
- Average case performance: $O(n^2)$
- Worst case performance: $O(n^2)$

10. Compare bubble and Insertion sort.

- Even though both the bubble sort and insertion sort algorithms have average case time

Complexities of $O(n^2)$, bubble sort is outperformed by the insertion sort(i.e) insertion sort is faster than bubble sort.

- This is due to the number of swaps needed by the two algorithms (bubble sorts needs more swaps).
- But due to the simplicity of bubble sort, its code size is very small.
- insertion sort is very efficient for sorting "nearly sorted" lists, when compared with the bubble sort.

11. Explain the concept of merge sort.

- Divide the list in half
- Merger sort the first half
- Merge sort the second half
- Merge both halves back together in sorted order.

12. What is radix sort?

- The Radix Sort perform sorting, by using bucket sorting technique.
- In Radix Sort, first, bucket sort is performed by least significant digit, then next digit and so on. It is sometime known as Card Sort.

13. What is Bubble Sort and Quick sort?

Bubble Sort: The simplest sorting algorithm. It involves the sorting the list in a repetitive fashion. It compares two adjacent elements in the list, and swaps them if they are not in the designated order. It continues until there are no swaps needed. This is the signal for the list that is sorted. It is also called as comparison sort as it uses comparisons.

Quick Sort: The best sorting algorithm which implements the 'divide and conquer' concept. It first divides the list into two parts by picking an element a 'pivot'. It then arranges the elements those are smaller than pivot into one sub list and the elements those are greater than pivot into one sub list by keeping the pivot in its original place.

14. What is the main idea behind the selection sort

The idea of selection sort is rather simple which repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array.

15. What is the main idea in Bubble sort?

The basic idea underlying the bubble sort is to pass through the file sequentially several times. Each pass consists of comparing each element in the file with its successor (x[i] and x[i+1] and interchanging the two elements if they are not in proper order.

16. When can we use insertion sort?

Insertion sort is useful only for small files or very nearly sorted files.

17. What is the main idea behind insertion sort?

The main idea of insertion sort is to insert in the ith pass the ith element in A (1) A (2)...A (i) in its right place. An insertion sort is one that sorts a set of records by inserting records into an existing file.

18. Justify that the selection sort is diminishing increment sort.(Nov 2012)

Selection sort is diminishing increment sort. Because the Number of swapping in selection sort is better than Insertion sort.

19. Define the term sorting.

The term sorting means arranging the elements of the array so that they are placed in some relevant order which may either be ascending order or descending order.

That is, if A is an array then the elements of A are arranged in sorted order (ascending order) in such a way that, A [0] < A [1] < A [2] < …… < A [N].

20. What is sorting algorithm?

A sorting algorithm is defined as an algorithm that puts elements of a list in a certain order that can either be numerical order, lexicographical order or any user-defined order.

21. Where do we use external sorting?

External sorting is applied when there is huge data that cannot be stored in computer's memory.

22. Where do we use external sorting?

External sorting is applied when there is huge data that cannot be stored in computer's memory.

23. List out the different sorting techniques.

- Insertion sort
- Selection sort
- Shell sort
- Bubble sort
- Quick sort
- Merge sort
- Radix sort

24. What is the advantage of quick sort?

Quick sort reduces unnecessary swaps and moves an item to a greater distance, in one move.

25. Define Hashing.

Hashing is used for storing relatively large amounts of data in a table called a hash table. Hashing is a technique used to perform insertions, deletions, and finds the element in **constant average time**

26. What do you mean by hash table?

Hash Table is a data structure in which keys are mapped to array positions by a hash function. Hash table is usually fixed as M-size, which is larger than the amount of data that we want to store.

27. Define hash function.

Hash function is a mathematical formula, produces an integer which can be used as an index for the key in the hash table.

- **Perfect Hash Function**

  Each key is transformed into a unique storage location

- **Imperfect hash Function**

Maps more than one key to the same storage location .

28. What are the different methods of hash function?

- Division Method
- Multiplication Method
- Mid Square Method
- Folding Method

29. Define Division Method

Division method is the most simple method of hashing an integer *x.* The method divides *x* by *M* and then use the remainder thus obtained.

In this case, the hash function can be given as

$$h(x) = x \bmod M$$

30. When collision does occur?

Collision occurs when the hash function maps two different keys to same location.

31. What are the various collision resolution techniques used?

Two major classes of collision resolution

Open Addressing

- When collision occurs, use organized method to find next open space.
- Maximum number of elements equal to table size.

Chained Addressing – Separate Chaining

- Make linked list of all element that hash to same location
- Allows number of elements to exceed table size

32. What are the methods used in Open addressing collision technique?

- Linear Probing
- Quadratic Probing
- Double Hashing

33. What is mean by rehashing?

If the table gets too full, the insertion might fail .A solution is to build another table that is almost twice as big & scan down the entire original table into new table.

34. When do we perform rehashing?

- Rehash as soon as table is half full.
- Rehash when insertion fails
- When table reaches certain load factor
- Performance degrades as the load factor increases

35. When do we use extendible hashing?

When the amount of data is too large to fit in main memory, the previous hashing techniques will not work properly. So we use a new technique called extensible **hashing**. It is one form of dynamic hashing.

15,25,5,40,2,70,18 – Insertion sort

5,15,30,6,3,95- Merge sort

36. What is radix sort?

- The Radix Sort perform sorting, by using bucket sorting technique.
- In Radix Sort, the numbers are sorted on the least significant digit first, followed by secondleast
  significant digit and so on till the most significant digit. It is sometime called as Card Sort.

37. What is bubble sort?

- Bubble sort is otherwise called as sinking sorts.
- The idea of bubble sort is to move the highest element to n**th** position.
- The principle of bubble sort is to scan or read the array in (n-1) times.
- It compares two adjacent elements in the list and swaps them if they are not in the designated order. It continues until there are no swaps needed.
- It is also called as comparison sort, as it uses comparisons.

38. What is quick sort?

- The best sorting algorithm which implements the ‘divide and conquer’ concept.
- It first divides the list into two parts by picking an element a ’pivot’.
- Then arranges the elements those are smaller than pivot into one sub list and the elements those are greater than pivot into one sub list by keeping the pivot in its original place.

- It is also called as partition exchange sort.

39. What is selection sort ? Why selection sort is better than insertion sort?

The idea of selection sort is rather simple which repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array.

40. What is insertion sort ?

- In insertion sort the elements are inserted at an appropriate place similar to card insertion.
- The elements in the list is divided into two parts- sorted and unsorted sub-lists.
- In each pass, the first element of unsorted sub-list is picked up and moved into the sorted sub-list by inserting it in suitable positon.

41. What is shell sort?

The shell sort improves upon bubble sort and insertion sort, by moving out of order elements more than one position at a time.

It compares the elements that are at a specific distance from each other, and interchanges them if necessary. The shell sort divides the list into smaller sub lists, and then sorts the sub lists separately using the insertion sort

42. What is Selection sort?

Selection sort is sorted by scanning the entire list to find the smallest element and exchange it with the first element, putting the first element in the final position in the sorted list. Then the scan starts from the second element to find the smallest among n-1 elements and exchange it with the second element.

43. What is Merge sort?

- It follows divide and conquer method for its operation.
- In Dividing phase, the problem is divided into smaller problem and solved recursively.
- In conquering phase, the partitioned array is merged together recursively.
- Merge sort is applied to first half and second half of the array.
- It gives two sorted halves which can then be recursively merged together using the merging algorithm.

44. What is the performance analysis of merge sort?

- Best case Performance: O(n log n)
- Average Case Performance: O(n log n)
- Worst case performance: O(n log n)

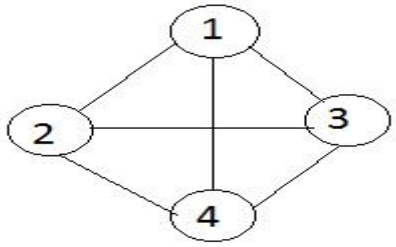
45. Define graph

o A graph consists of two sets V, E.

o V is a finite and non empty set of vertices.

o E is a set of pair of vertices; each pair is called an edge. o V(G),E(G) represents set of vertices ,set of edges .

G= (V,E)



46. Define digraph (Nov 13)

If an edge between any two nodes in a graph is directionally oriented, a graph is called as directed .it is also referred as digraph.

47. Define undirected graph.

If an edge between any two nodes in a graph is not directionally oriented, a graph is called as undirected .it is also referred as unqualified graph.

48. Define path in a graph

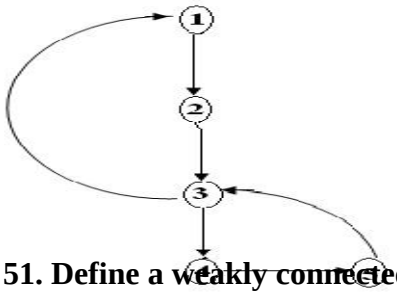
A path in a graph is defined as a sequence of distinct vertices each adjacent to the next except possibly the first vertex and last vertex is different.

49. Define a cycle in a graph

A cycle is a path containing atleast three vertices such that the starting and the ending vertices are the same. The cycles are (1, 2, 3, 1), (1, 2, 3, 4, 5, 3, 1)

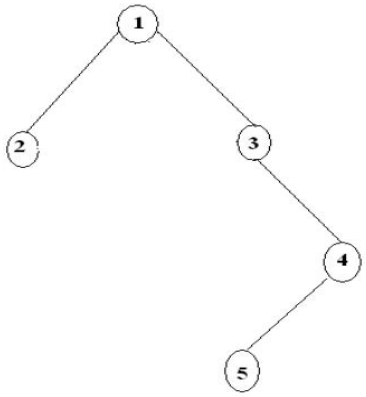
50. Define a strongly connected graph (Nov 14)

A graph is said to be a strongly connected graph, if for every pair of distinct vertices there is a directed path from every vertex to every other vertex. It is also referred as a complete graph.



51. Define a weakly connected graph. (May 14)

A directed graph is said to be a weakly connected graph if any vertex doesn't have a directed path to any other vertices.



52. Define a weighted graph.

A graph is said to be a weighted graph if every edge in the graph is assigned some weight or value. The weight of an edge is a positive value that may be representing the distance between the vertices or the weights of the edges along the path.

-

53. How we can represent the graph?

We can represent the graph by three ways

1.adjacent matrix

2.adjacent list

3. adjacent multi list

54.Define adjacency matrix

Adjacency matrix is a representation used to represent a graph with zeros and ones.

A graph containing n vertices can be represented by a matrix with n rows and n columns.

The matrix is formed by storing 1 n its i^{th} and j^{th} column of the matrix, if there exists an edge between ith and jth vertex of the graph and 0 if there is no edge between i^{th} and j^{th} vertex of the graph .

Adjacency matrix is also referred as incidence matrix.

55. What is meant by traversing a graph? State the different ways of traversing a graph. (Nov 12)

In undirected graph ,G=(V,E)

The vertex V in V(G)

To visit all the vertices that are reached from

the vertex V, that is all the vertices are
connected to vertex V

There are two types of graph traversals

1) Depth first search

2) Breadth first search

56. Define depth first search?

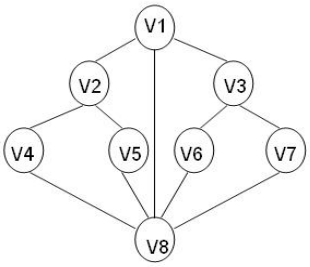
In DFS, we don't have any special vertex, we can start from any vertex.

Let us start from vertex (v), its adjacent vertex is selected and DFS is initialized.

Let us consider the adjacent vertices to V are V1, V2, and V3 …Vk.

Now pick V1 and visit its adjacent vertices then pick V2 and visit the adjacent vertices .to continue and process till the nodes are visited.

Consider the graph



V1	V2, V3, V8
V2	V4, V5
V4	V2, V8
V8	V4, V5, V1, V6, V7
V5	V2, V8
V6	V3, V8
V3	V6, V7, V1
V7	V8, V3

57. Define breadth first search?

Is BFS ,an adjacent vertex is selected then visit its adjacent vertices then backtrack the unvisited adjacent vertex

In BFS ,to visit all the vertices of the start vertex ,then visit the unvisited vertices to those adjacent vertices

58. Define topological sort?

A directed graph *G* in which the vertices represent tasks or activities and the edges represent activities to move from one event to another then the task is known as *activity on vertex network* or AOV-network/pert network

5 Marks

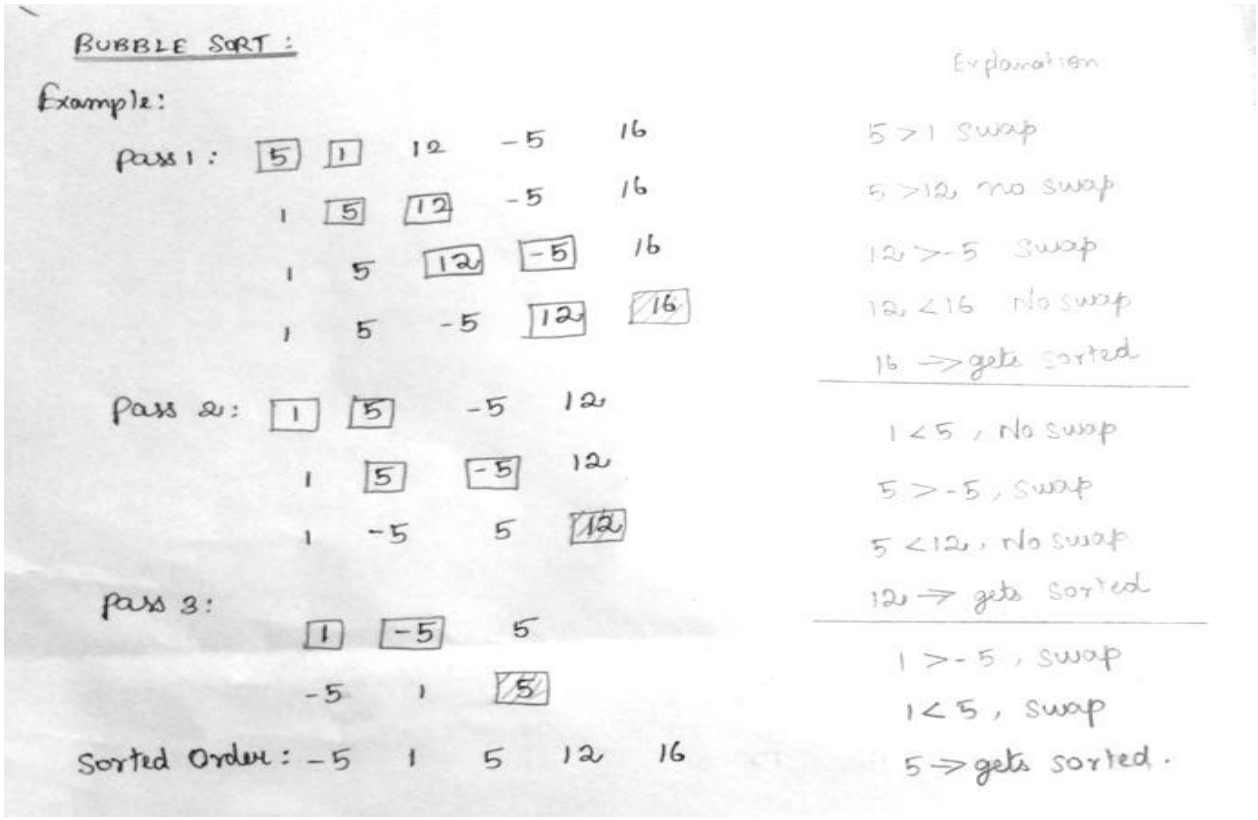
SORTING

1. BUBBLE SORT

1. The easiest and the most widely used sorting technique among students and engineers is the bubble sort.
2. ***This sort is also referred as sinking sort. The idea of bubble sort is to repeatedly move the smallest element to the lowest index position in the list.***
3. To find the smallest element, the bubble sort algorithm begins by comparing the first element of the list with its next element, and upto the end of the list and interchanges the two elements if they are not in proper order.
4. In either case, after such a pass the smaller element will be in the lowest index position of the list.
5. The focus then moves to the next smaller element, and the process is repeated. Swapping occurs only among successive elements in the list and hence only one element will be placed in its sorted order each pass.

```
BUBBLE (ARR,N)

    where ARR is an arrary of N elements

  Step 1.   REPEAT FOR I=0,1,2.....N-1
  Step 2.   REPEAT FOR J=I+1 TO N-1
  Step 3.   IF(ARR[I]>ARR[J] THEN INTERCHANGE ARR[I] AND ARR[J]
              (END OF IF STRUCTURE)
  Step 4.   INCREMENT J BY1
  Step 5.   (END OF STEP 2 FOR LOOP)
  Step 6.   (END OF STEP 1 FOR LOOP)
  Step 7.   PRINT THE SORTED ARRAY ARR

END BUBBLE()
```

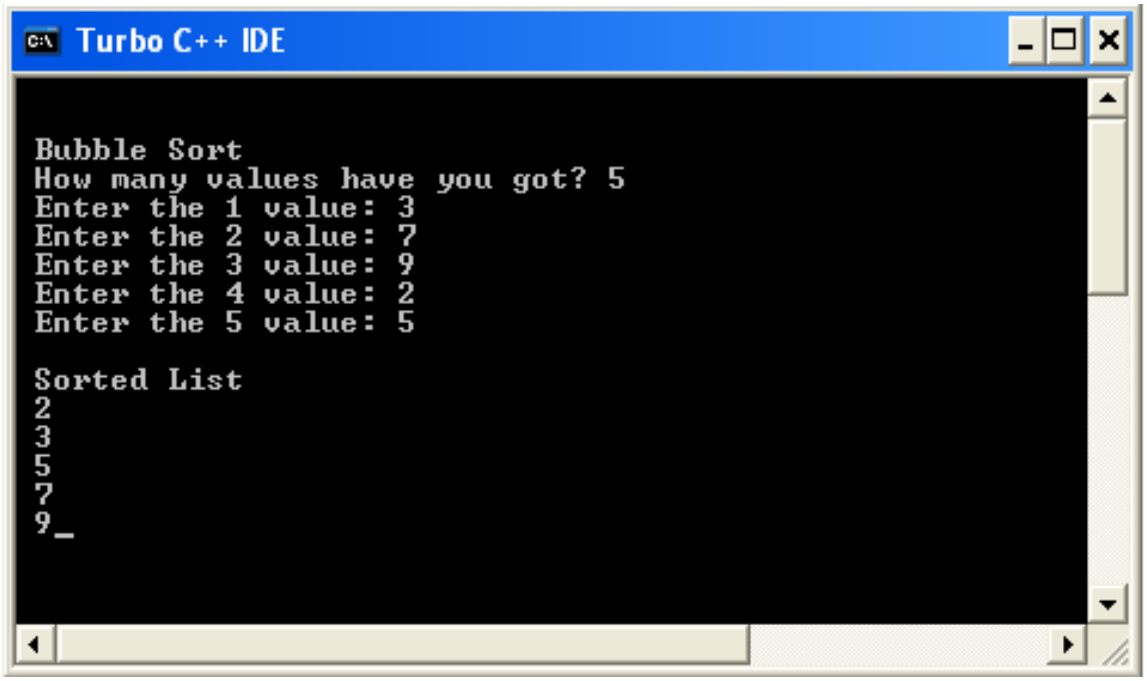
Example :-




```
//Bubble Sort
#include<stdio.h>
#include<conio.h>
int main()
{
  int i,j,temp,n,a[200];
  clrscr();
  printf("\n\n Bubble Sort");
  printf("\n How many values have you got? ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
        {
```

```
        printf(" Enter the %d value: ",i+1);
        scanf("%d",&a[i]);
      }
 printf("\n Sorted List");
 for(i=0;i<n;i++)
     {
          for(j=0;j<n-1;j++)
               {
                    if(a[j]>a[j+1])
                         {
                              temp=a[j];
                              a[j]=a[j+1];
                              a[j+1]=temp;
                         }
               }
      }
 for(i=0;i<n;i++)
   printf("\n %d",a[i]);
 getch();
 return 0;   }
```

