

UNIT-1

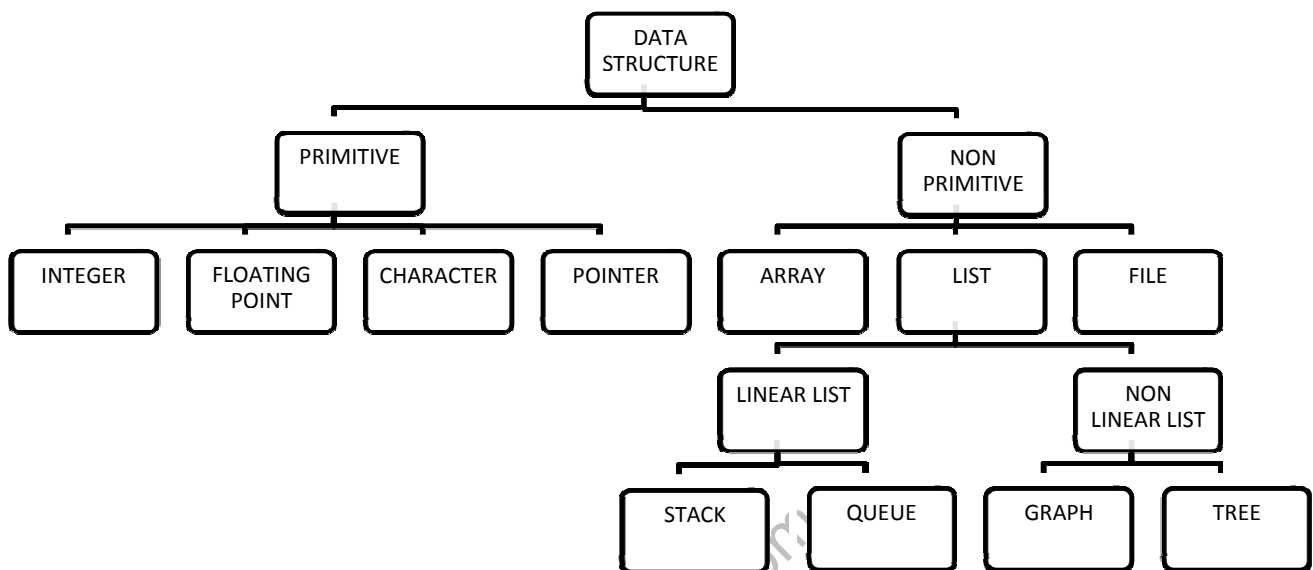
Introduction to Data Structure

- Computer is an electronic machine which is used for data processing and manipulation.
- When programmer collects such type of data for processing, he would require to store all of them in computer's main memory.
- In order to make computer work we need to know
 - Representation of data in computer.
 - Accessing of data.
 - How to solve problem step by step.
- For doing this task we use data structure.

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**.
- The storage structure representation in auxiliary memory is called as **file structure**.
- It is defined as the way of storing and manipulating data in organized form so that it can be used efficiently.
- Data Structure mainly specifies the following four things
 - Organization of Data
 - Accessing methods
 - Degree of associativity
 - Processing alternatives for information
- Algorithm + Data Structure = Program
- Data structure study covers the following points
 - Amount of memory require to store.
 - Amount of time require to process.
 - Representation of data in memory.
 - Operations performed on that data.

Classification of Data Structure



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.

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.
 - A tree can be viewed as restricted graph.
 - Graphs have many types:

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

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. |

Operation on Data Structures

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

1. 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.

2. Destroy

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

3. Selection

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

4. Updation

It updates or modifies the data in the data structure.

5. 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.

6. 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.

7. Merging

Merging is a process of combining the data items of two different sorted list into a single sorted list.

8. Splitting

Splitting is a process of partitioning single list to multiple list.

9. Traversal

Traversal is a process of visiting each and every node of a list in systematic manner.

Time and space analysis of algorithms

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.

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.

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 $(n+1)$.

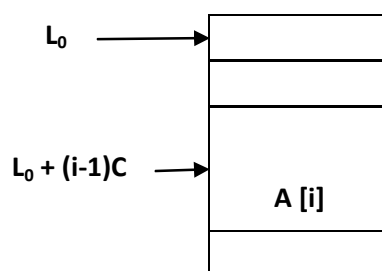
Best Case Analysis

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.

Explain Array in detail

One Dimensional Array

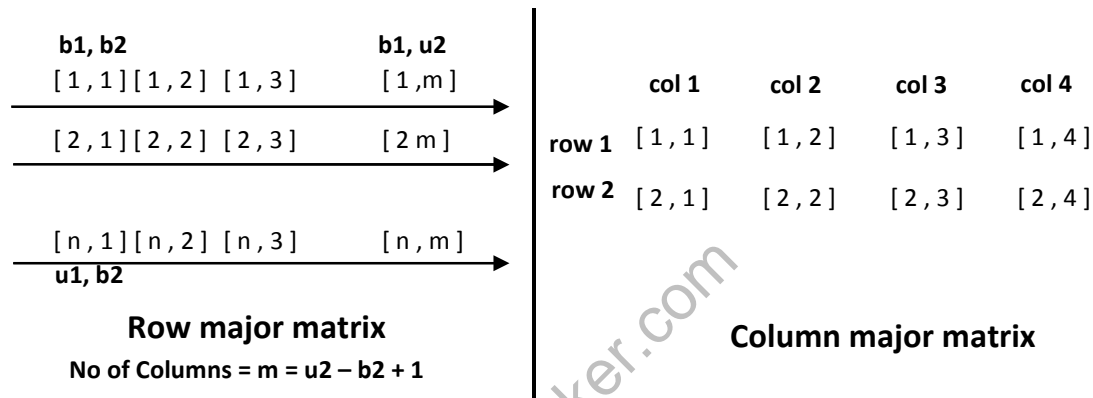
- Simplest data structure that makes use of computed address to locate its elements is the one-dimensional array or vector; number of memory locations is sequentially allocated to the vector.
- A vector size is fixed and therefore requires a fixed number of memory locations.
- Vector A with subscript lower bound of "one" is represented as below....



- L_0 is the address of the first word allocated to the first element of vector A .
- C words are allocated for each element or node
- The address of A_i is given equation **Loc (A_i) = $L_0 + C (i-1)$**
- Let's consider the more general case of representing a vector A whose lower bound for its subscript is given by some variable b . The location of A_i is then given by **Loc (A_i) = $L_0 + C (i-b)$**

Two Dimensional Array

- Two dimensional arrays are also called table or matrix, two dimensional arrays have two subscripts
- Two dimensional array in which elements are stored column by column is called as column major matrix
- Two dimensional array in which elements are stored row by row is called as row major matrix
- First subscript denotes number of rows and second subscript denotes the number of columns
- Two dimensional array consisting of two rows and four columns as above Fig is stored sequentially by columns : $A[1, 1]$, $A[2, 1]$, $A[1, 2]$, $A[2, 2]$, $A[1, 3]$, $A[2, 3]$, $A[1, 4]$, $A[2, 4]$
- The address of element $A[i, j]$ can be obtained by expression $\text{Loc}(A[i, j]) = L_0 + (j-1)*2 + i-1$
- In general for two dimensional array consisting of n rows and m columns the address element $A[i, j]$ is given by $\text{Loc}(A[i, j]) = L_0 + (j-1)*n + (i-1)$
- In row major matrix, array can be generalized to arbitrary lower and upper bound in its subscripts, assume that $b_1 \leq i \leq u_1$ and $b_2 \leq j \leq u_2$



- For row major matrix : $\text{Loc}(A[i, j]) = L_0 + (i - b_1) * (u_2 - b_2 + 1) + (j - b_2)$

Applications of Array

- Symbol Manipulation (matrix representation of polynomial equation)
- Sparse Matrix

Symbol Manipulation using Array

- We can use array for different kind of operations in polynomial equation such as addition, subtraction, division, differentiation etc...
- We are interested in finding suitable representation for polynomial so that different operations like addition, subtraction etc... can be performed in efficient manner
- Array can be used to represent Polynomial equation
- Matrix Representation of Polynomial equation**

| | Y | Y^2 | Y^3 | Y^4 |
|-------|--------|----------|----------|----------|
| X | XY | XY^2 | XY^3 | XY^4 |
| X^2 | X^2Y | X^2Y^2 | X^2Y^3 | X^2Y^4 |
| X^3 | X^3Y | X^3Y^2 | X^3Y^3 | X^3Y^4 |
| X^4 | X^4Y | X^4Y^2 | X^4Y^3 | X^4Y^4 |

- Once we have algorithm for converting the polynomial equation to an array representation and another algorithm for converting array to polynomial equation, then different operations in array (matrix) will be corresponding operations of polynomial equation

What is sparse matrix? Explain

- An $m \times n$ matrix is said to be sparse if “many” of its elements are zero.
- A matrix that is not sparse is called a dense matrix.
- We can devise a simple representation scheme whose space requirement equals the size of the non-zero elements.

Example:-

- The non-zero entries of a sparse matrix may be mapped into a linear list in row-major order.
- For example the non-zero entries of 4×8 matrix of below fig.(a) in row major order are 2, 1, 6, 7, 3, 9, 8, 4, 5

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 2 | 0 | 0 | 1 | 0 |
| 0 | 6 | 0 | 0 | 7 | 0 | 0 | 3 |
| 0 | 0 | 0 | 9 | 0 | 8 | 0 | 0 |
| 0 | 4 | 5 | 0 | 0 | 0 | 0 | 0 |

Fig (a) 4×8 matrix

| Terms | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|--------|---|---|---|---|---|---|---|---|---|
| Row | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 4 |
| Column | 4 | 7 | 2 | 5 | 8 | 4 | 6 | 2 | 3 |
| Value | 2 | 1 | 6 | 7 | 3 | 9 | 8 | 4 | 5 |

Fig (b) Linear Representation of above matrix

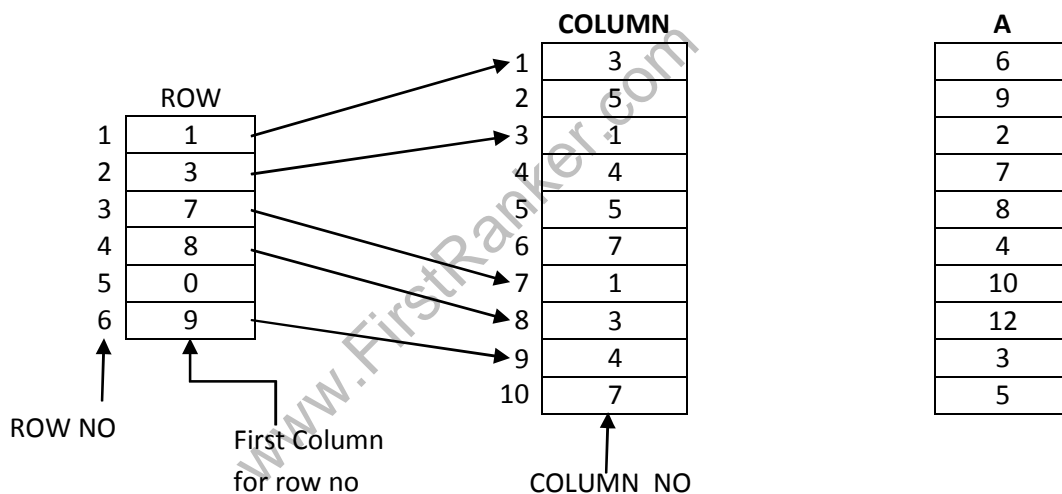
- To construct matrix structure we need to record
 - Original row and columns of each non zero entries
 - No of rows and columns in the matrix
- So each element of the array into which the sparse matrix is mapped need to have three fields: row, column and value
- A corresponding amount of time is saved creating the linear list representation over initialization of two dimension array.

$$A = \begin{bmatrix} 0 & 0 & 6 & 0 & 9 & 0 & 0 \\ 2 & 0 & 0 & 7 & 8 & 0 & 4 \\ 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 5 \end{bmatrix}$$

- Here from $6 \times 7 = 42$ elements, only 10 are non zero. $A[1,3]=6$, $A[1,5]=9$, $A[2,1]=2$, $A[2,4]=7$, $A[2,5]=8$, $A[2,7]=4$, $A[3,1]=10$, $A[4,3]=12$, $A[6,4]=3$, $A[6,7]=5$.
- One basic method for storing such a sparse matrix is to store non-zero elements in one dimensional array and to identify each array elements with row and column indices fig (c).