OUTPUT:

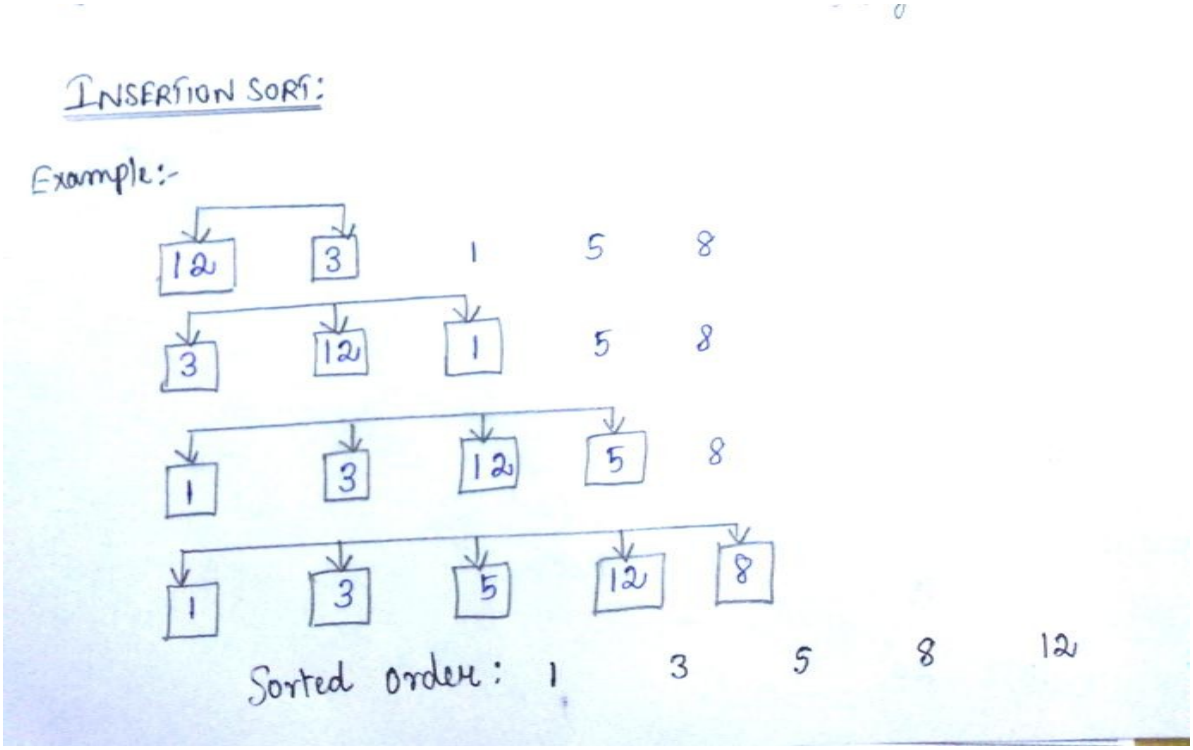


2. INSERTION SORT

1. The main idea of insertion sort is to consider each element at a time, into the appropriate position relative to the sequence of previously ordered elements, such that the resulting sequence is also ordered.
2. The insertion sort can be easily understood if you know to play cards. Imagine that you are arranging cards after it has been distributed before you in front of the table.
3. As each new cards is taken, it is compared with the cards in hand. The card is inserted in proper place within the cards in hand, by pushing one position to the left or right. This procedure proceeds until all the cards are placed in the hand are in order.

Algorithm

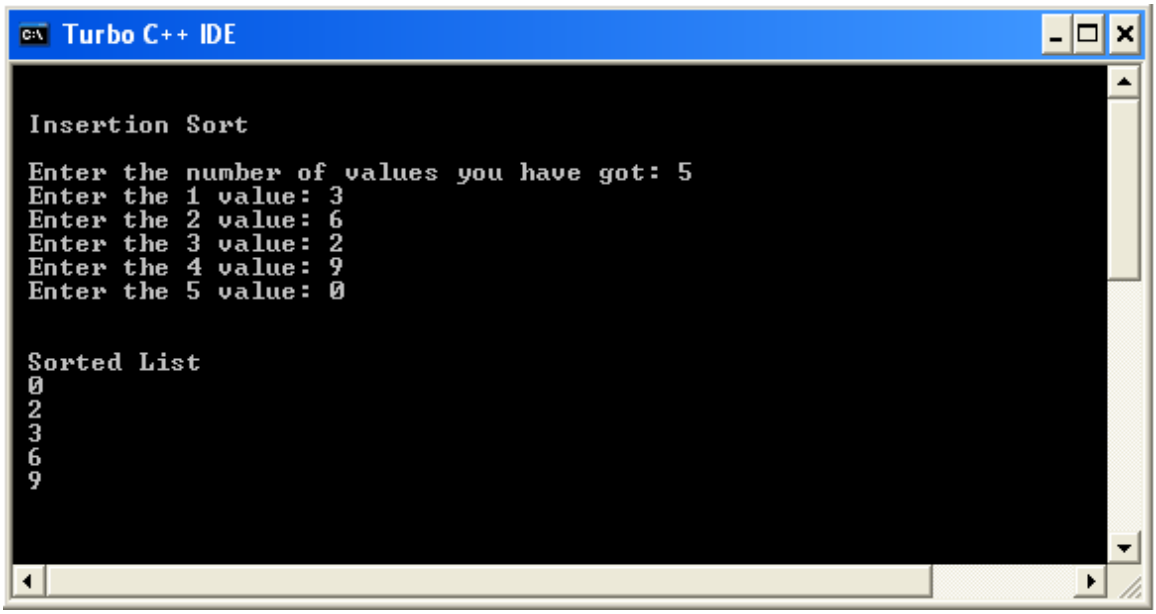
```
INSERT(ARR,N)

        Where ARR is an array of N elements

    Step 1. REPEAT FOR I=1,2,3..N+1
    Step 2. ASSIGN  TEMP=ARR[I] Step
    3.   REPEAT FOR J=I TO 1 Step 4.
    IF(TEMP < ARR[J-1] THEN
            ARR[J] = ARR[J-1]
            ELSE
            GOTO STEP7
            (END OF IF STRUCTURE)
    Step 5. DECREMENT J BY 1
    Step 6. (END OF STEP 3 FOR LOOP)
    Step 7.   ASSIGN ARR[J] = TEMP Step
    8. (END OF STEP 1 FOR LOOP)
    Step 9. PRINT THE SORTED ARRAY ARR
END INSERT()
```



// **INSERTION SORT**

```
#include<stdio.h>
#include<conio.h>
int main()
{          int i,j,key,a[200],n;
          clrscr();
          printf("\n\n Insertion Sort");
          printf("\n\n Enter the number of values you have got: ");
          scanf("%d",&n);
          for(i=0;i<n;i++)
                {
                     printf(" Enter the %d value: ",i+1);
                    scanf("%d",&a[i]);
                }
          printf("\n\n Sorted List");
          for(i=1;i<n;i++)
              {
                    key=a[i];
                    j=i-1;
                    while((j>=0)&&(key<a[j]))
                                {
                                     a[j+1]=a[j];
                                      j=j-1;
                                }
                      a[j+1]=key;
              }
          for(i=0;i<n;i++)
                 {
                      printf("\n %d",a[i]);
                 }
         getch();
        return 0;
}
```

OUTPUT:

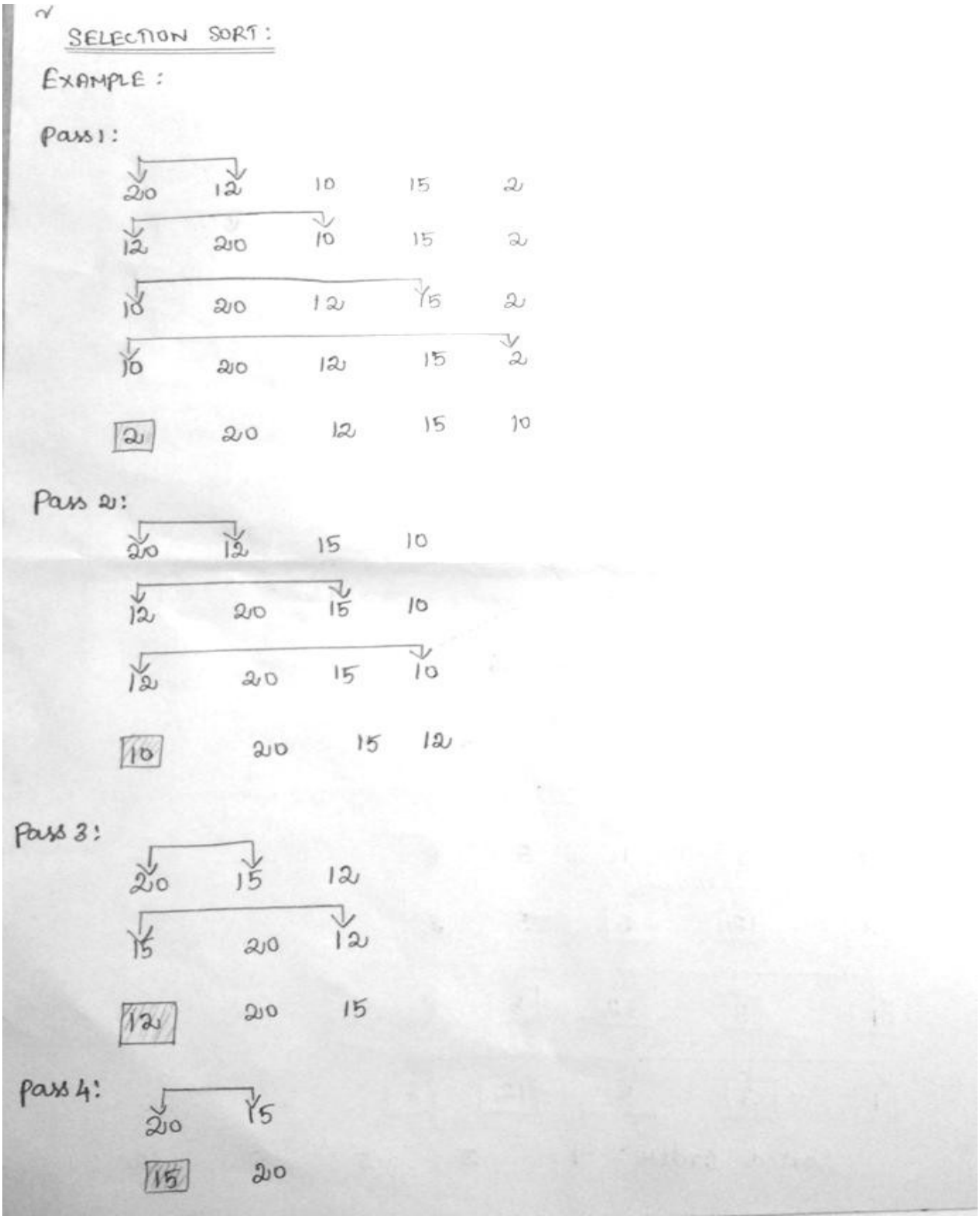


3. SELECTION SORT

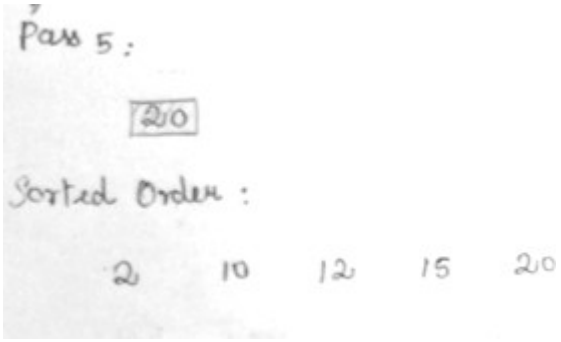
1. Another easiest method for sorting elements in the list is the selection sort. The main idea of selection sort is to search for the smallest element in the list.
2. When the element is found, it is swapped with the first element in the list. The second smallest element in the list is then searched.
3. When the element is found, it is swapped with the second element in the list.
4. The process of searching the next smallest element is repeated, until all the elements in the list have been sorted in ascending order.

Algorithm

```
SELECT(ARR,N)

        WHERE ARR IS AN ARRAY OF N ELEMENTS
        1. REPEAT FOR I=1,2,3...N-1
        2. ASSIGN K=1, MIN =ARR[I]
        3. REPEAT FOR J=I+1 TO N-1
        4. IF ARR[J] <MIN THEN

             MIN=ARR[J]

             K=J

             (END OF IF STRUCTURE)

        5. (END OF STEP 3 FOR LOOP)
        6. ASSIGN ARR[K]=ARR[I], ARR[I]=MIN
        7. (END OF STEP 1 FOR LOOP)
        8. PRINT THE SORTED ARRARY ARR
END SELECT()
```


SELECTION SORT:
EXAMPLE:
Pass 1:
20 12 10 15 2
12 20 10 15 2
10 20 12 15 2
10 20 12 15 2
2 20 12 15 10
Pass 2:
20 12 15 10
12 20 15 10
12 20 15 10
10 20 15 12
Pass 3:
20 15 12
15 20 12
12 20 15
pass 4:
20 15
15 20

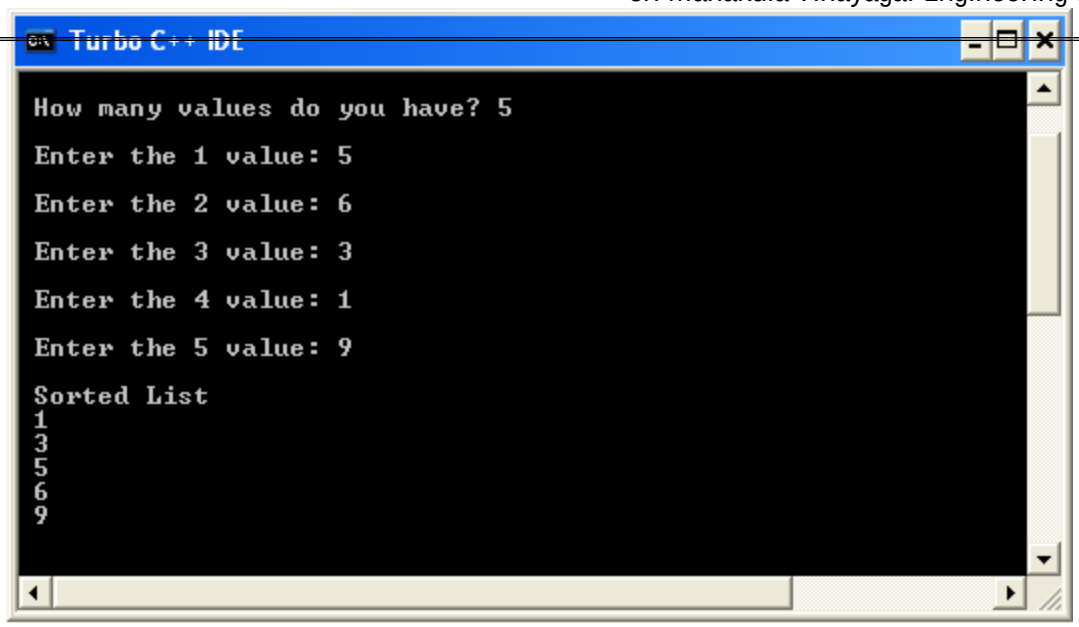
Example :




```
// Selection Sort
#include<stdio.h>
#include<conio.h>
int main()
{
  int i,j,k,min,n,a[200];
  clrscr();
  printf("\n\n Selection Sort \n ");
  printf("\n How many values do you have? ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
        {
          printf("\n Enter the %d value: ",i+1);
          scanf("%d",&a[i]);
        }
  printf("\n Sorted List");
  for(i=0;i<n;i++)
         {
          k=i;
```

```
        min=a[i];


for(j=i+1;j<n;j++)
                {
                    if(a[j]<min)
                            {
                              min=a[j];
                                k=j;
                            } }
             a[k]=a[i];
            a[i]=min;


            printf("\n %d",a[i]);
         }
  getch();
  return 0;
}
```

OUTPUT:



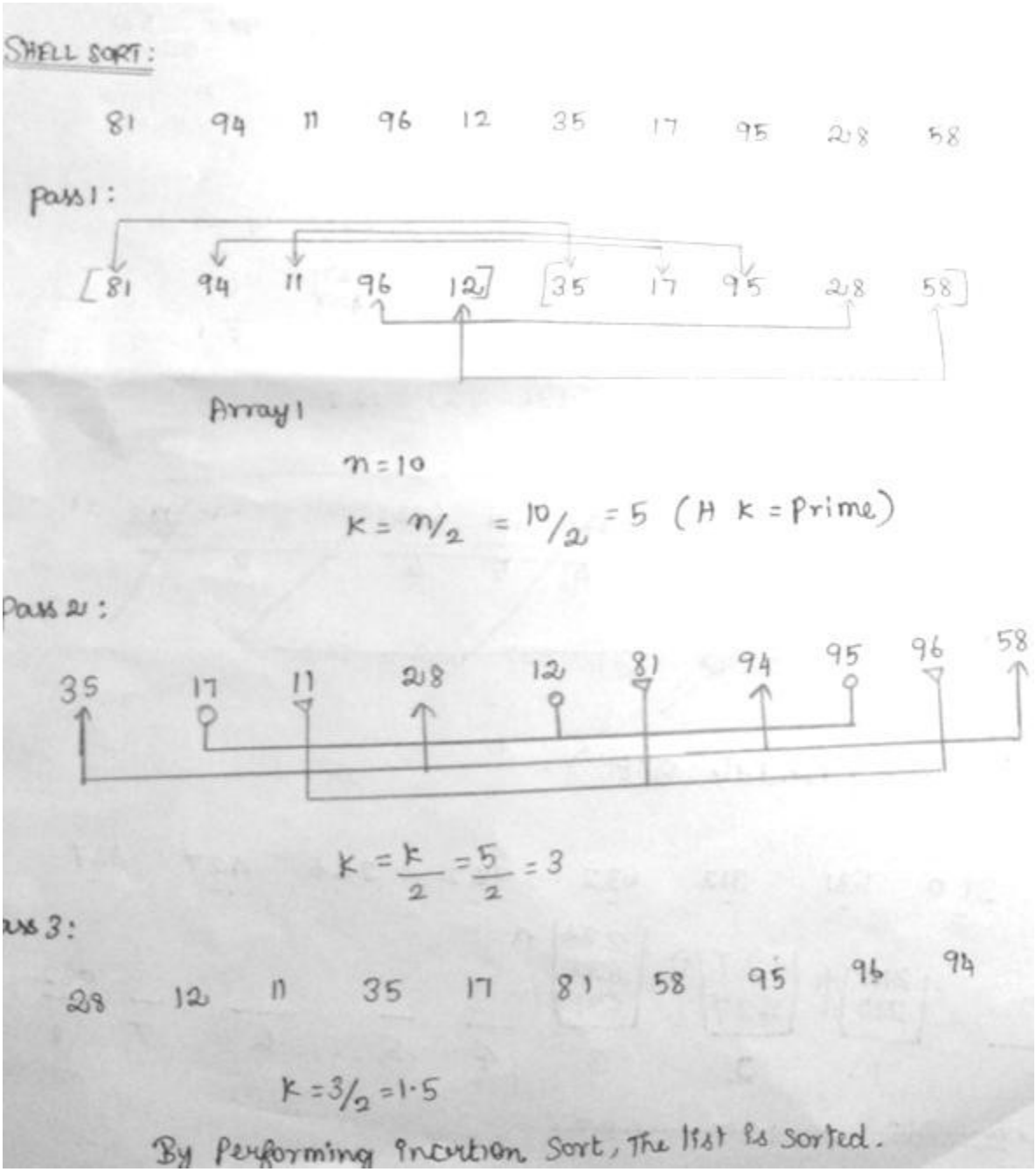
4.SHELL SORT

1. The shell sort , named after its developer Donald.L.shell in 1959
2. This sort is an extension of the insertion sort, which has the limitation, that it compares only the consecutive elements and interchange the elements by only one space.
3. The smaller elements that are far away require many passes through the sort, to properly insert them in its correct position.
4. The shell sort overcomes this limitation, gains speed than insertion sort, by comparing elements that are at a specific distance from each other, and interchanges them if necessary.
5. The shell sort divides the list into smaller sub lists, and then sorts the sub lists separately using the insertion sort. This is done by considering the input list being n-sorted.
6. This method splits the input list into h-independent sorted files. The procedure of h-sort is insertion sort considering only the h th element (starting anywhere).
7. The value of h will be initially high and its repeatedly decremented until it reaches 1. When h is equal to 1, a regular insertion sort is performed on the list, but by then the list of data is guaranteed to be almost sorted.
8. Using the above procedure for any sequence values of h, always ending in 1 will produce a sorted list.

Algorithm :-

```
SHELL(ARR,N)

    WHERE ARR IS AN ARRAY OF N ELEMENTS.

    1.  REPEAT FOR I=(N+1)/2 TO 1
    2.  REPEAT FOR J=I TO N-1
    3.  ASSIGN TEMP=ARR[J] , K=J-1
    4.  REPEAT WHILE (K>=0 AND TEMP < ARR[K])
    5.  ASSIGN ARR[K+1] = ARR[K], K=K-1
    6.  (END OF STEP4 WHILE LOOP)
    7.  ASSIGN ARR[K+1] =TEMP
    8.  INCREMENT J BY 1
    9.  (END OF STEP 2 FOR LOOP)
    10. ASSIGN I = I/2
    11. (END OF STEP1 FOR LOOP)
    12. PRINT THE SORTED ARRAY ARR.
```

Example :-



IX

Ans:- 11 12 17 28 35 58 81 94 95 96

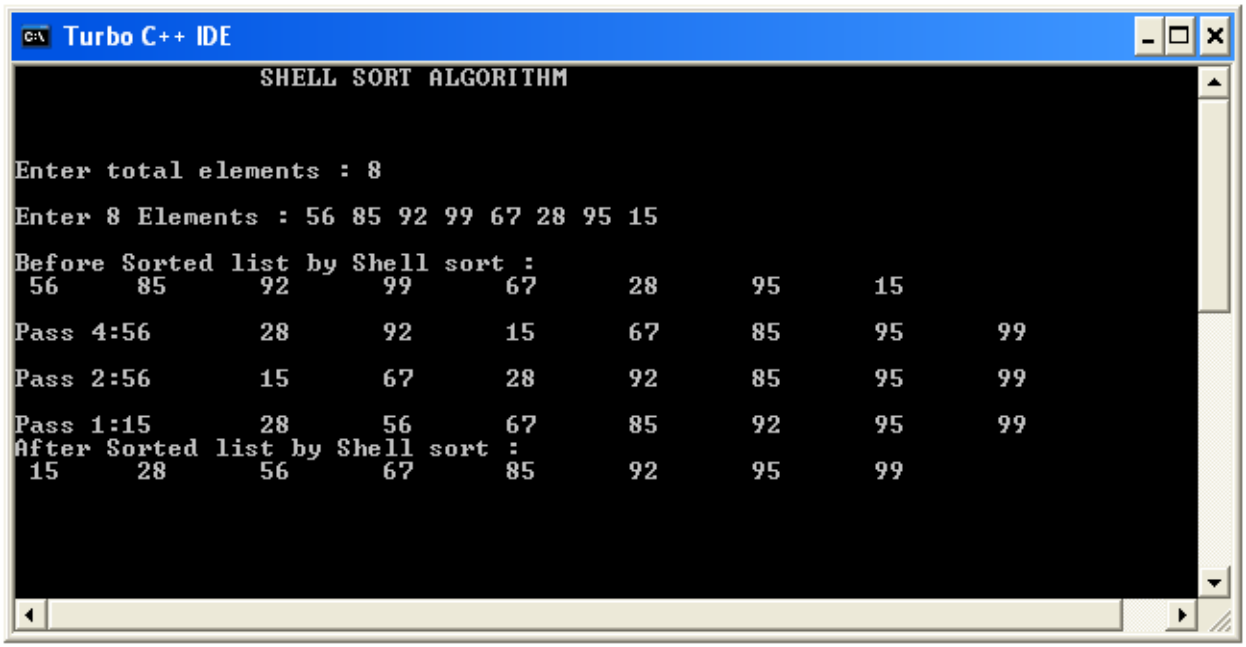
//Shell Sort

```
#include<stdio.h>
#include<conio.h>
void shellsort(int a[],int n)
{
  int j,i,k,m,mid;
  for(m = n/2;m>0;m/=2)
  {
        for(j = m;j< n;j++)
        {
            for(i=j-m;i>=0;i-=m)
            {
                if(a[i+m]>=a[i])
                        break;
                else
                {
                 mid = a[i];
                 a[i] = a[i+m];
                 a[i+m] = mid;
                }
            }
         }
    printf("\n\nPass %d:",m);
```

```
    for(k=0;k<n;k++)
        printf("%d\t",a[k]);
  }
}
int main()
{
   int a[20],i,n;
   clrscr();
   printf("\t\tSHELL SORT ALGORITHM\n\n\n\n");
   printf("Enter total elements : ");
   scanf("%d", &n);
   printf("\nEnter %d Elements : ", n);
   for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
   printf("\nBefore Sorted list by Shell sort :\n ");
   for(i=0;i< n;i++)
   printf("%d\t",a[i]);
   shellsort(a,n);
   printf("\nAfter Sorted list by Shell sort :\n ");
   for(i=0;i< n;i++)
   printf("%d\t",a[i]);
   getch();
   return 0;
```

}

OUTPUT


Turbo C++ IDE
SHELL SORT ALGORITHM
Enter total elements : 8
Enter 8 Elements : 56 85 92 99 67 28 95 15
Before Sorted list by Shell sort :
56 85 92 99 67 28 95 15
Pass 4:56 28 92 15 67 85 95 99
Pass 2:56 15 67 28 92 85 95 99
Pass 1:15 28 56 67 85 92 95 99
After Sorted list by Shell sort :
15 28 56 67 85 92 95 99

5.RADIX SORT

The sorting techniques that we have seen so far sorts the list of elements by comparing the sequence of elements and swaps them if necessary. The radix sort also referred, as the **bucket sort** is little bit different. It manages to sort values without actually performing any comparisons on the input data. The values are successively ordered on digit positions from right to left (i.e., lower order byte to higher order byte). This is accomplished by copying the values into buckets, where the index is given by the position of the digit being sorted. Once all digit positions have been examined, the list must be sorted.

Radix is just a position in a number. In hexadecimal representation of a decimal number, a radix indicates a digit. Radix sort gets its name from the radices, because the method first sorts the input values according to their first radix, then according to the second and so on. The number of passes in the radix sort equals the number of radices in the input values. For example you will need 2 passes to sort 16-bit integers and 4 passes to sort 32-bit integers. In a hexadecimal number system, radix is referred as a byte, and hence the radix sort is often referred as a **byte sort**. Radix sort uses 255 buckets for placing 1 byte data from 00 to FF.

Algorithm for radix sort

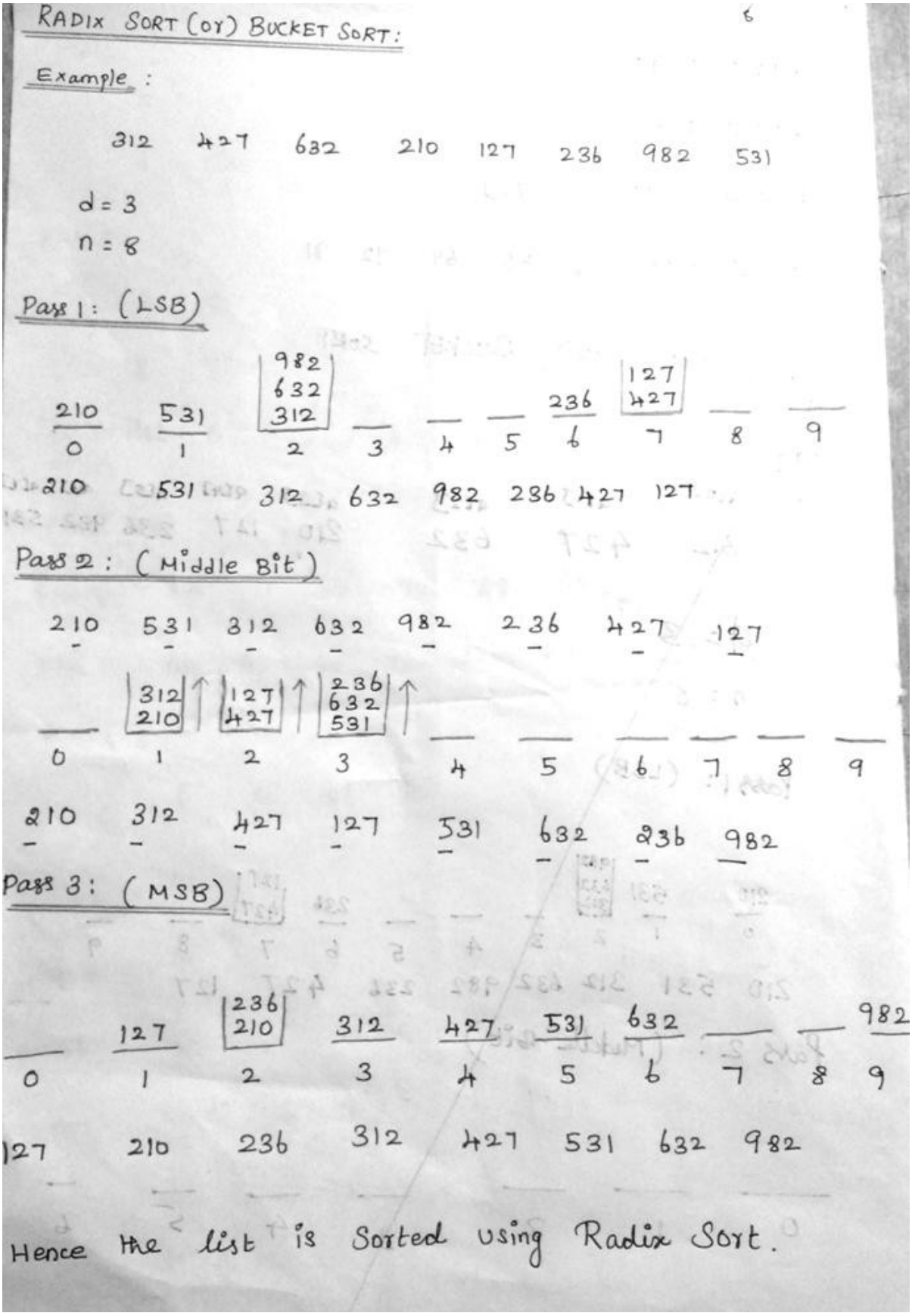
RADIX (ARR, N)

where ARR is an array of N elements

STEP 1 : Repeat For I = 1, 2, 3, . . ., 255
STEP 2 : Initialize bucket_size[I] = 0
STEP 3 : [End of **STEP 1** For loop]
STEP 4 : Repeat For M = 1 to 4 where M refers to the byte position (from lower order to higher order)
STEP 5 : Assign DATA1 = Mth lower order byte of ARR[J], bucket_size[DATA1] = bucket_size[DATA1]+1, TEMP[J] = ARR[J]
STEP 6 : [End of **STEP 4** For loop]
STEP 7 : Assign bucket_size[0] = 0
STEP 8 : Repeat For K = 1, 2, 3, . . ., 255
STEP 9 : Assign FIRST_IN_BUCKET[K] = FIRST_IN_BUCKET[K–1] + BUCKET_SIZE[K–1]
STEP 10 : [End of **STEP 1** For loop]

STEP 11 : Repeat For R = 1, 2, . . ., N–1
STEP 12 : Assign DATA2 = Mth lower order byte of TEMP[R]
ARR[FIRST_IN_BUCKET[DATA2]] = TEMP[R]
FIRST_IN_BUCKET[DATA2] = FIRST_IN_BUCKET[DATA2]+1
STEP 13 : [End of **STEP 11** For loop]
STEP 14 : Print the sorted array ARR
END RADIX()

Example :-

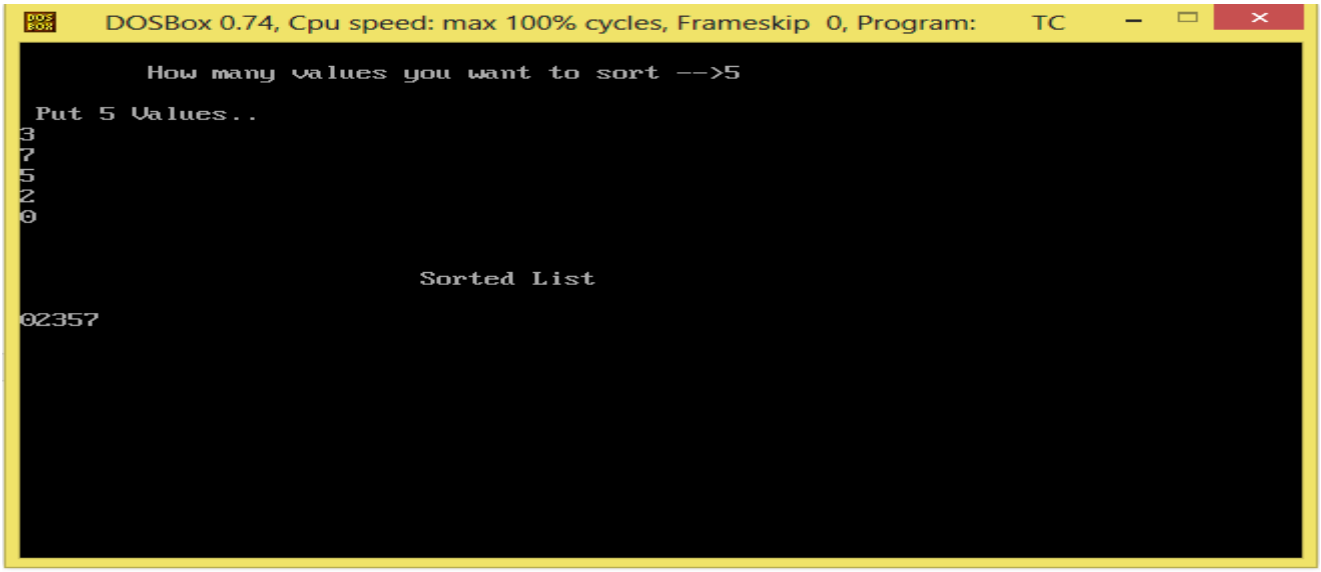

RADIX SORT (or) BUCKET SORT:
Example :
312 427 632 210 127 236 982 531
d = 3
n = 8
Pass 1: (LSB)
210 531 982 632 312 236 127 427
0 1 2 3 4 5 6 7 8 9
210 531 312 632 982 236 427 127
Pass 2: (Middle Bit)
210 531 312 632 982 236 427 127
312 210 127 427 236 632 531
0 1 2 3 4 5 6 7 8 9
210 312 427 127 531 632 236 982
Pass 3: (MSB)
127 236 210 312 427 531 632 982
0 1 2 3 4 5 6 7 8 9
127 210 236 312 427 531 632 982
Hence the list is Sorted using Radix Sort.

// Radix Sort

```
#include <stdio.h>
#include <conio.h>
int MAX = 100;
int A[100];
int C[100][10]; // Two dimensional array, where each row has ten pockets
int N;
void get_Values()
        {
                int i=0;
                printf("\n\tHow many values you want to sort -->");
                scanf("%d",&N);
                if(N>MAX)
                        {
                                printf("\n\n \t Maximum values Limits %d so try again...",MAX);
                                get_Values(); // recursive call
                        }
                printf("\n Put %d Values..\n",N);
                for(i=0 ; i< N ; i++)
                scanf("%d",&A[i]);
        }
void RadixSort (int R)
        {
                int i,j,X,D,m,p;
                int Z=0;
                D=R/10;
                for(i=0;i<N;i++)
                        {
                            for(j=0 ; j < 10 ; j++) C[i][j]=-1;
                        }
        for(i=0; i<N ; i++)
                        {
                                X=A[i]%R;
                                m=X/D;
                                if(m>0)Z=1;
                                        C[i][m]=A[i];
                        }
                p=-1;
                for(j=0 ; j < 10 ; j++)
                        {
                                for( i=0 ; i < N ; i++)
                                    {
                                        if(C[i][j]!=-1)
                                                {
                                                        p++;
                                                        A[p]=C[i][j];
```

```
                                                    }
                                        }
                                }
                        if(Z==1)



                                {
                                        R*=10;
                                       RadixSort(R); // recursive call
                                }
        } // end of RadixSort( ) function


void display()
        {
                int i;
                printf("\n\n\t\t\t Sorted List\n\n");
                for(i=0; i<N ; i++)
                printf("%d",A[i]);
          }

void main()
        {
                clrscr();
                get_Values();
                if(N>1) RadixSort (10); // Here 10 is the Radix of Decimal Numbersl
                        display();
                getch();
        }
```

OUTPUT



6.Explain various hashing techniques with suitable example.