| | ROW | COLUMN | A |
|----|-----|--------|----|
| 1 | 1 | 3 | 6 |
| 2 | 1 | 5 | 9 |
| 3 | 2 | 1 | 2 |
| 4 | 2 | 4 | 7 |
| 5 | 2 | 5 | 8 |
| 6 | 2 | 7 | 4 |
| 7 | 3 | 1 | 10 |
| 8 | 4 | 3 | 12 |
| 9 | 6 | 4 | 3 |
| 10 | 6 | 7 | 5 |

Fig (c)

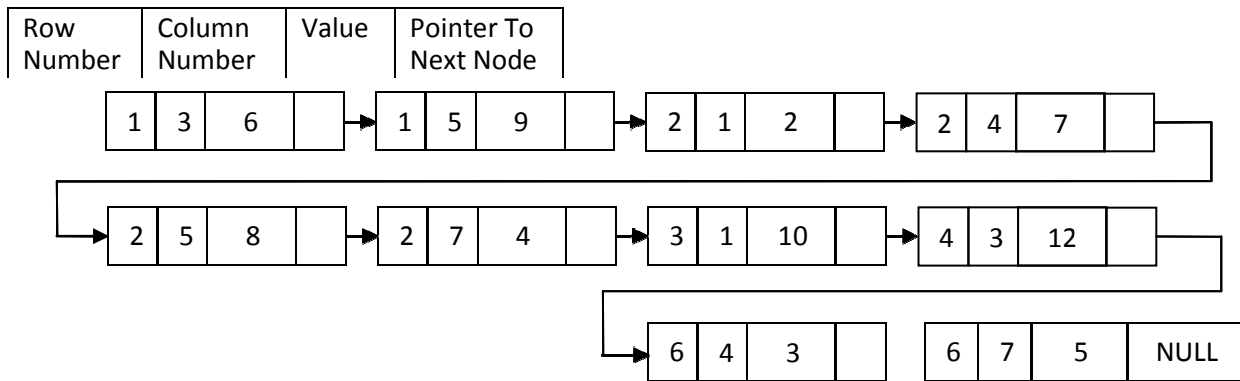


Fig(d)

- A more efficient representation in terms of storage requirement and access time to the row of the matrix is shown in fig (d). The row vector changed so that its i^{th} element is the index to the first of the column indices for the element in row i of the matrix.

Linked Representation of Sparse matrix

Typical node to represent non-zero element is



What is Stack?

- It is type of **linear data structure**.
- It follows **LIFO** (Last In First Out) property.
- It has only one pointer **TOP** that points the last or top most element of Stack.
- Insertion and Deletion in stack can only be done from top only.
- Insertion in stack is also known as a **PUSH operation**.
- Deletion from stack is also known as **POP operation** in stack.

Stack Implementation

- Stack implementation using array.
- Stack implementation using linked list.

Applications of Stack

- Conversion of polish notations
There are three types of notations:
 - > Infix notation - Operator is between the operands : $x + y$
 - > Prefix notation - Operator is before the operands : $+ xy$
 - > Postfix notation - Operator is after the operands : $xy +$
- To reverse a string
A string can be reversed by using stack. The characters of string pushed on to the stack till the end of the string. The characters are popped and displays. Since the end character of string is pushed at the last, it will be printed first.

- **When function (sub-program) is called**

When a function is called, the function is called last will be completed first. It is the property of stack. There is a memory area, specially reserved for this stack.

```
#include <iostream>
using namespace std;
#define MAX 5
class Stack
{
    private:
        int top;
        int ele[MAX];
    public:
        Stack();
        int  isFull();
        int  isEmpty();
        void push(int item);
        int  pop(int *item);
};
//Initialization of stack
Stack:: Stack()
{
    top = -1;
}
//Check stack is full or not
int Stack:: isFull()
{
    int full = 0;

    if( top == MAX-1 )
        full = 1;
    return full;
}
//Check stack is empty or not
int Stack:: isEmpty()
{
    int empty = 0;

    if( top == -1 )
        empty = 1;
    return empty;
}
//Insert item into stack
void Stack:: push( int item )
{
    if( isFull() )
    {
```

```
        cout<<"\nStack Overflow";
        return;
    }

    top++;
    ele[top] = item;

    cout<<"\nInserted item : "<< item;
}
//Delete item from stack
int Stack:: pop( int *item )
{
    if( isEmpty() )
    {
        cout<<"\nStack Underflow";
        return -1;
    }

    *item = ele[top--];
    return 0;
}
int main()
{
    int item = 0;
    Stack s = Stack();

    s.push( 10 );
    s.push( 20 );
    s.push( 30 );
    s.push( 40 );
    s.push( 50 );
    s.push( 60 );
    if( s.pop(&item) == 0 )
        cout<<"\nDeleted item : "<< item;
    if( s.pop(&item) == 0 )
        cout<<"\nDeleted item : "<< item;
    if( s.pop(&item) == 0 )
        cout<<"\nDeleted item : "<< item;
    if( s.pop(&item) == 0 )
        cout<<"\nDeleted item : "<< item;
    if( s.pop(&item) == 0 )
        cout<<"\nDeleted item : "<< item;
    if( s.pop(&item) == 0 )
        cout<<"\nDeleted item : "<< item;

    cout<< endl;
```



```
    return 0;  
}
```

Queue

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations on Queue:

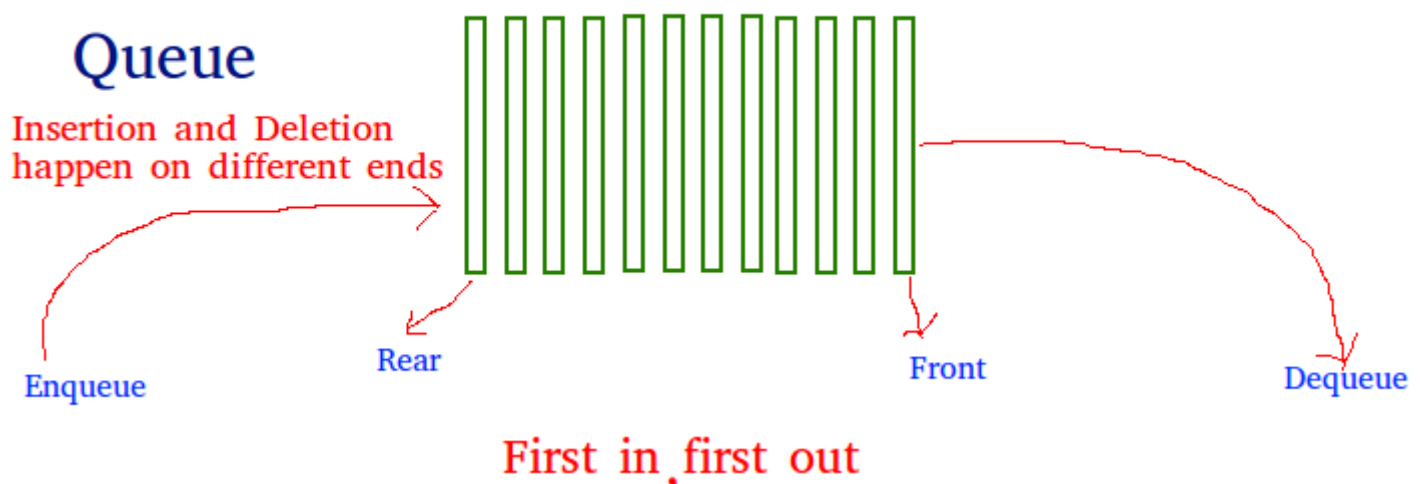
Mainly the following four basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front: Get the front item from queue.

Rear: Get the last item from queue.



Applications of Queue:

Queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See this for more detailed applications of Queue and Stack.

Array implementation Of Queue:

For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from front. If we simply increment front and rear indices, then there may be problems, front may reach end of the array. The solution to this problem is to increase front and rear in circular manner

Trace the conversion of infix to postfix form in tabular form.

(i) $(A + B * C / D - E + F / G / (H + I))$

| Input Symbol | Content of stack | Reverse polish | Rank |
|--------------|------------------|-----------------------------------|------|
| | (| | 0 |
| (| ((| | 0 |
| A | ((| | 0 |
| + | ((+ | A | 1 |
| B | ((+ B | A | 1 |
| * | ((+ * | A B | 2 |
| C | ((+ * C | A B | 2 |
| / | ((+ / | A B C * | 2 |
| D | ((+ / D | A B C * | 2 |
| - | ((- | A B C * D / + | 1 |
| E | ((- E | A B C * D / + | 1 |
| + | ((+ | A B C * D / + E - | 1 |
| F | ((+ F | A B C * D / + E - | 1 |
| / | ((+ / | A B C * D / + E - F | 2 |
| G | ((+ / G | A B C * D / + E - F | 2 |
| / | ((+ / | A B C * D / + E - F G / | 2 |
| (| ((+ / (| A B C * D / + E - F G / | 2 |
| H | ((+ / (H | A B C * D / + E - F G / | 2 |
| + | ((+ / (+ | A B C * D / + E - F G / H | 3 |
| I | ((+ / (+ I | A B C * D / + E - F G / H | 3 |
|) | ((+ / | A B C * D / + E - F G / H I + | 3 |
|) | (| A B C * D / + E - F G / H I + / + | 1 |
|) | | A B C * D / + E - F G / H I + / + | 1 |

Postfix expression is: **A B C * D / + E - F G / H I + / +**

(ii) **(A + B) * C + D / (B + A * C) + D**

| Input Symbol | Content of stack | Reverse polish | Rank |
|--------------|------------------|----------------|------|
| | (| | 0 |
| (| ((| | 0 |
| A | ((A | | 0 |
| + | ((+ | A | 1 |
| B | ((+ B | A | 1 |
|) | (| A B + | 1 |
| * | (* | A B + | 1 |
| C | (* C | A B + | 1 |
| + | (+ | A B + C * | 1 |
| D | (+ D | A B + C * | 1 |
| / | (+ / | A B + C * D | 2 |
| (| (+ / (| A B + C * D | 2 |
| B | (+ / (B | A B + C * D | 2 |
| + | (+ / (+ | A B + C * D B | 3 |
| A | (+ / (+ A | A B + C * D B | 3 |

| | | | |
|---|---------------|-----------------------------|---|
| * | (+ / (+ * | $AB + C * DBA$ | 4 |
| C | (+ / (+ * C | $AB + C * DBA$ | 4 |
|) | (+ / | $AB + C * DBAC * +$ | 3 |
| + | (+ | $AB + C * DBAC * + / +$ | 1 |
| D | (+ D | $AB + C * DBAC * + / +$ | 1 |
|) | | $AB + C * DBAC * + / + D +$ | 1 |

Postfix expression is: **$AB + C * DBAC * + / + D +$**

www.FirstRanker.com

Convert the following string into prefix: $A-B/(C*D^E)$

Step-1 : reverse infix expression

$) E ^) D * C ((/ B - A$

Step-2 : convert '(' to ')' and ')' to '(' and append extra ')' at last

$(E ^ (D * C)) / B - A$

Step-3 : Now convert this string to postfix

| Input Symbol | Content of stack | Reverse polish | Rank |
|--------------|------------------|-------------------|------|
| | (| | 0 |
| (| ((| | 0 |
| E | ((E | | 0 |
| ^ | ((^ | E | 1 |
| (| ((^ (| E | 1 |
| D | ((^ (D | E | 1 |
| * | ((^ (* | E D | 2 |
| C | ((^ (* C | E D | 2 |
|) | ((^ | E D C * | 2 |
|) | (| E D C * ^ | 1 |
| / | (/ | E D C * ^ | 1 |
| B | (/ B | E D C * ^ | 1 |
| - | (- | E D C * ^ B / | 1 |
| A | (- A | E D C * ^ B / | 1 |
|) | | E D C * ^ B / A - | 1 |

Step 4 : Reverse this postfix expression

$- A / B ^ * C D E$

Write an algorithm for evaluation of postfix expression and evaluation the following expression showing every status of stack in tabular form.