Separate chaining:

Chaining

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

For example:

Consider the keys to be placed in their home buckets are

131, 3, 4, 21, 61, 24, 7, 97, 8, 9

Then we will apply a hash function as

H(key) = keytoD

where D is the size of table. The hash table will be: Here D = 10.

h(131)=131%10 =1

h(3)=3%10=3

h(4)=4%10=4

h(21)=21%10=1

h(61)=61%10=1

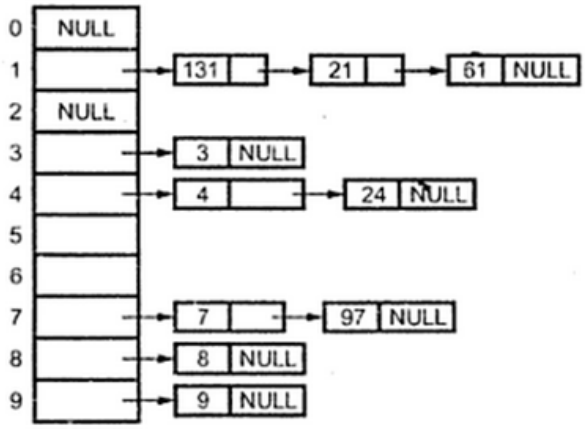
h(24)=24%10=4

h(7)=7%10=7

h(97)=97%10=7

h(8)=8%10=8

h(9)=9%10=9



A chain is maintained for colliding elements. For instance 131 has a home bucket (key) 1. Similarly keys 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1. Similarly the chain at index 4 and 7 is maintained.

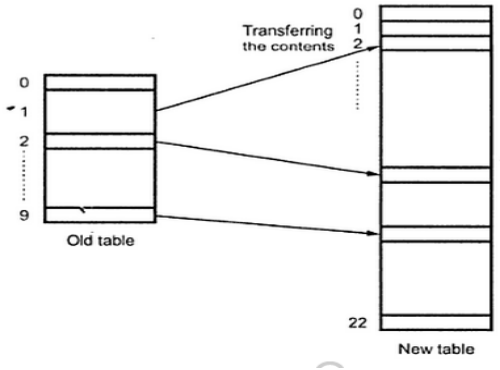
7. Explain in detail about Rehashing

Rehashing

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable if the total size of table is a prime number. There are situations in which the rehashing is required

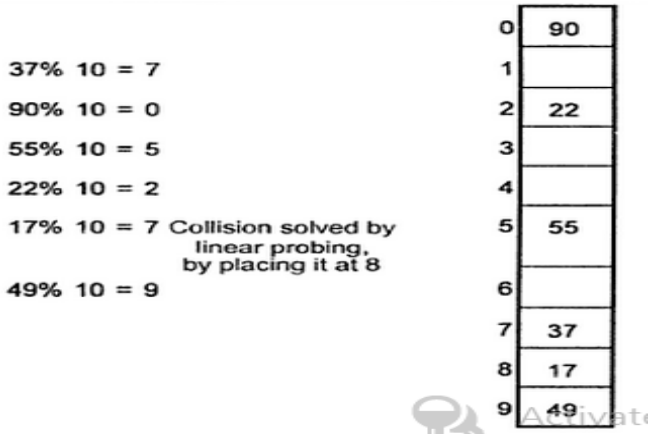
- When table is completely full.
- With quadratic probing when the table is filled half.
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re-computing their positions using suitable hash functions.



Consider we have to insert the elements 37, 90, 55, 22, 17, 49 and 87. The table size is 10 and will use hash function,

H(key) = key mod tablesize



Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size.

The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be

H(key) = key mod 23

		0
		1
		2
	49	3
		4
		5
		6
		7
		8
	55	9
		10
		11
		12
		13
37% 23 = 14	37	14
90% 23 = 21		15
		16
55% 23 = 9	17	17
22% 23 = 22	87	18
17% 23 = 17		19
		20
49% 23 = 3	90	21
87% 23 = 18	22	22

Rehashing Example

8. Explain about Extensible hashing.

Extensible hashing:

- When the amount of data is too large to fit in main memory, the previous hashing techniques will not work properly.
- So we use a new technique called **extensible hashing**. It is one form of dynamic hashing.

Here the

- Keys are stored in buckets.
- Each bucket can only hold a fixed size of keys.
- Consider initially the directory splitting (g) as 2 so the directory is 00,01,10,11.

Example:

Suppose that g=2 and bucket size = 4.

Suppose that we have records with these keys and hash function

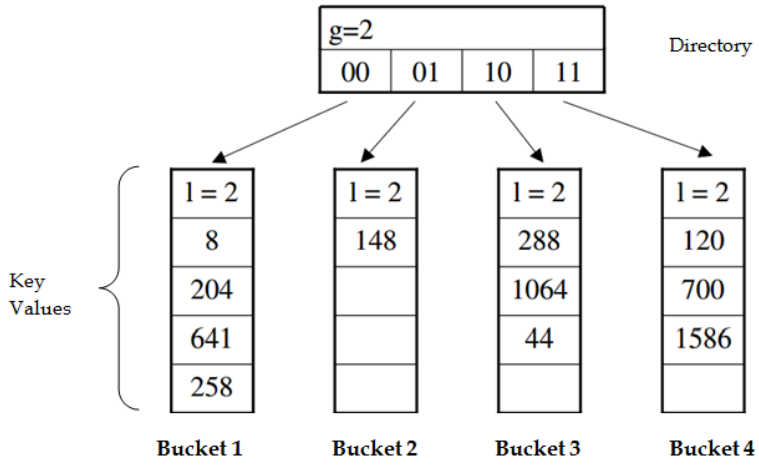
h(key) = key mod 64

the value 64 is obtained by 2^4 X4=16x4=64

key	h(key) = key mod 64	bit pattern
288	32	100000
8	8	001000
1064	40	101000
120	56	111000
148	20	010100
204	12	001100
641	1	000001
700	60	111100
258	2	000010
1586	50	110010
44	44	101010

Calculating bit pattern of each key

Check the first 2 bits(prefix bits) of the bit pattern and then place the key in the corresponding directory's bucket.



Extendible Hashing Example –directory and bucket structure

10-Marks

1. Briefly explain the three common collision strategies in open addressing hashing.

(OR)

Explain Linear Probing with an example.

Collision Resolution by Open Addressing

- Once a collision takes place, open addressing computes new positions using a probe sequence and the next record is stored in that position.
- In this technique of collision resolution, all the values are stored in the hash table.
- The process of examining memory locations in the hash table is called **probing.**
- Open addressing technique can be implemented using:
    - Linear probing
    - Quadratic probing
    - Double hashing.

Linear Probing

The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at location generated by *h(k),* then the following hash function is used to resolve the collision.

$$h(k, i) = [h'(k) + i] \bmod m$$

where, *m* is the size of the hash table, $h'(k) = k \bmod m$ and *i* is the probe number and varies from *0* to *m-1*.

Example: Consider a hash table with size = 10. Using linear probing insert the keys 72, 27, 36, 24, 63, 81 and 92 into the table.

Let h'(k) = k mod m, m = 10

Initially the hash table can be given as,

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Step1: Key = 72

h(72, 0) = (72 mod 10 + 0) mod 10

= (2) mod 10

= 2

Since, T[2] is vacant, insert key 72 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step2: Key = 27

h(27, 0) = (27 mod 10 + 0) mod 10

= (7) mod 10

= 7

Since, T[7] is vacant, insert key 27 at this location

Step3: Key = 36

h(36, 0) = (36 mod 10 + 0) mod 10

= (6) mod 10

= 6

Since, T[6] is vacant, insert key 36 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step4: Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10

= (4) mod 10

= 4

Since, T[4] is vacant, insert key 24 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step5: Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10

= (3) mod 10

= 3

Since, T[3] is vacant, insert key 63 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step6: Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10

= (1) mod 10

= 1

Since, T[1] is vacant, insert key 81 at this location

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step7: Key = 92

h(92, 0) = (92 mod 10 + 0) mod 10

= (2) mod 10

= 2

Now, T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for next location. Thus probe, i = 1, this time.

Key = 92

h(92, 1) = (92 mod 10 + 1) mod 10

= (2 + 1) mod 10

= 3

Now, T[3] is occupied, so we cannot store the key 92 in T[3]. Therefore, try again for next location. Thus probe, i = 2, this time.

Key = 92

h(92, 2) = (92 mod 10 + 2) mod 10

= (2 + 2) mod 10

= 4

Now, T[4] is occupied, so we cannot store the key 92 in T[4]. Therefore, try again for next location. Thus probe, i = 3, this time.

Key = 92

h(92, 3) = (92 mod 10 + 3) mod 10

= (2 + 3) mod 10

= 5

Since, T[5] is vacant, insert key 92 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	92	36	27	-1	-1

Quadratic probing

- It eliminates the primary clustering problems of linear probing

  $f(i)=i^2$

- If quadratic probing is used and the table size is prime , then a new element can always be inserted if the table is at least half empty.
- If the table is even one more than half full, the insertion could fail [prime].

{250, 567, 252, 763, 424}

	Empty table	After 250	567	252	763	424
0		250	250	250	250	250
1						
2			567	567	567	567
3				252	252	252
4					763	763

When 424 collides with 763, the next position attempted in one cell away. But another collision occurs.
$1^2, 2^2, 3^2, \ldots$

Double Hashing:

The double hashing is performed by

$$F(i)= i.hash_2(x)$$

here , $hash_2(x) = R-(x \bmod R)$ with R is a prime smaller than table size.

Example:

Let us consider following key values 89, 18, 49, 58,69

For first value apply the normal hash function i.e **key mod tablesize**

hash(89)=89%10=9

hash(18)=18%10=8

Now the key values 89 and 18 are stored at the corresponding location ,when for inserting the 3rd element 49

hash(49)=49%10=9 , collision occurs

so apply double hashing technique and Choose "R" a prime number smaller than table size, so we choose R=7 then

$hash_2(49)$=7-(49%7)= 7 - 0 =7

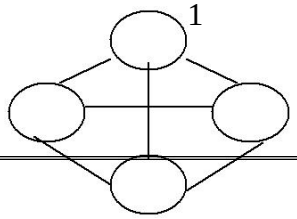
$hash_2(58)$=7-(58%7)= 7 – 2 =5

$hash_2(69)$=7-(69%7)= 7 – 6 = 1

	Empty table	89	18	49	58	69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

2. What is a graph? Write short notes on its basic terminologies. DEFINING GRAPH

- A graphs G consists of a set V of vertices (nodes) and a set E of edges (arcs).
- We write G=(V,E). V is a finite and non-empty set of vertices.
- E is a set of pair of vertices; these pairs are called as edges .
- Therefore (G).read as V of G, is a set of vertices and E(G),read as E of G is a set of edges.
- An edge e= (v, w) is a pair of vertices v and w, and to be incident with v and w.
- **A graph can be pictorially represented as follows,**



2 3

4

FIG: Graph G

- We have numbered the graph as 1,2,3,4.
- Therefore, V(G)=(1,2,3,4) and E(G) = {(1,2),(1,3),(1,4),(2,3),(2,4)}.

BASIC TERMINOLGIES OF GRAPH UNDIRECTED GRAPH

An undirected graph is that in which, the pair of vertices representing the edges is unordered.

DIRECTED GRAPH

- ***A directed graph is that in which, each edge is an ordered pair of vertices, (i.e.) each edge is represented by a directed pair.***
- ***It is also referred to as digraph.***

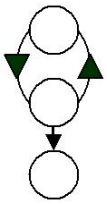


Fig.: Directed Graph

COMPLETE GRAPH

A vertex undirected graph with exactly n(n-1)/2 edges is said to be complete graph.

SUBGRAPH

A sub-graph of G is a graph G‘ such that V(G’) follow,

V(G) and E(G ‘) E(G). Some of the sub-graphs are as

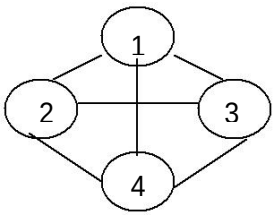
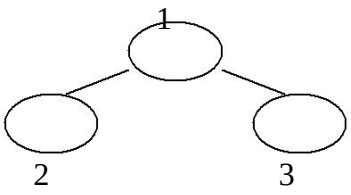
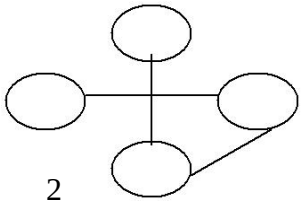


FIG: Graph G



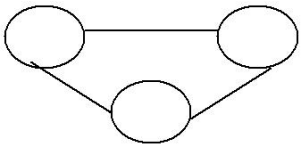
(a) (b)



2 3

FIG: Sub graphs of G (c) (d)

4



ADJACENT VERTIC

A vertex v1 is said to be a adjacent vertex if there exist an edge (v1,v2) or (v2,v1).

PATH:

A path from vertex V to the vertex W is a sequence of vertices, each adjacent to the next. The length of the graph is the number of edges in it.

CONNECTED GRAPH

A graph is said to be connected if there exist a path from any vertex to another vertex.

UNCONNECTED GRAPH

A graph is said to be an unconnected graph if there exist any two unconnected components.

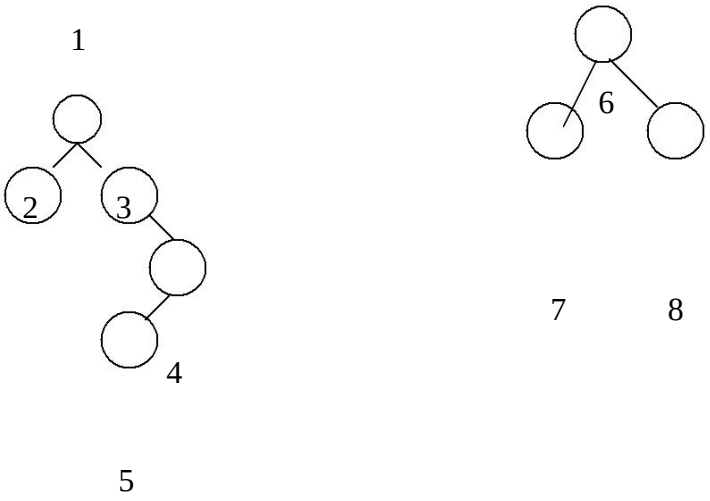
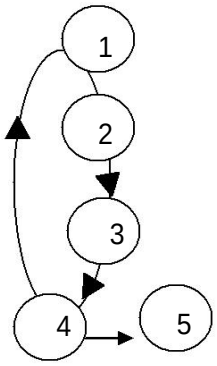


FIG: Unconnected Graph

STRONGLY CONNECTED GRAPH

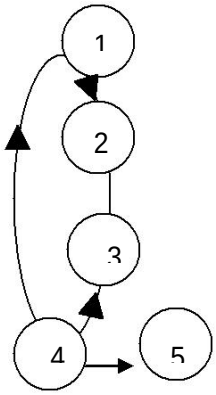
A digraph is said to be strongly connected if there is a directed path from any vertex to any other vertex

.



WEAKLY CONNECTED GRAPH

If there does not exist a directed path from one vertex to another vertex then it is said to be a weakly connected graph.



CYCLE

A cycle is a path in which the first and the last vertices are the same.

Eg. 1,3,4,7,2,1

DEGREE:

The number of edges incident on a vertex determines its degree. There are two types of degrees In-degree and out-degree.

- IN-DEGREE of the vertex V is the number of edges for which vertex V is a head.
- OUT-DEGREE is the number of edges for which vertex is a tail.

A GRAPH IS SAID TO BE A TREE, IF IT SATISFIES THE TWO PROPERTIES:

1. It is connected
2. There are no cycles in the graph.

GRAPH REPRESENTATION:

The graphs can be represented by the follow three methods,

1. Adjacency matrix.
2. Adjacency list.
3. Adjacency multi-list.

ADJACENCY MATRIX:

The adjacency matrix A for a graph G = (V,E) with n vertices, is an n* n matrix of bits ,such that A Ij = 1 , if there is an edge from vi to vj and

Aij = 0, if there is no such edge. The adjacency matrix for the graph G is,

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

The space required to represent a graph using its adjacency matrix is n* n bits. About half this space can be saved in case of an undirected graph, by storing only the upper or lower triangle of the matrix. From the adjacency matrix, one may readily determine if there an edge connecting any two vertices I and j. for an undirected graph the degree of any vertices I is its row $\sum^{n} (j=1) A (I, j)$. For a directed graph the row is the out-degree and column sum is the in-degree.

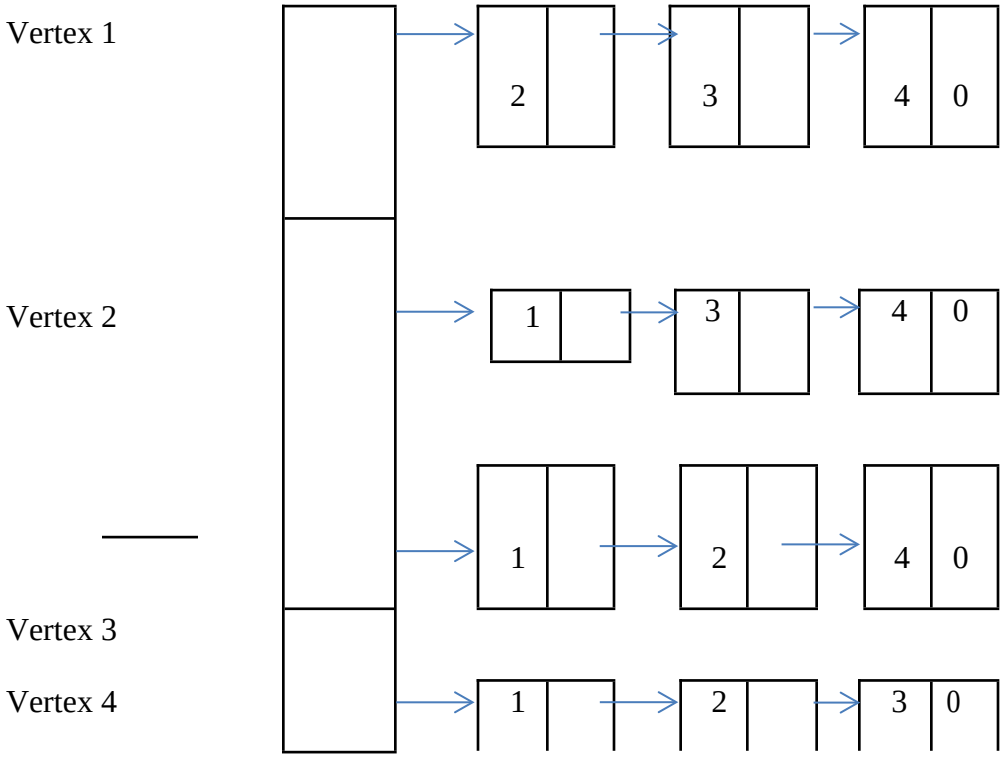
The adjacency matrix is a simple way to represent a graph , but it has 2 disadvantages,

- It takes O(n*n) space to represent a graph with n vertices ; even for sparse graphs and
- It takes… (n*n) times to solve most of the graph problems.

ADJACENCY LIST

In the representation of the n rows of the adjacency matrix are represented as n linked lists. There is one list for each vertex in G. the nodes in list I represent the vertices that are adjacent from vertex i. Each node has at least 2 fields: VERTEX and LINK. The VERTEX fields contain the

indices of the vertices adjacent to vertex i. In the case of an undirected graph with n vertices and E edges, this representation requires n head nodes and 2e list nodes.



The degree of the vertex in an undirected graph may be determined by just counting the number of nodes in its adjacency list. The total number of edges in G may , therefore be determined in time O(n+e).in the case of digraph the number of list nodes is only e. the out-degree of any vertex can be determined by counting the number of nodes in an adjacency list. The total number of edges in G can, therefore be determined in O(n+e).

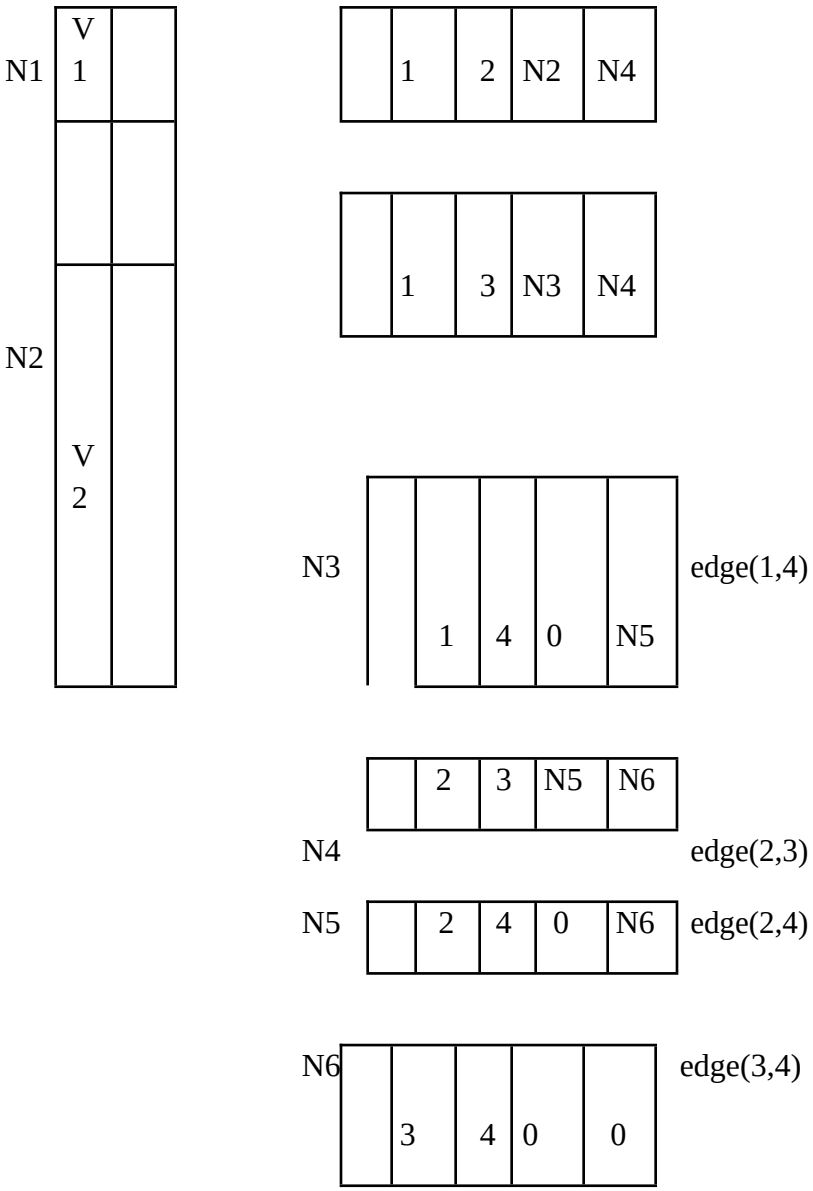
ADJACENCY MULTILIST:

In the adjacency list representation of an undirected graph each edge (vi, vj) is represented by two entries ,one on the list for Vi and the other on the list for Vj.

For each edge there will be exactly one node , but this node will be in two list , (i.e) the adjacency list for each of the two nodes it is incident to .the node structure now becomes

M	V_1	V_2	Link for V1	Link for V_2

Where M is a one bit mark field that may be used to indicate whether or not the edge has been examined. The adjacency multi-list diagram is as follow,



The lists are : v1: N1 N2 N3

V2: N1
N4 N5

V3: N2
N4 N6

V4: N3 N5 N6

FIG: Adjacency Multilists for G

3. What is graph traversal? What are its types? Explain (Nov 13, Nov 14, May 15)

GRAPH TRAVERSAL

Given an undirected graph G = (V.E) and a vertex v in V(G) we are interested in visiting all vertices in G that are reachable from v (that is all vertices connected to v). We have two ways to do the traversal. They are

DEPTH FIRST SEARCH

BREADTH FIRST SEARCH.

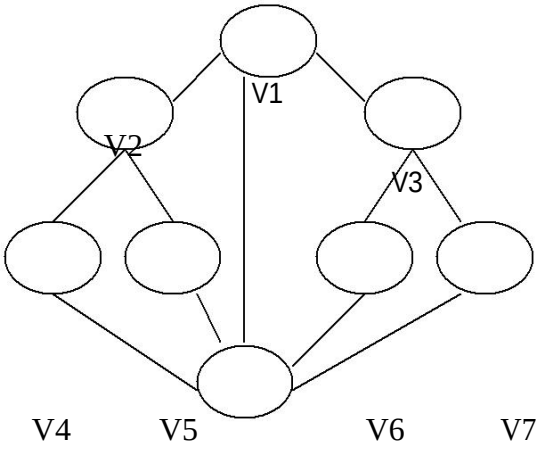
DEPTH FIRST SEARCH

In graphs, we do not have any start vertex or any special vertex singled out to start traversal from. Therefore the traversal may start from any arbitrary vertex.

We start with say, vertex v. An adjacent vertex is selected and a Depth First search is intimated from it, i.e. let V1, V2.. Vk are adjacent vertices to vertex v. We may select any vertex from this list. Say, we select V1. Now all the adjacent vertices to v1 are identified and all of those are visited; next V2 is selected and all its adjacent vertices visited and so on.

This process continues till are the vertices are visited. It is very much possible that we reach a traversed vertex second time. Therefore we have to set a flag somewhere to check if the vertex is already visited.

Let us see an example, consider the following graph.



Let us start with V1,

V8

1. Its adjacent vertices are V2, V8, and V3. Let us pick on v2.
2. Its adjacent vertices are V1, V4, V5, V1 is already visited.
3. Let us pick on V4.
4. Its adjacent vertices are V2, V8.
5. V2 is already visited .let us visit V8.
6. Its adjacent vertices are V4, V5, V1, V6, V7.

7. V4 and V1 are visited. Let us traverse V5.
8. Its adjacent vertices are V2, V8. Both are already visited therefore, we back track.
9. We had V6 and V7 unvisited in the list of V8. We may visit any. We may visit any. We visit V6.
10. Its adjacent is V8 and V3. Obviously the choice is V3.
11. Its adjacent vertices are V1, V7. We visit V7.
12. All the adjacent vertices of V7 are already visited, we back track and find that we have visited all the vertices.

Therefore the sequence of traversal is

V1, V2, V4, V5, V6, V3,
V7.

This is not a unique or the only sequence possible using this traversal method.

We may implement the Depth First search by using a stack, pushing all unvisited vertices to the one just visited and popping the stack to find the next vertex to visit

This procedure is best described recursively as in,

Procedure DFS(v)

// Given an undirected graph G = (V.E) with n vertices and an array visited (n) initially set to zero . This algorithm visits all vertices reachable from v .G and VISITED are global > //VISITED (v) 1

for each vertex w adjacent to v do *if*
VISITED (w) =0 then call DFS (w)
end

end DFS

COMPUTING TIME

1. In case G is represented by adjacency lists then the vertices w adjacent to v can be determined by following a chain of links. Since the algorithm DFS would examine each node in the adjacency lists at most once and there are 2e list nodes. The time to complete the search is O (e).
2. If G is represented by its adjacency matrix, then the time to determine all vertices adjacent to v is O(n).

Since at most n vertices are visited. The total time is $O(n^2)$.

BREADTH FIRST SEARCH

- In DFS we pick on one of the adjacent vertices; visit all of its adjacent vertices and back track to visit the unvisited adjacent vertices.
- In BFS , we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.
- Let us consider the same example, given in figure. We start say, with V1. Its adjacent vertices are V2, V8 , V3.
- We visit all one by one. We pick on one of these, say V2. The unvisited adjacent vertices to V2 are V4, V5 . we visit both.
- We go back to the remaining visited vertices of V1 and pick on one of this, say V3. T The unvisited adjacent vertices to V3 are V6,V7. There are no more unvisited adjacent vertices of V8, V4, V5, V6 and V7.

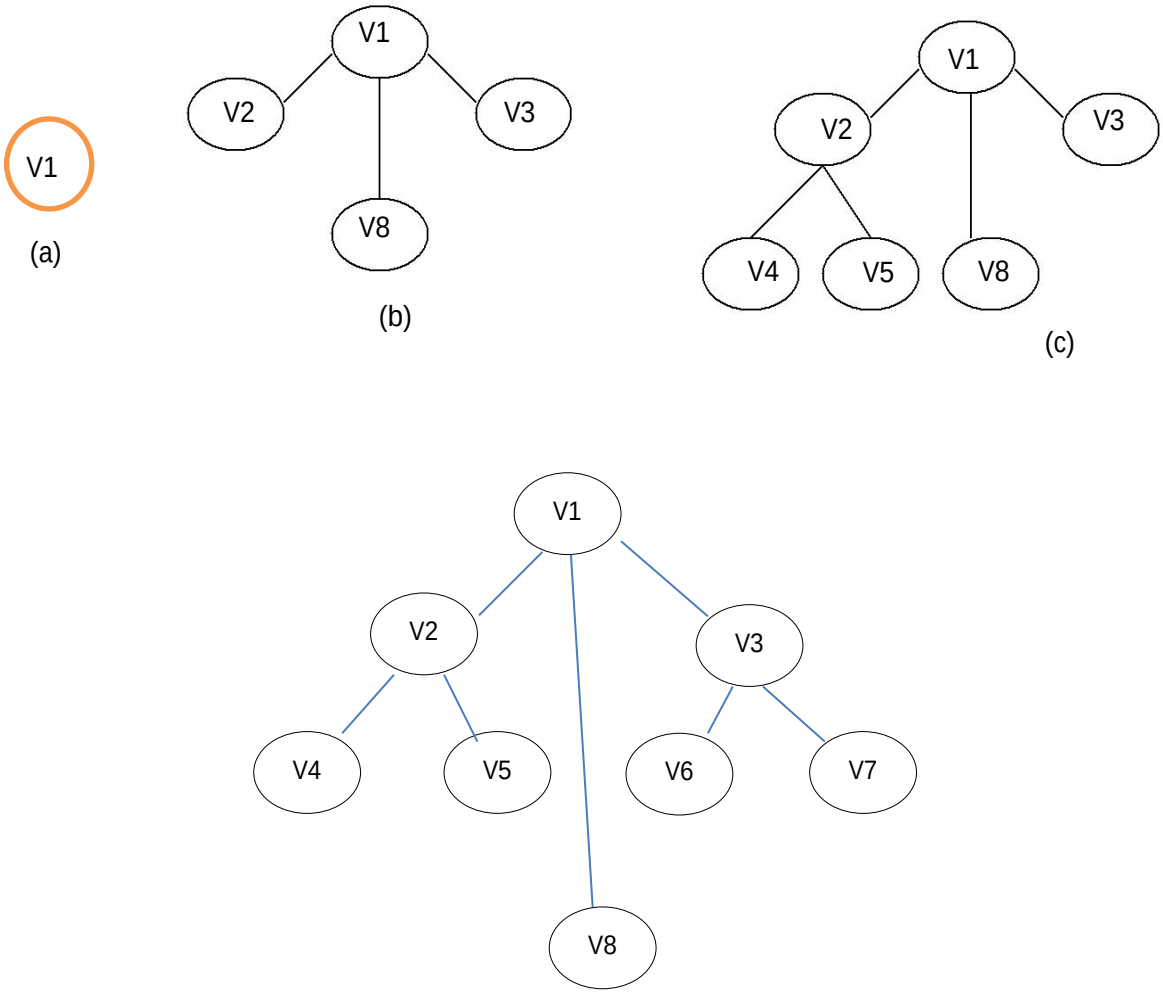


FIG: Breadth First Search

14. Thus the sequence so generated is V1,V2, V8, V3,V4, V5,V6, V7. Here we need a queue instead of a

stack to implement it.

15. We add unvisited vertices adjacent to the one just visited at the rear and read at from to find the next vertex to visit.

Algorithm BFS gives the details.

Procedure BFS(v)

//A breadth first search of G is carried out beginning at vertex v. All vertices visited are marked as VISITED(I) = 1. The graph G and array VISITED are global and VISITED is initialised to 0.//

1. VISITED(v) 1
2. *Initialise Q to be empty //Q is a queue//*
3. loop
4. for all vertices w adjacent to v do
5. if VISITED(w) = 0 //add w to queue//
6. then [call ADDQ(w, Q); VISITED(w) 1] //mark w as VISITED//
7. end
8. if Q is empty then return
9. call DELETEQ(v,Q)
10. forever
11. end BFS

COMPUTING TIME

iii. Each vertex visited gets into the queue exactly once, so the loop forever is iterated at most n times.

iv. If an adjacency matrix is used, then the for loop takes O(n) time for each vertex visited. The total time is, therefore, $O(n^2)$.

v. In case adjacency lists are used the for loop as a total cost of d1+……..+dn = O(e) where di = degree(vi). Again, all vertices visited. Together with all edges incident to from a connected component of G.

3. What is graph traversal? What are its types? Explain (April 2011, April 2012, April 2013) Graph Traversal

Some applications require the graph to be traversed . This means that starting from some designated vertex the graph is walked around in a symmetric way such that vertex is visited only once. Two algorithms are commonly used:

1. Depth-First Search

2. Breadth-First Search

DEPTH FIRST SEARCH

1. DFS stands for "Depth First Search".
2. DFS in not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible.
3. DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise.
4. Depth First Search can be done with the help of Stack i.e. LIFO implementations.
5. This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped- off.
6. DFS is more faster than BFS.

Applications of DFS

1. Useful in Cycle detection
2. In Connectivity testing
3. Finding a path between V and W in the graph
4. useful in finding spanning trees & forest.

Algorithm: Depth First Search

1. PUSH the starting node in the Stack.

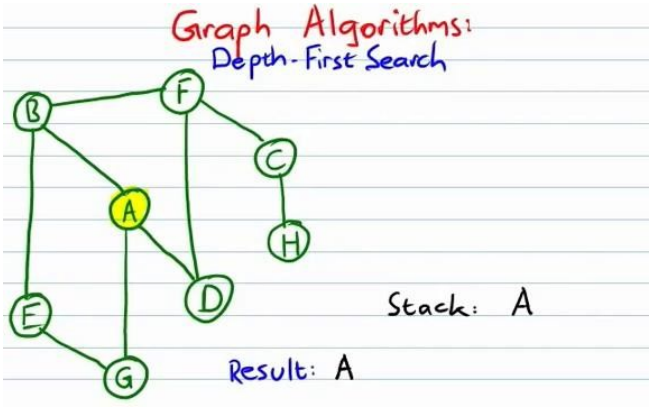
2. If the Stack is empty, return failure and stop.

3. If the first element on the Stack is a goal node g , return succeed and stop otherwise.

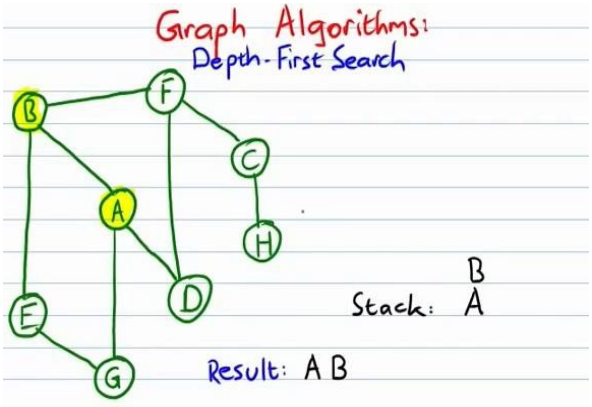
4. POP and expand the first element and place the children at the front of the Stack.