(i) $5\ 4\ 6\ +\ *\ 4\ 9\ 3\ /\ +\ *$ (ii) $7\ 5\ 2\ +\ *\ 4\ 1\ 1\ +\ /\ -$

Algorithm: EVALUAE_POSTFIX

- Given an input string POSTFIX representing postfix expression. This algorithm is going to evaluate postfix expression and put the result into variable VALUE. A vector S is used as a stack PUSH and POP are the function used for manipulation of stack. Operand2 and operand1 are temporary variable TEMP is used for temporary variable NEXTCHAR is a function which when invoked returns the next character. PERFORM_OPERATION is a function which performs required operation on OPERAND1 AND OPERAND2.

1. [Initialize stack and value]

TOP \leftarrow 1

VALUE \leftarrow 0

2. [Evaluate the prefix expression]

Repeat until last character

TEMP \leftarrow NEXTCHAR (POSTFIX)

If TEMP is DIGIT

Then PUSH (S, TOP, TEMP)

Else OPERAND2 \leftarrow POP (S, TOP)

OPERAND1 \leftarrow POP (S, TOP)

VALUE \leftarrow PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)

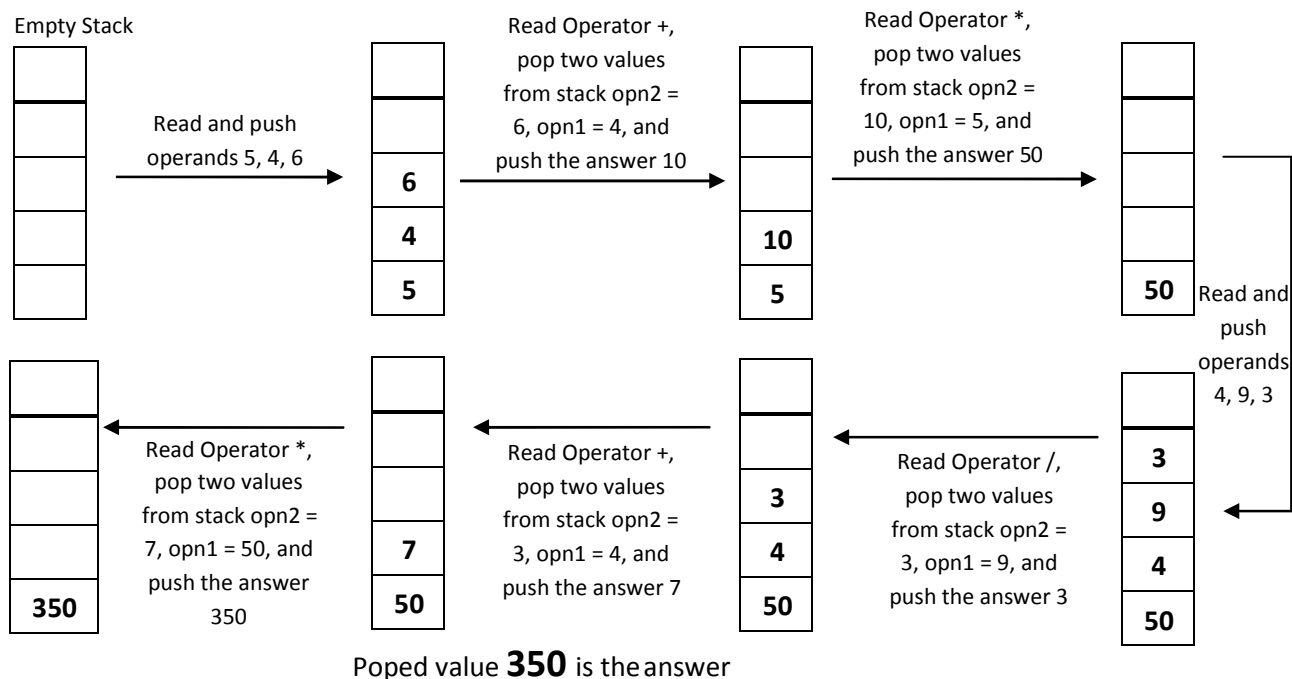
PUSH (S, TOP, VALUE)

3. [Return answer from stack]

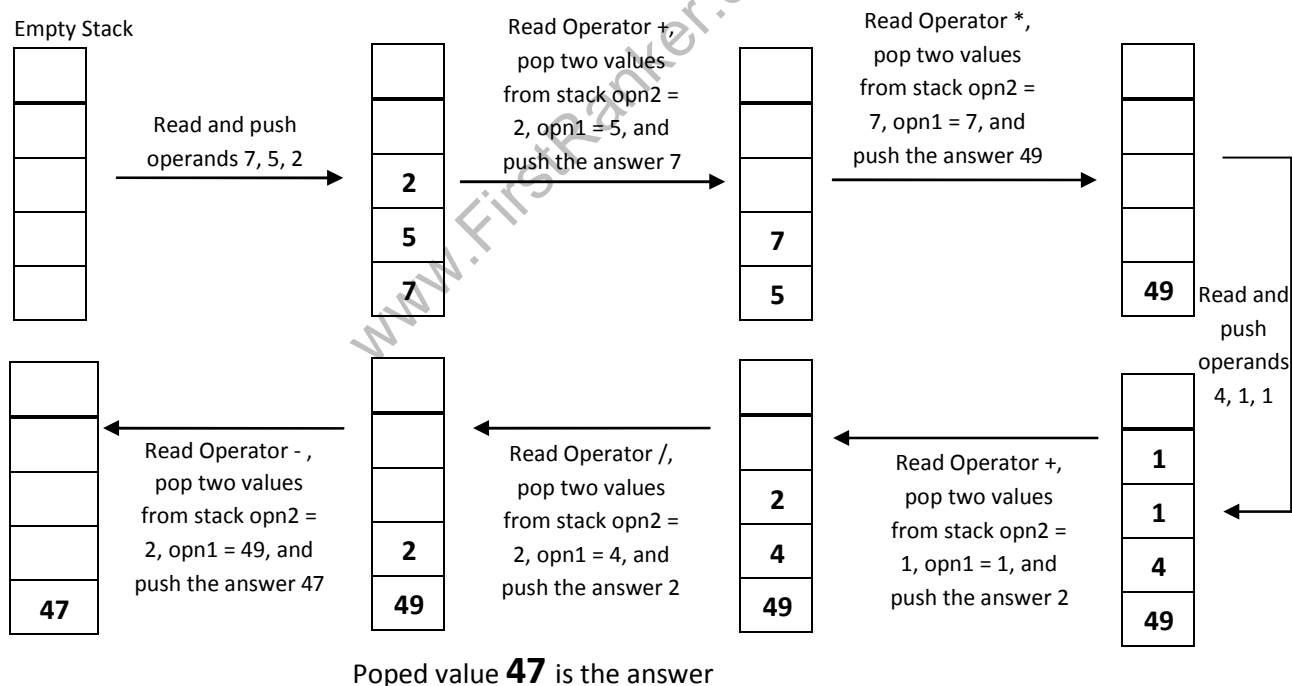
Return (POP (S, TOP))