5. Go back to step 2

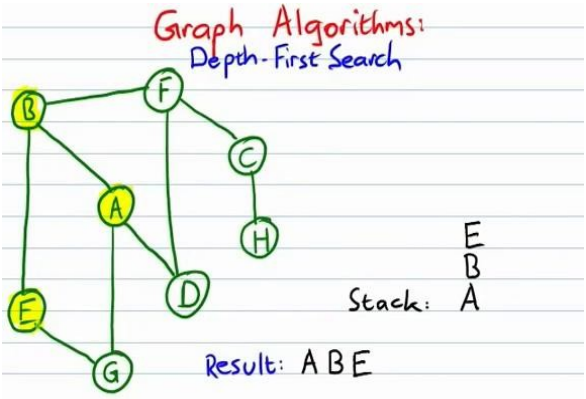
Step 1



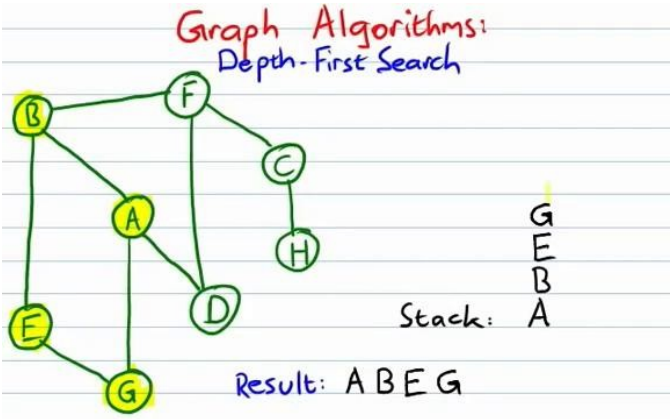
Step 2



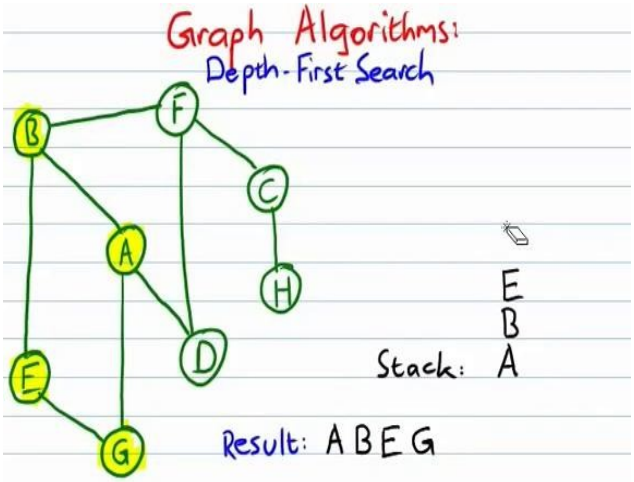
Step 3



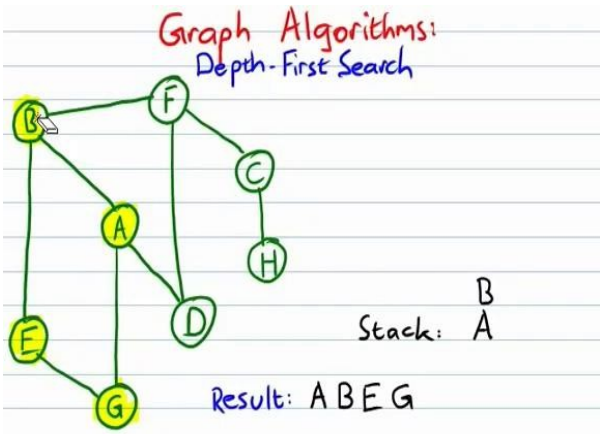
Step 4



Step 5

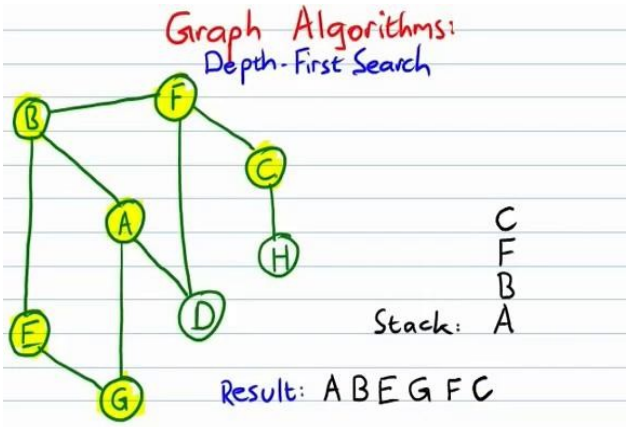


Step 6

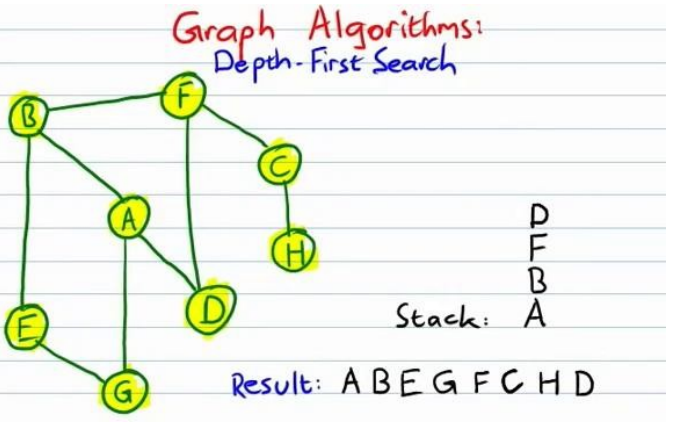


Step 7

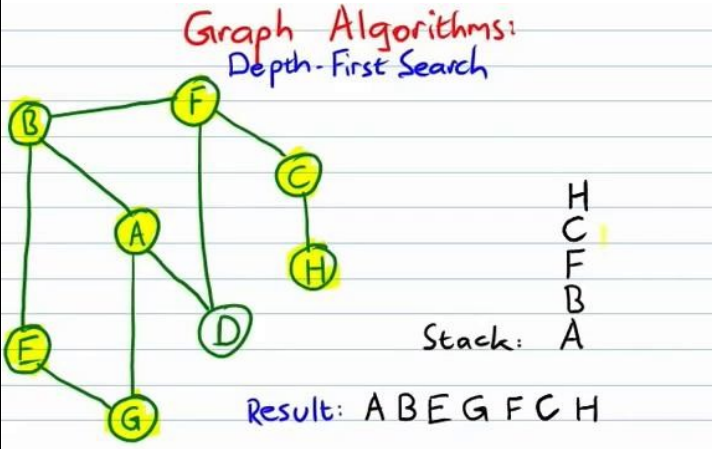
Step 8

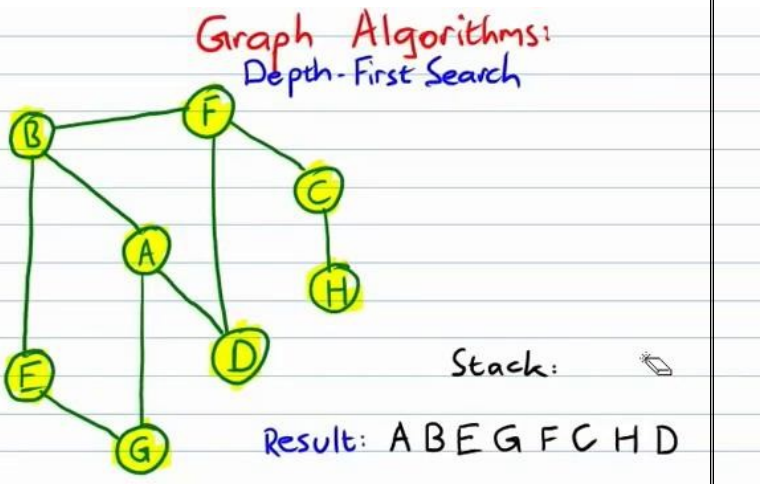


Step 9



Step 10





2. BREADTH FIRST SEARCH:-

1. BFS Stands for “Breadth First Search”.
2. BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.
3. BFS is useful in finding shortest path.BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph.
4. Breadth First Search can be done with the help of queue i.e. FIFO implementation.
5. This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.
6. BFS is slower than DFS.
7. BFS requires more memory compare to DFS.

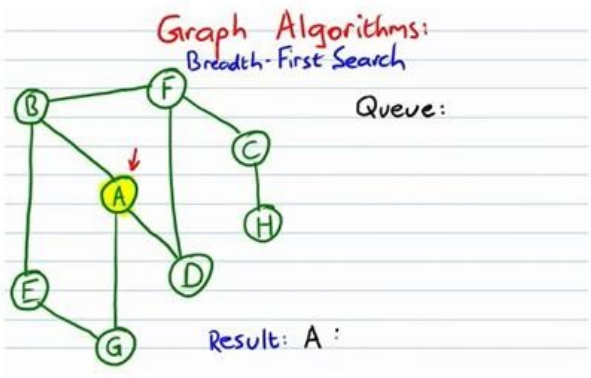
Applications of BFS

1. To find Shortest path
2. Single Source & All pairs shortest paths
3. In Spanning tree
4. In Connectivity

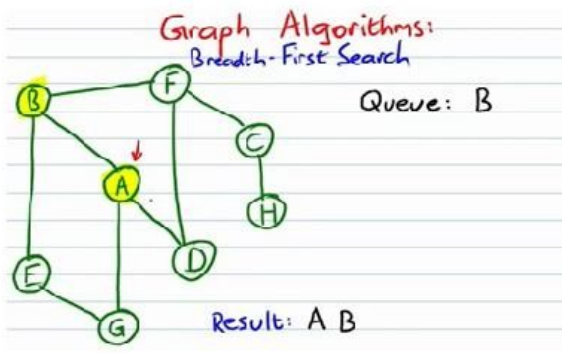
ALGORITHM: BREADTH - FIRST SEARCH

1. Place the starting node on the queue.
2. If the queue is empty return failure and stop.
3. If the first element on the queue is a goal node, return success an stop otherwise.
4. Remove and expand the first element from the queue and place all children at the end of the queue in any order.
5. Go back to step 1.

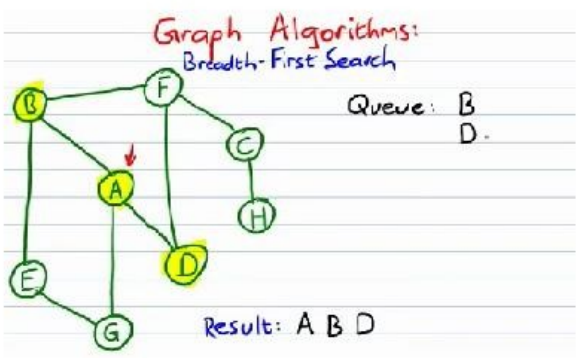
Step 1



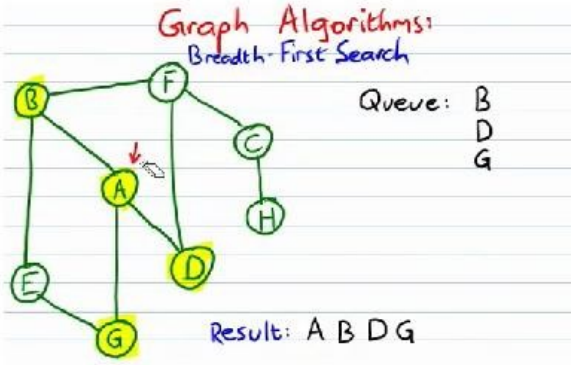
Step 2



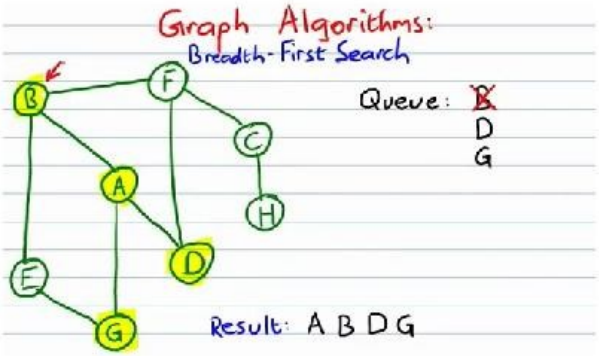
Step 3



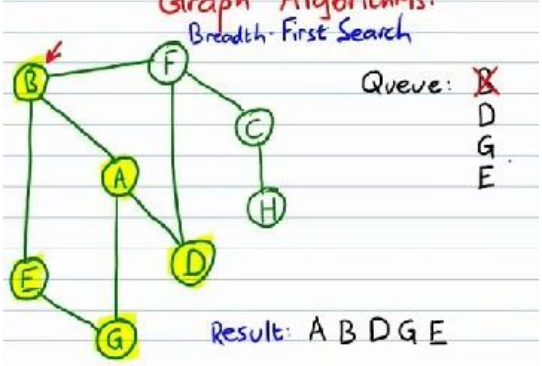
Step 4



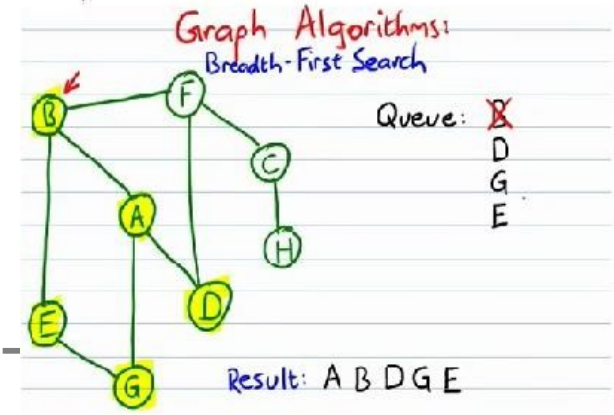
Step 5



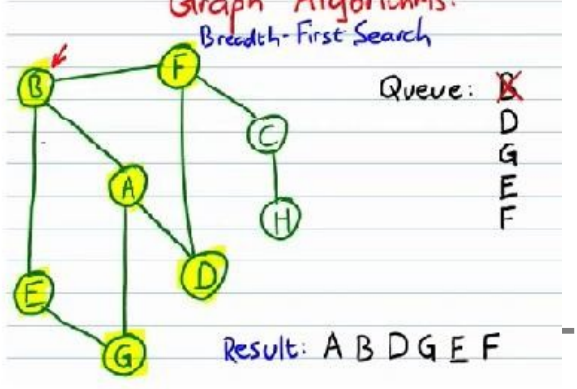
Step 6



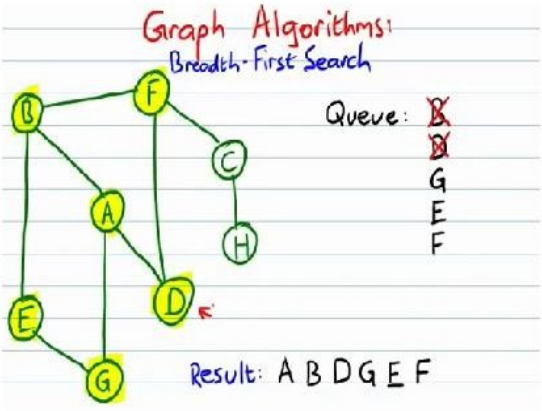
Step 7



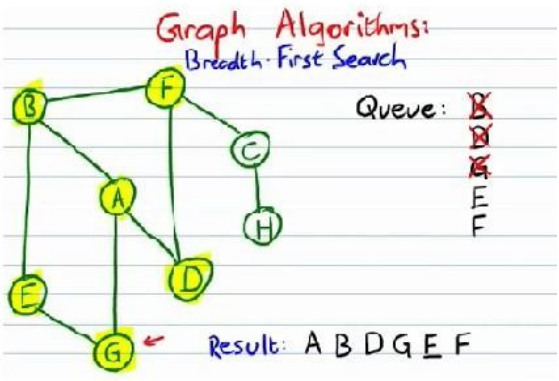
Step8



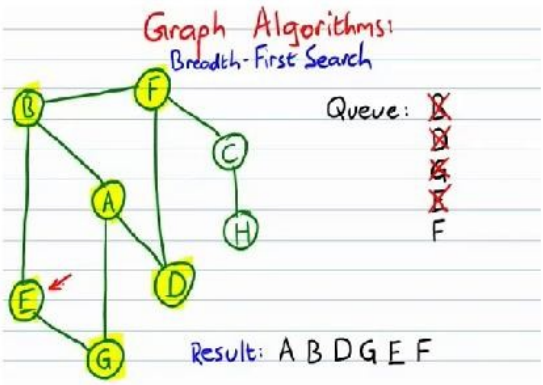
Step 9



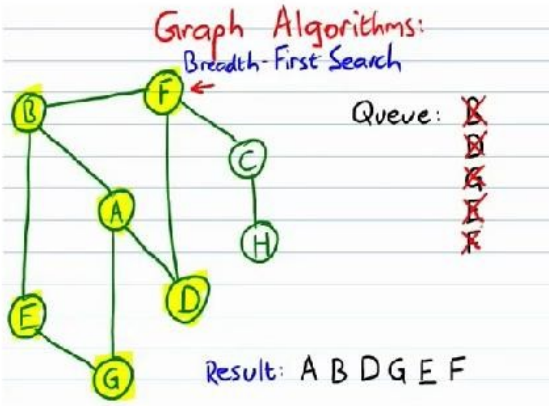
Step 10



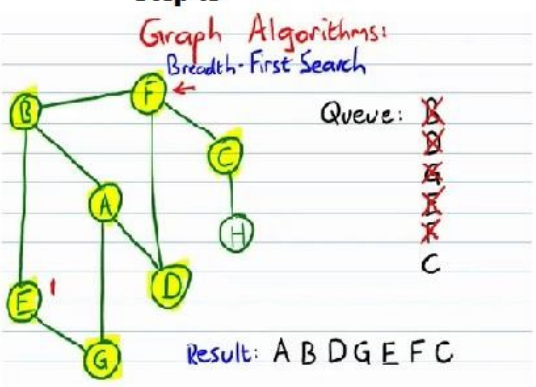
Step 11



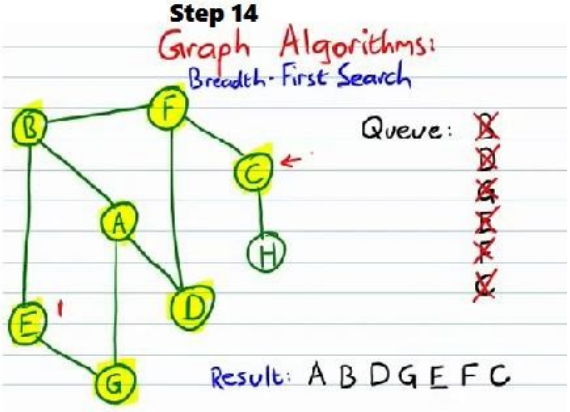
Step 12



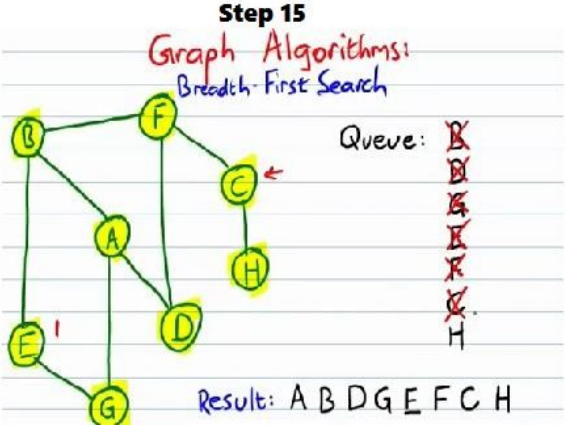
Step 13



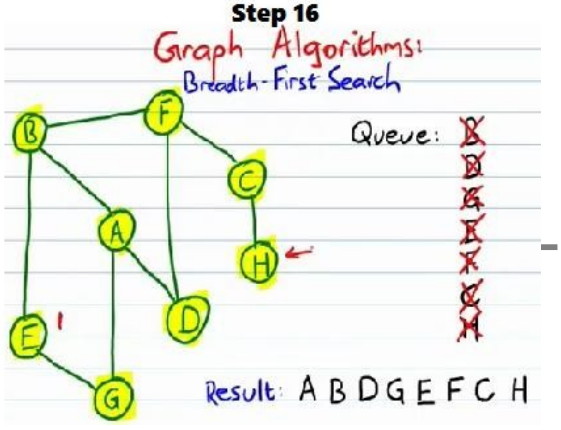
Step 14



Step 15

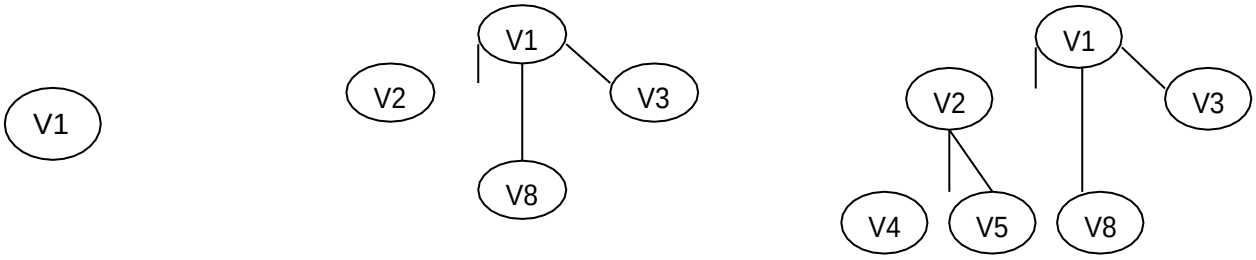


Step 16

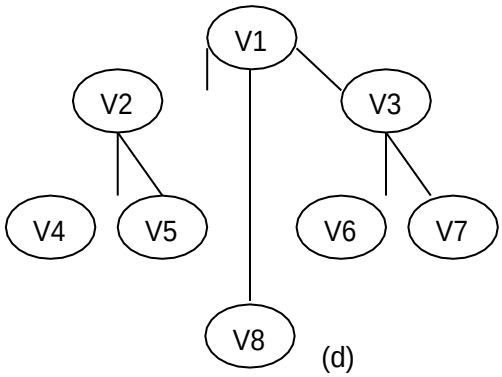


BREADTH FIRST SEARCH

- In DFS we pick on one of the adjacent vertices; visit all of its adjacent vertices and back track to visit the unvisited adjacent vertices.
- In BFS , we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.
- Let us consider the same example, given in figure. We start say, with V1. Its adjacent vertices are V2, V8, V3.
- We visit all one by one. We pick on one of these, say V2. The unvisited adjacent vertices to V2 are V4, V5 . we visit both.
- We go back to the remaining visited vertices of V1 and pick on one of this, say V3. T The unvisited adjacent vertices to V3 are V6,V7. There are no more unvisited adjacent vertices of
- V8, V4, V5, V6 and V7.