www.FirstRanker.com

Evaluate (i): $5\ 4\ 6\ +\ *\ 4\ 9\ 3\ /\ +\ *$

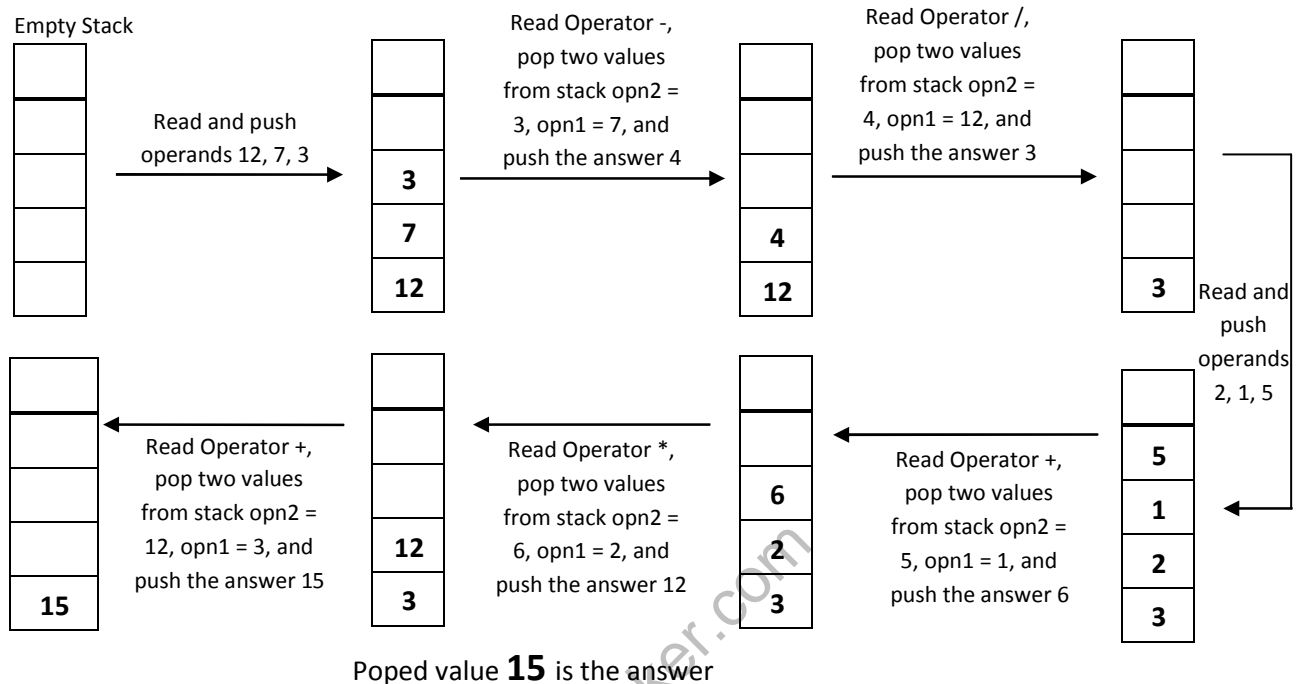


Evaluate (ii): $*\ 7\ 5\ 2\ +\ *\ 4\ 1\ 1\ +\ /\ -$



Consider the following arithmetic expression P, written in postfix notation. Translate it in infix notation and evaluate. P: 12, 7, 3, -, /, 2, 1, 5, +, *, +

Same Expression in infix notation is : $(12 / (7 - 3)) + ((5 + 1) * 2)$



Explain Difference between Stack and Queue.

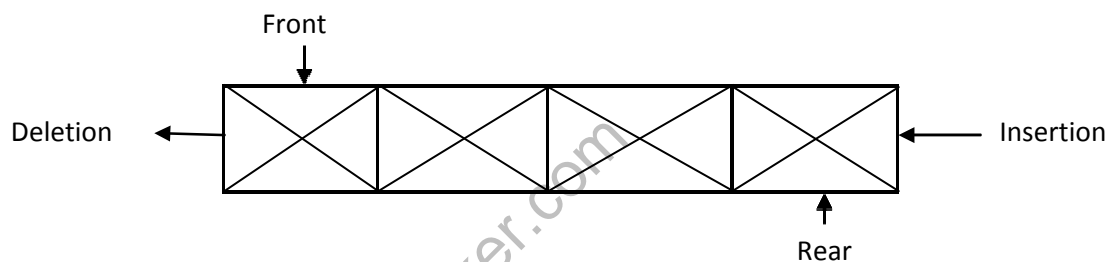
| Stack | Queue |
|--|---|
| A Linear List Which allows insertion or deletion of an element at one end only is called as Stack | A Linear List Which allows insertion at one end and deletion at another end is called as Queue |
| Since insertion and deletion of an element are performed at one end of the stack, the elements can only be removed in the opposite order of insertion. | Since insertion and deletion of an element are performed at opposite end of the queue, the elements can only be removed in the same order of insertion. |
| Stack is called as Last In First Out (LIFO) List. | Queue is called as First In First Out (FIFO) List. |
| The most and least accessible elements are called as TOP and BOTTOM of the stack | Insertion of element is performed at FRONT end and deletion is performed from REAR end |
| Example of stack is arranging plates in one above one. | Example is ordinary queue in provisional store. |
| Insertion operation is referred as PUSH and deletion operation is referred as POP | Insertion operation is referred as ENQUEUE and deletion operation is referred as DQUEUE |
| Function calling in any languages uses Stack | Task Scheduling by Operating System uses queue |