(a) (b) (c)



(d)

FIG: Breadth First Search

- Thus the sequence so generated is V1,V2, V8, V3,V4, V5,V6, V7. Here we need a queue instead of a stack to implement it.
- We add unvisited vertices adjacent to the one just visited at the rear and read at from to find the next vertex to visit.

Algorithm BFS gives the details.

Procedure BFS(v)

//A breadth first search of G is carried out beginning at vertex v. All vertices visited are marked as VISITED(I)

= 1. The graph G and array VISITED are global and VISITED is initialised to 0.//

1. VISITED(v) □ 1
2. Initialise Q to be empty //Q is a queue//
3. loop
4. for all vertices w adjacent to v do
5. if VISITED(w) = 0 //add w to queue//
6. then [call ADDQ(w, Q); VISITED(w) □ 1] //mark w as VISITED//
7. end
8. if Q is empty then return
9. call DELETEQ(v,Q)
10. forever
11. end BFS

Computing Time

1. Each vertex visited gets into the queue exactly once, so the loop forever is iterated at most n times.
2. If an adjacency matrix is used, then the for loop takes O(n) time for each vertex visited. The Total time is, therefore, $O(n^2)$.
3. In case adjacency lists are used the for loop as a total cost of $d_1+\ldots\ldots..+d_n = O(e)$ where d_i = degree(v_i). Again, all vertices visited. Together with all edges incident to from a connected component of G.

4. What is minimum Spanning tree? Explain Kruskal's algorithm with the graph.(Nov 2012) Spanning Tree

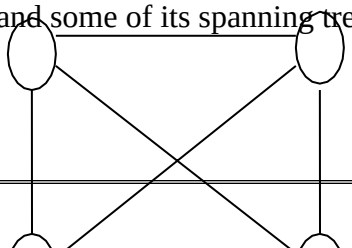
- When the graph G is connected, a depth first search starting at any vertex, visits all the vertices in G.
- In this case the edges of G are partitioned into two sets T (for tree edges) and B (for back edges), where T is the set of edges used or traversed during the search and B the set of remaining edges.
- The set T may be determined by inserting the statement

$$T \longleftarrow T \cup \{(v,w)\}$$

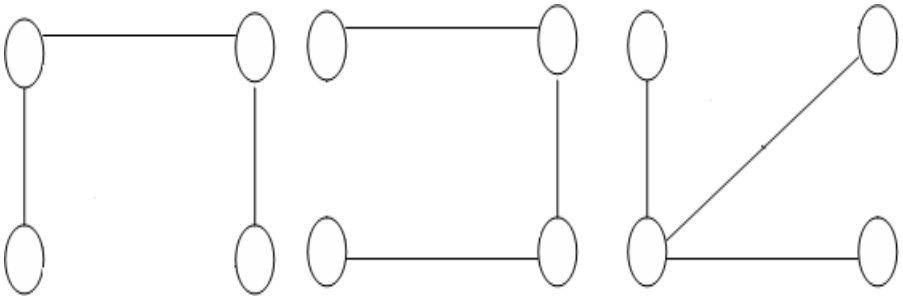
In the then clauses of DFS and BFS.

- The edges in T form a tree which includes all the vertices of G
- Any tree consisting solely of edges in G and including all vertices in G is called **spanning tree.**

The below diagram shows the graph G and some of its spanning trees.



Graph G



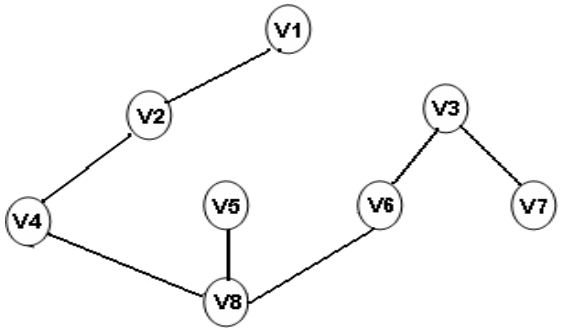
Its spanning trees

➢ **DEPTH FIRST SPANNING TREE**

When DFS are used the edges of T form a spanning tree

o The spanning tree resulting from a call to DFS is known as a depth first spanning tree.

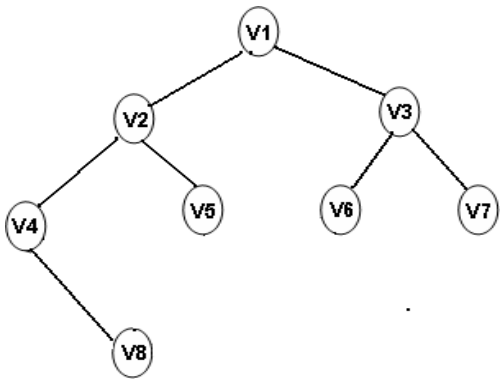
The below diagram shows the Depth first spanning tree



➢ BREADTH FIRST SPANNING TREE

- When BFS are used the edges of T form a spanning tree.
- The spanning tree resulting from a call to BFS is known as a breadth first spanning tree .

The below diagram shows the breadth first spanning tree



MINIMUM COST SPANNING TREES (APPLICATIONS OF SPANNING TREE):

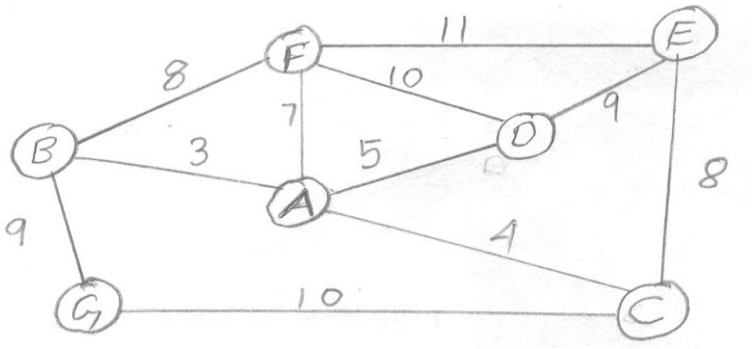
BUILDING A MINIMUM SPANNING TREE

- If the nodes of G represent cities and the edges represent possible communication links connecting two cities then the minimum number of links needed to connect the n cities is n-1. The spanning tree of G will represent all feasible choices.
- This application of spanning tree arises from the property that a spanning tree is a minimal sub-graph G' of G such that V (G') =V (G) and G' is connected where minimal sub-graph is one with fewest number of edges.
- The edges will have weights associated to there i.e. cost of communication. Given the weighted graph, one has to construct a communication links that would connect all the cities with minimum total cost.
- Since, the links selected will form a tree. We are going to find the spanning tree of a given graph with minimum cost. The cost of a spanning tree is the sum of the costs of the edges in the tree.

KRUSKAL'S METHOD FOR CREATING A SPANNING TREE

One approach to determine a minimum cost spanning of a graph has been given by kruskal. In this approach we need to select (n-1) edges in G, edge by edge, such that these form an MST or G. We can select another least cost edge and so on. An edge is included in the tree if it does not form a cycle with the edges already in T.

For example, consider the following graph,



The minimum cost edge is that connects vertex A and B. let us choose that Fig.(a)

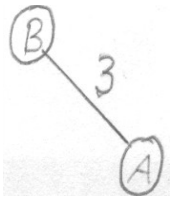


Fig. (a)

Now we have, Vertices that may be added,

	Edge	cost
F	BF, AF	8, 7
G	BG, CG	9, 10
D	AD	5
E	CE	8
C	AC	4

The least cost edge is AC therefore we choose AC. Now we have Fig. (b),

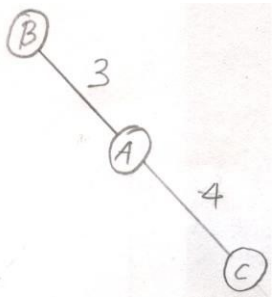
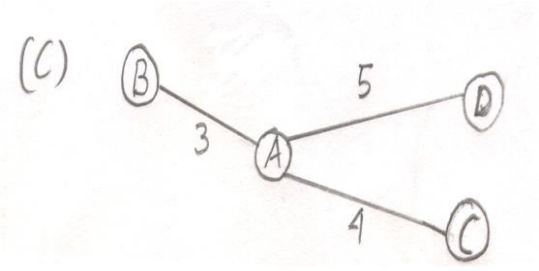


Fig. b

	EDGE	COST
F	BF	8
	AF	7
G	BG	9
	CG	10
D	AD	5
E	CE	8
C	AC	4

The least cost edge is Ad therefore we have



Vertices that may be added,

	EDGE	COST
F	BF	8
	AF	7
	DF	10
G	BG	9
	CG	10
E	CE	8
	DE	9

AF is the minimum cost edge therefore we add it to the partial tree we have

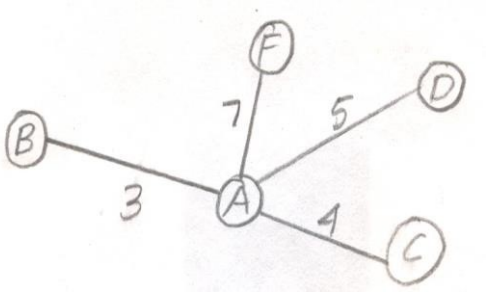
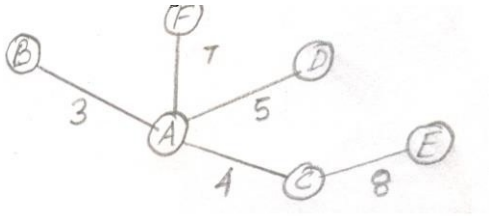


Fig.: (d)

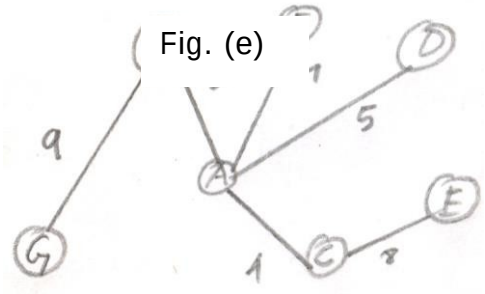
	edge	cost
G	BG	9
	CG	10
E	CE	8
	DE	9
	FE	11

Obvious choice is CE, shown in figure (e),



The only vertex left is G and we have the minimum cost edge that connects it to the tree constructed so for is BG. Therefore add it and the costs 9+3+7+5+4+8+36 and is given by the following figure,

Fig. (e)



ALGORITHM

1. T<-0

2. While T contains less than(n-1) edges and E not empty do
3. Choose an edge(V,W) from E of lowest cost
4. Delete (V,W) from E

5. if (V,W) does not create a cycle in T
6. then add(V,W) to T
7. else discard (V,W)
8. end
9. if T contains fewer than (n-1) edges then print (no spanning tree)

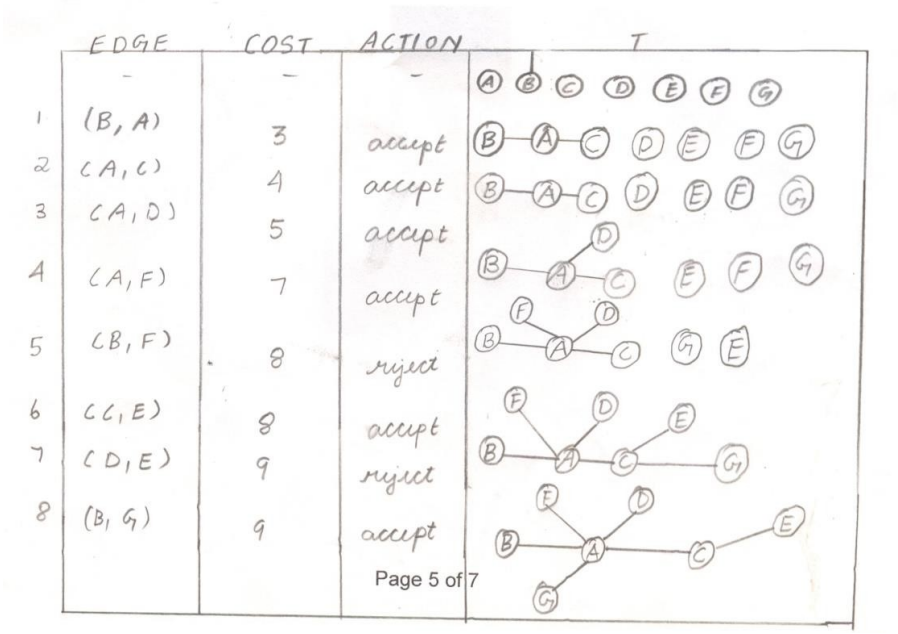
Initially, E is the set of all edges in G. In this set, we are going to (I) determining an edge with minimum cost (in line 3)

(ii) Delete that edge (line 4)

This can be done efficiently if the edges in E are maintained as a sorted sequential list.

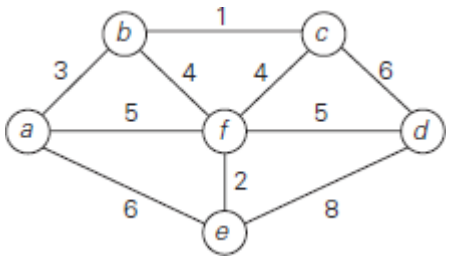
In order to be able to perform steps 5&6 efficiently, the vertices in G should be grouped together in such way that one may easily determine of the vertices V and W are already connected by the earlier selection of edges.

In case they are, then the edge (V,W) is to be discarded. If they are not then (V,W) is to be added to T. Our possible grouping is to place all vertices in the same connected components of T into the set. Then two vertices V,W are connected in t if they are in the same set.



Here for example when the edge (B,F) is to be considered, the sets would be {A,B,C,D,F},{E},{F}.vertices B and F are in the same set and so the edge (B,F) is rejected. The next edge to be considered is (C, E).since vertices C and E are in different sets the edge is accepted. This edge connects the two components {A, B, C, D, F} and {E} together and so these two sets should be joined to obtain the set representing the new component.

5.Explain Prim's Algorithm in detail and find the Minimum Spanning Tree for the below graph.



Spanning tree:

In a Graph G(V,E), Spanning tree of 'G' is a selection of edges of G that form a tree spanning every vertex.(i.e) every vertex lies in the tree, but, no cycle are formed

Minimum Spanning Tree:

Minimum spanning tree is a spanning tree with weight less or equal to weight of every other spanning tree.

Algorithms used to find minimum spanning tree:

- Prim's Algorithm
- kruskal's algorithm

Algorithm:

To find minimum spanning tree T.

Step 1:

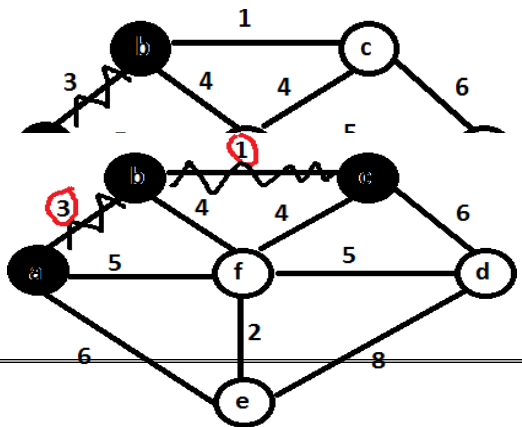
Select any node to be the first of T.

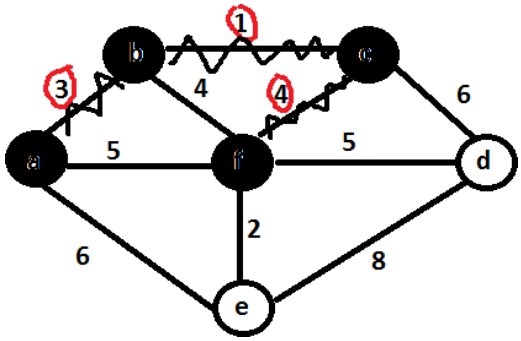
Step 2:

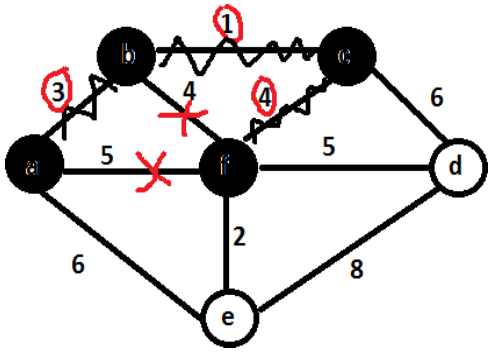
Consider which arcs connects nodes in T to nodes outside T. Pick one with minimum weight(if more than one, choose any). Add this arc and node to T.

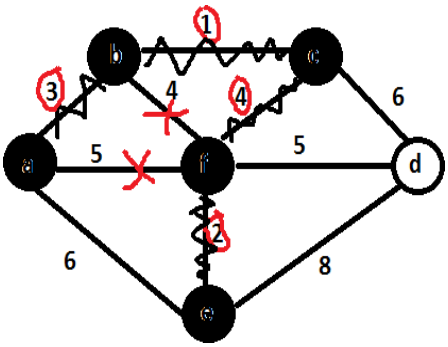
Step 3:

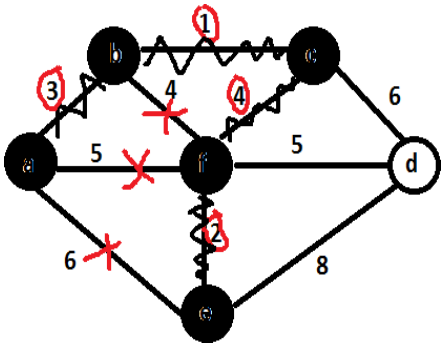
Repeat step 2 until T contains every node of the graph.











3 + 1 + 4 + 6 + 5 + 2 = 21.

Running time Performance:

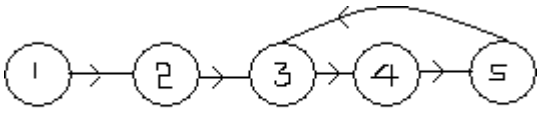
Adjacency matrix: O $|V^2|$

Binary heap: O (E log V)

Fibonacci heap: O (E + V log V)

6. Explain transitive closure with examples.

A problem related to the all pairs shortest path problem is that of determining for every pair of vertices *i,j* in *G* the existence of a path from *i* to *j*. Two cases are of interest, one when all path lengths (i.e., the number of edges on the path) are required to be positive and the other when path lengths are to be nonnegative. If *A* is the adjacency matrix of *G*, then the matrix A^+ having the property $A^+(i,j) = 1$ if there is a path of length > 0 from *i* to *j* and 0 otherwise is called the *transitive closure* matrix of *G*. The matrix A^* with the property $A^*(i,j) = 1$ if there is a path of length 0 from *i* to *j* and 0 otherwise is the *reflexive transitive closure* matrix of *G*.



a Digraph G

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

b Adjacency matrix A for G

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

c A^+

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

d A^*

Figure Graph G and Its Adjacency Matrix A, A^+ and A^*

Figure shows A^+ and A^* for a digraph. Clearly, the only difference between A^* and A^+ is in the terms on the diagonal. $A^+(i,i) = 1$ iff there a cycle of length > 1 containing vertex i while $A^*(i,i)$ is always one as there is a path of length 0 from i to i. If we use algorithm ALL_COSTS with COST(i,j) = 1 if $<i,j>$ is an edge in G and

COST(i,j) = + if $<i,j>$ is not in G, then we can easily obtain A^+ from the final matrix A by letting A^+ (i,j) = 1 iff A (i,j) < +∞. A^* can be obtained from A^+ by setting all diagonal elements equal 1. The total time is $O(n^3)$. Some simplification is achieved by slightly modifying the algorithm. In this modification the computation of line 9 of ALL_COSTS becomes $A(i,j)$ □A (i,j) **or** ($A(i,k)$ **and** $A(k,)$) and COST(i,j) is just the adjacency matrix of G. With this modification, A need only be a bit matrix and then the final matrix A will be A^+.

7. Describe the applications of graph with example.

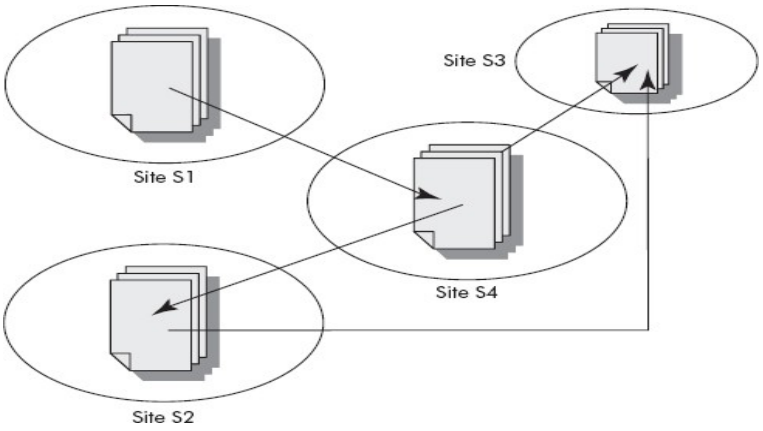
Structures that can be represented as graphs are ubiquitous, and many problems of practical interest can be represented by graphs. The link structure of a website could be represented by a directed graph: the vertices are the web pages available at the website and a directed edge from page A to page B exists if and only if A contains a link to B. A similar approach can be taken to problems in travel, biology, computer chip design, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science. There, the transformation of graphs is often formalized and represented by graph rewrite systems. They are either directly used or properties of the rewrite systems(e.g. confluence) are studied.

A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pairwise connections have some numerical values. For example if a graph represents a road network, the weights could represent the length of each road. A digraph with weighted edges in the context of graph theory is called a network.

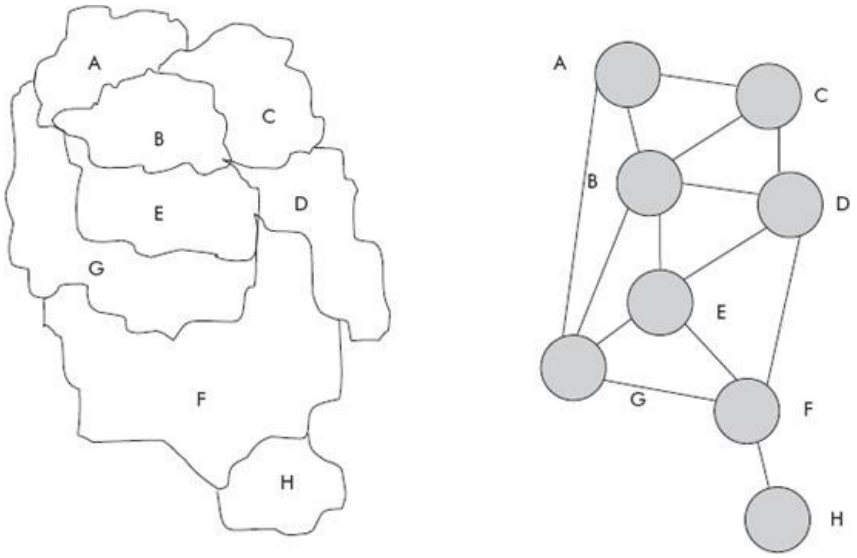
Networks have many uses in the practical side of graph theory, network analysis (for example, to model and analyze traffic networks). Within network analysis, the definition of the term "network" varies, and may often refer to a simple graph.

Model of www: The model of world wide web (www) can be represented by a graph (directed) wherein nodes denote the documents, papers, articles, etc. and the edges represent the outgoing hyperlinks between them as shown in

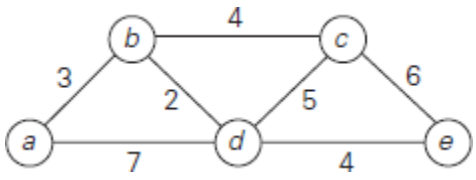
Figure



Coloring of maps: Coloring of maps is an interesting problem wherein it is desired that a map has to be colored in such a fashion that no two adjacent countries or regions have the same color. The constraint is to use minimum number of colors. A map can be represented as a graph wherein a node represents a region and an edge between two regions denote that the two regions are adjacent.



8. <u>Discuss in detail Dijikstra's algorithm and Find the shortest path for the below graph using the same Algorithm.</u>



Shortest Path: The shortest path problem is the problem of finding a PATH between 2 nodes(vertices), such that the sum of weight of its constituent edge is minimized.

Eg: finding quickest way to go to one location to another.

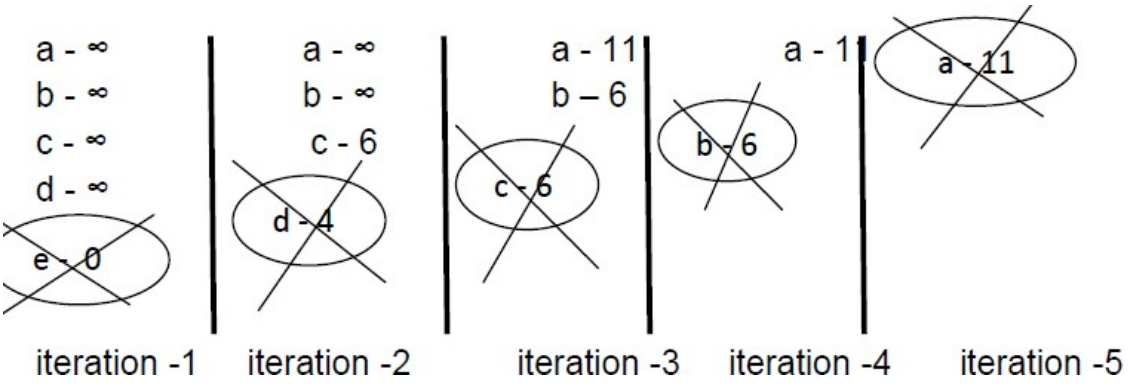
Single source shortest path: Finding shortest path from source vertex to all other vertex in the graph.

Dijkstra Algorithm :

```
1 function Dijkstra(Graph, source):
2 for each vertex v in Graph: // Initializations
3 dist[v] := infinity ; // Unknown distance function from
4 // source to v
5 previous[v] := undefined ; // Previous node in optimal path
6 end for // from source
7
8 dist[source] := 0 ; // Distance from source to source
9 Q := the set of all nodes in Graph ; // All nodes in the graph are
10 // unoptimized – thus are in Q
11 while Q is not empty: // The main loop
12 u := vertex in Q with smallest distance in dist[] ; // Source node in first case
13 remove u from Q ;
14 if dist[u] = infinity:
15 break ; // all remaining vertices are
16 end if // inaccessible from source
17
18 for each neighbor v of u: // where v has not yet been
19 // removed from Q.
20 alt := dist[u] + dist_between(u, v) ;
21 if alt < dist[v]: // Relax (u,v,a)
22 dist[v] := alt ;
23 previous[v] := u ;
24 decrease-key v in Q; // Reorder v in the Queue
25 end if
26 end for
27 end while
28 return dist;
29 endfunction
```

From the graph..'e' is selected as Source.

Min heap:



iteration	S	d[a]	d[b]	d[c]	d[d]	p[a]	p[b]	p[c]	p[d]
0	Ø	∞	∞	∞	∞	-	-	-	-
1	e = 0	∞	∞	0+6=6	0+4=4	-	-	e	e
2	D = 4	4+7 = 11	4+2=6	4+5=9 9>6, So, **6**	**4**	d	d	**e**	**e**
3	C = 6	**11**	**6**	**6**	**4**	**d**	**d**	**e**	**e**
4	B = 6	6+3=9 9< 11 So, 9	**6**	**6**	**4**	b	**d**	**e**	**e**
5	A = 11	**9**	**6**	**6**	**4**	**b**	**d**	**e**	**e**

Running Performance:

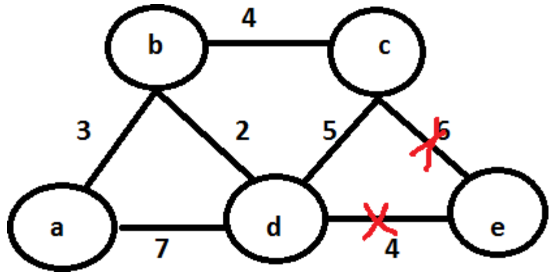
On array – O | V ² |

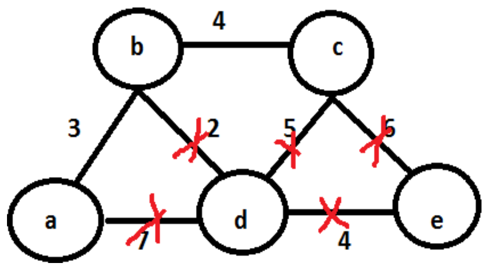
On binary heap – O (E + V log

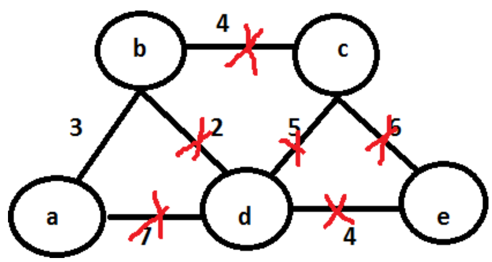
V) On Fibonacci heap – O (log

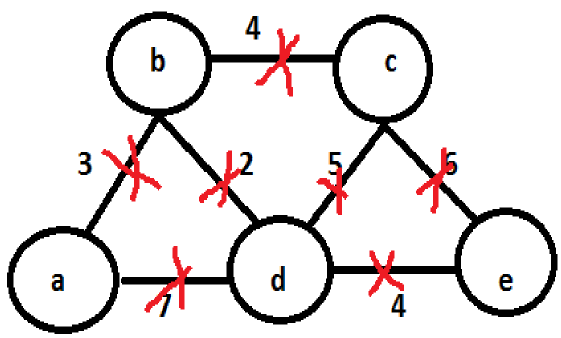
V) Where E – edges, V –

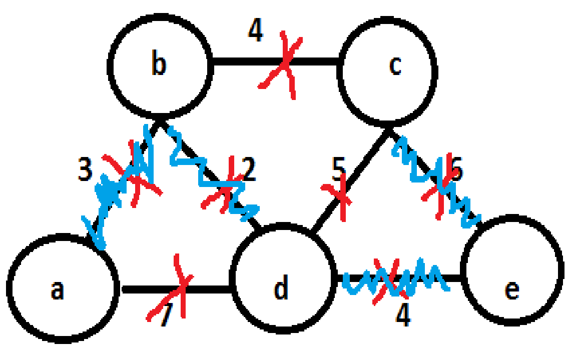
vertices











RESULT: e -> a = 9

e -> b = 6

e -> c = 6

e -> d = 4

Important Program

1. SHORTEST PATH ALGORITHM - FLOYD'S ALGORITHM

```
#include<stdio.h>

void all_path();

int cost[20][20],n;

void main()

{

        int i,j;

        clrscr();

        printf("ALL PAIR SHORTEST PATH");

        printf("\nEnter the number of nodes");

        scanf("%d",&n);

        printf("\nEnter the cost matrix");

        for(i=1;i<=n;i++)

        {

        for(j=1;j<=n;j++)

             {

                    scanf("%d",&cost[i][j]);

              }

        }

        all_path();

        getch();

}

        void all_path()
```

```
{
        int i,j,k,a[20][20];
        for(i=1;i<=n;i++)
            {
               for(j=1;j<=n;j++)
                   {
                       a[i][j]=cost[i][j];
                   }
               }

         for(k=1;k<=n;k++)
                {
                     for(i=1;i<=n;i++)
                          {
                               for(j=1;j<=n;j++)
                                    {
                                         if(a[i][j]<(a[i][k]+a[k][j]))
                                              {
                                                   a[i][j]=a[i][j];
                                              }
                                          else
                                              {
                                                   a[i][j]=(a[i][k]+a[k][j]);
```

```
                    }
                }
            }
            printf("STEP:%d\n",k);
            for(i=1;i<=n;i++)
            {
                for(j=1;j<=n;j++)
                {
                    printf(" %d",a[i][j]);
                }
                printf("\n");
            }
        }
    }
```

2. SHORTEST PATH ALGORITHM - DIJKSTRA'S ALGORITHM

```
#include <stdio.h>
#include <conio.h>
#define infinity 999

void dij(int n,int v,int cost[10][10],int dist[])
{
 int i,u,count,w,flag[10],min;
 for(i=1;i<=n;i++)
```

```
 flag[i]=0,dist[i]=cost[v][i];
 count=2;
 while(count<=n)
 {
      min=99;
          for(w=1;w<=n;w++)
               if(dist[w]<min && !flag[w])
                    min=dist[w],u=w;
                    flag[u]=1;
                   count++;
        for(w=1;w<=n;w++)
            if((dist[u]+cost[u][w]<dist[w]) && !flag[w])
                  dist[w]=dist[u]+cost[u][w];
  }
}


void main()
{
 int n,v,i,j,cost[10][10],dist[10];
 clrscr();
 printf("\n Enter the number of nodes:");
 scanf("%d",&n);
 printf("\n Enter the cost matrix:\n");
```

```
 for(i=1;i<=n;i++)


for(j=1;j<=n;j++)
  {
        scanf("%d",&cost[i][j]);
        if(cost[i][j]==0)
        cost[i][j]=infinity;
  }
 printf("\n Enter the source matrix:");
 scanf("%d",&v);
 dij(n,v,cost,dist);
 printf("\n Shortest path:\n");
 for(i=1;i<=n;i++)
         if(i!=v)
              printf("%d->%d,cost=%d\n",v,i,dist[i]);
              getch();
}
```

3. SPANNING TREE - KRUSKAL'S ALGORITHM

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
```

```
int find(int);
int uni(int,int);
void main()
{
 clrscr();
 printf("\n\n\tImplementation of Kruskal's algorithm\n\n");
 printf("\nEnter the no. of vertices\n");
 scanf("%d",&n);
 printf("\nEnter the cost adjacency matrix\n");
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=n;j++)
        {
             scanf("%d",&cost[i][j]);
             if(cost[i][j]==0)
             cost[i][j]=999;
        }
 }
 printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
 while(ne<n)
        {
               for(i=1,min=999;i<=n;i++)
                     {
```

```
                for(j=1;j<=n;j++)
                    {
                        if(cost[i][j]<min)
                            {

                                min=cost[i][j];
                                a=u=i;
                                b=v=j;
                            }
                    }
                }
            u=find(u);
            v=find(v);
           if(uni(u,v))
                {
                    printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
                    mincost +=min;
                }
          cost[a][b]=cost[b][a]=999;
    }
 printf("\n\tMinimum cost = %d\n",mincost);
 getch();
}
```

```
int find(int i)
{
        while(parent[i])
        i=parent[i];
        return i;
}
int uni(int i,int j)
{
 if(i!=j)
 {
  parent[j]=i;
  return 1;
 }return 0;
}
```

4. SPANNING TREE - PRIM'S ALGORITHM

```
#include<stdio.h>
#include<conio.h>
int a,b,u,v,n,i,j,ne=1;
int visited[10]={0},min,mincost=0,cost[10][10];
void main()
{
 clrscr();
 printf("\n Prim's Algorithm");
 printf("\n Enter the number of nodes:");
```

```
scanf("%d",&n);
printf("\n Enter the adjacency matrix:\n");
for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
        {
             scanf("%d",&cost[i][j]);
             if(cost[i][j]==0)
             cost[i][j]=999;
         }
visited[1]=1;
printf("\n");
while(ne<n)
     {
          for(i=1,min=999;i<=n;i++)
              for(j=1;j<=n;j++)
                if(cost[i][j]<min)
                    if(visited[i]!=0)
                         {
                              min=cost[i][j];
                              a=u=i;
                              b=v=j;
                         }
                 if(visited[u]==0 || visited[v]==0)
```

```
                {
                        printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
                        mincost+=min;
                        visited[b]=1;
                }
              cost[a][b]=cost[b][a]=999;
          }
 printf("\n Minimun cost=%d",mincost);
 getch();
}
```

5. GRAPH TRAVERSAL- BREATH FIRST SEARCH

```
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int v)
{
 for(i=1;i<=n;i++)
  if(a[v][i] && !visited[i])
          q[++r]=i;
 if(f<=r)
        {
           visited[q[f]]=1;
           bfs(q[f++]);
```

```
        }
    }
void main()
{
 int v;
 clrscr();
 printf("\nBreadth First Search");
 printf("\n Enter the number of vertices:");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
       {
              q[i]=0;
              visited[i]=0;
        }
 printf("\n Enter graph data in matrix form:\n");
 for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
         scanf("%d",&a[i][j]);
 printf("\n Enter the starting vertex:");
 scanf("%d",&v);
 bfs(v);
 printf("\n The node which are reachable are:\n");
 for(i=1;i<=n;i++)
```

```
        if(visited[i])
            printf("%d\t",i);
        else
            printf("\n Bfs is not possible");
        getch();
}
```

6. GRAPH TRAVERSAL- DEPTH FIRST SEARCH

```
#include<stdio.h>
#include<conio.h>
int a[20][20],reach[20],n;
void dfs(int v)
{
   int i;
   reach[v]=1;
        for(i=1;i<=n;i++)
            if(a[v][i] && !reach[i])
                {
                    printf("\n %d->%d",v,i);
                    dfs(i);
                }
}
void main()
{
 int i,j,count=0;
```

```
clrscr();
printf("Depth First Search");
printf("\n Enter number of vertices:");
scanf("%d",&n);
for(i=1;i<=n;i++)
      {
            reach[i]=0;
            for(j=1;j<=n;j++)
            a[i][j]=0;
     }
printf("\n Enter the adjacency matrix:\n");
for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
  scanf("%d",&a[i][j]);
dfs(1);
printf("\n");
for(i=1;i<=n;i++)
{
 if(reach[i])
  count++;
}
if(count==n)
        printf("\n Graph is connected");
```

```
 else
        printf("\n Graph is not connected");
 getch();
}
```

7. **BINARY TREE TRAVERSAL**

```
struct Node
{
    int data;
    struct Node* left, *right;
    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
}
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);
```

```
    // now deal with the node
    cout << node->data << " ";
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
```

```
    cout << node->data << " ";

    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    struct Node *root = new Node(1);
    root->left            = new Node(2);
    root->right         = new Node(3);
    root->left->left     = new Node(4);
    root->left->right = new Node(5);

    cout << "\nPreorder traversal of binary tree is \n";
    printPreorder(root);

    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);

    cout << "\nPostorder traversal of binary tree is \n";
    printPostorder(root);
```

```
    return 0;
}
```

Output:

```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```