Explain following:

(i) Queue (ii) Circular Queue (iii) DQUEUE (iv) Priority Queue

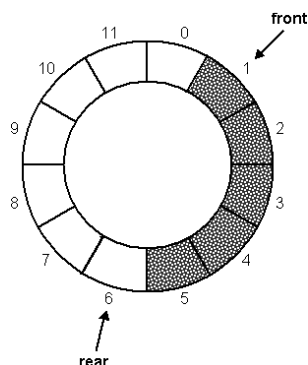
(i) Queue

- A linear list which permits deletion to be performed at one end of the list and insertion at the other end is called queue.
- The information in such a list is processed FIFO (first in first out) or FCFS (first come first served) pattern.
- Front is the end of queue from that deletion is to be performed.
- Rear is the end of queue at which new element is to be inserted.
- The process to add an element into queue is called **Enqueue**
- The process of removal of an element from queue is called **Dequeue**.
- The familiar and traditional example of a queue is Checkout line at Supermarket Cash Register where the first person in line is usually the first to be checked out.



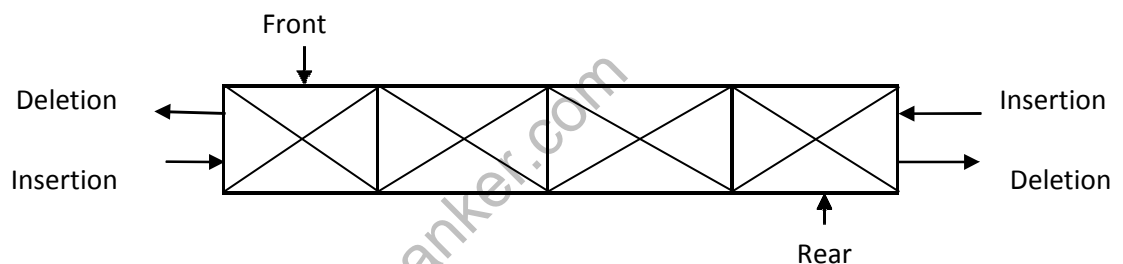
(ii) Circular Queue

- A more suitable method of representing simple queue which prevents an excessive use of memory is to arrange the elements $Q[1], Q[2], \dots, Q[n]$ in a circular fashion with $Q[1]$ following $Q[n]$, this is called circular queue
- In a standard queue data structure re-buffering problem occurs for each **dequeue** operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue
- Circular queue is a linear data structure. It follows FIFO principle.
- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list follow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers point to the beginning of the array.
- It is also called as "Ring buffer".



(iii) Dequeue

- A dequeue (double ended queue) is a linear list in which insertion and deletion are performed from the either end of the structure.
- There are two variations of Dqueue
 - Input restricted dqueue- allows insertion at only one end
 - Output restricted dqueue- allows deletion from only one end
- Such a structure can be represented by following fig.



(iv) Priority Queue

- A queue in which we are able to insert remove items from any position based on some property (such as priority of the task to be processed) is often referred as priority queue.
- Below fig. represent a priority queue of jobs waiting to use a computer.
- Priorities of 1, 2, 3 have been attached with jobs of real time, online and batch respectively. Therefore if a job is initiated with priority i, it is inserted immediately at the end of list of other jobs with priorities i. Here jobs are always removed from the front of queue

UNIT-3

Linked List

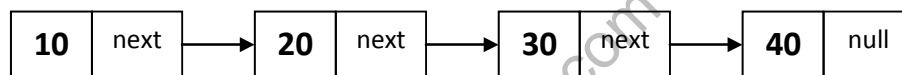
1. Linear Data Structure and their linked storage representation.

There are many applications where sequential allocation method is unacceptable because of following characteristics

- Unpredictable storage requirement
- Extensive manipulation of stored data

The linked allocation method of storage can result in both efficient use of computer storage and computer time.

- A linked list is a non-sequential collection of data items.
- The concept of a linked list is very simple, for every data item in the linked list, there is an associated pointer that would give the memory allocation of the next data item in the linked list.
- The data items in the linked list are not in a consecutive memory locations but they may be anywhere in memory.
- Accessing of these data items is easier as each data item contains within itself the address of the next data item.



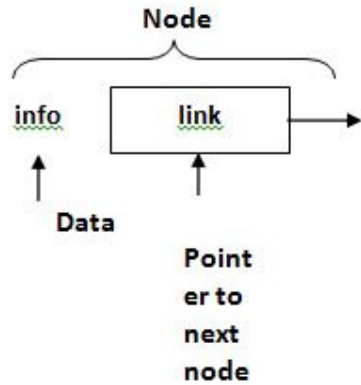
A Linked List

2. What is linked list? What are different types of linked list?

OR Write a short note on singly, circular and doubly linked list. OR

Advantages and disadvantages of singly, circular and doubly linked list.

- A linked list is a collection of objects stored in a list form.
- A linked list is a sequence of items (objects) where every item is linked to the next.
- A linked list is a non-primitive type of data structure in which each element is dynamically allocated and in which elements point to each other to define a linear relationship.
- Elements of linked list are called nodes where each node contains two things, data and pointer to next node.
- Linked list require more memory compared to array because along with value it stores pointer to next node.
- Linked lists are among the simplest and most common data structures. They can be used to implement other data structures like stacks, queues, and symbolic expressions, etc...



```
// C++ Structure to represent a node
class node
{
    int info
    struct node *link
};
```

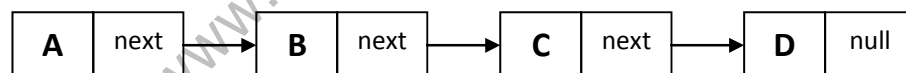
Operations on linked list

- Insert
 - Insert at first position
 - Insert at last position
 - Insert into ordered list
- Delete
- Traverse list (Print list)
- Copy linked list

Types of linked list

Singly Linked List

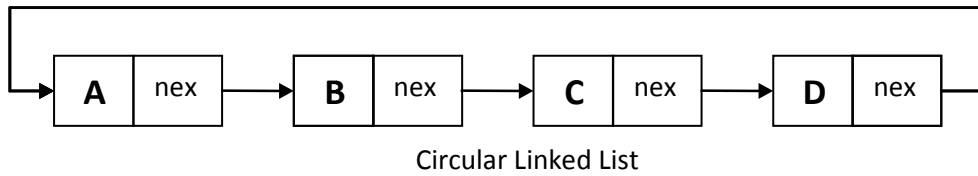
- It is basic type of linked list.
- Each node contains data and pointer to next node.
- Last node's pointer is null.
- Limitation of singly linked list is we can traverse only in one direction, forward direction.



Singly Linked List

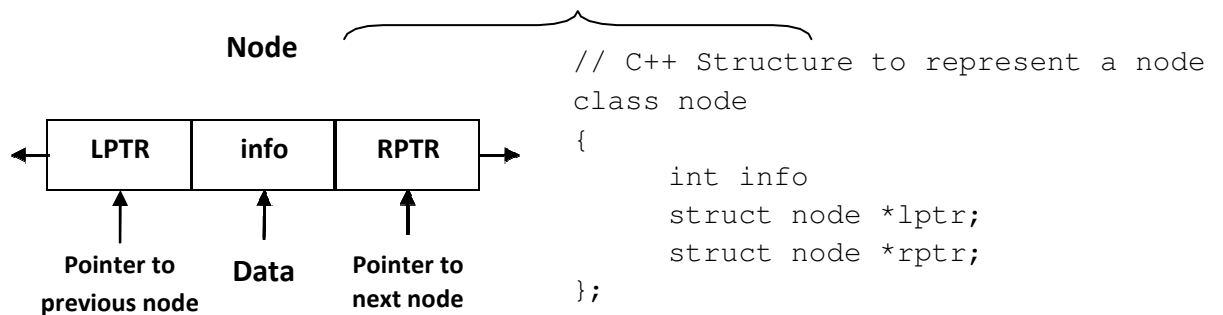
Circular Linked List

- Circular linked list is a singly linked list where last node points to first node in the list.
- It does not contain null pointers like singly linked list.
- We can traverse only in one direction that is forward direction.
- It has the biggest advantage of time saving when we want to go from last node to first node, it directly points to first node.
- A good example of an application where circular linked list should be used is a timesharing problem solved by the operating system.



Doubly Linked list

- Each node of doubly linked list contains data and two pointers to point previous (LPTR) and next (RPTR) node.



3. Discuss advantages and disadvantages of linked list over array.

Advantages of an array

1. We can access any element of an array directly means random access is easy
2. It can be used to create other useful data structures (queues, stacks)
3. It is light on memory usage compared to other structures

Disadvantages of an array

1. Its size is fixed
2. It cannot be dynamically resized in most languages
3. It is hard to add/remove elements
4. Size of all elements must be same.
5. Rigid structure (Rigid = Inflexible or not changeable)

Advantages of Linked List

1. **Linked lists are dynamic data structures:** That is, they can grow or shrink during execution of a program.
2. **Efficient memory utilization:** Here memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (free) when it is no longer needed.
3. **Insertion and deletions are easier and efficient:** Linked list provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. **Elements of linked list are flexible:** It can be primary data type or user defined data types

Disadvantages of Linked List

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2. It cannot be easily sorted
3. We must traverse 1/2 the list on average to access any element
4. More complex to create than an array
5. Extra memory space for a pointer is required with each element of the list

3. What are the advantages and disadvantages of stack and queue implemented using linked list over array?

Advantages and disadvantages of stack & queue implemented using linked list over array is described below,

Insertion & Deletion Operation

- Insertion and deletion operations are known as push and pop operation in stack and as insert and delete operation in queue.
- In the case of an array, if we have n-elements list and it is required to insert a new element between the first and second element then n-1 elements of the list must be moved so as to make room for the new element.
- In case of linked-list, this can be accomplished by only interchanging pointers.
- Thus, insertion and deletions are more efficient when performed in linked list than array.

Searching a node

- If a particular node in a linked list is required, it is necessary to follow links from the first node onwards until the desired node is found.
- Where as in the case of an array, directly we can access any node

Join & Split

- We can join two linked list by assigning pointer of second linked list in the last node of first linked list.
- Just assign null address in the node from where we want to split one linked list in two parts.
- Joining and splitting of two arrays is much more difficult compared to linked list.

Memory

- The pointers in linked list consume additional memory compared to an array

Size

- Array is fixed sized so number of elements will be limited in stack and queue.
- Size of linked list is dynamic and can be changed easily so it is flexible in number of elements

Insertion and deletion operations in Array and Linked-List

UNIT-1

the example?

for Transpose of the Sparse Matrix?

DT?

UNIT-2

the example C++ program for the container class?

ed list?

ked list?

ne expression from infix to postfix using Stack?

rogramme?

rogramme?

UNIT-3

d perform the insertion and deletion operation on it.

rogramme?

ed list?

ed with Templates?

TREES

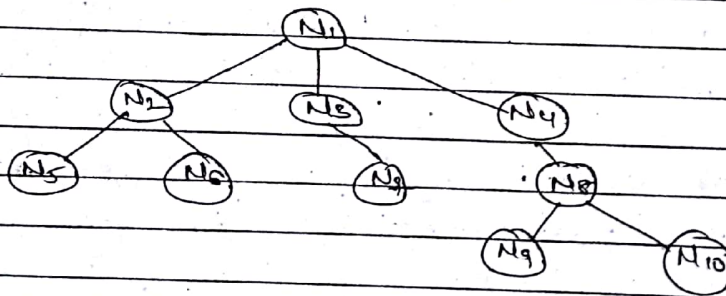
Definition:- A Tree is a non linear data structure which consist of finite set of one or more nodes such that, there is a specially designated node called "root".

The remaining nodes are partitioned into $n \geq 0$ disjoint sets where T_1, T_2, \dots, T_n are called sub tree of the root.

* Every node in the tree is the root of some subtree.

Note:- A Tree structure means that the data are organized so the items are related by branches. The representation of the tree is "hierarchical representation".

Representation of A Tree:-



Some terminologies:-

node:- The data + branch representation in a tree is known as node.

degree:- The number of subtree of a node

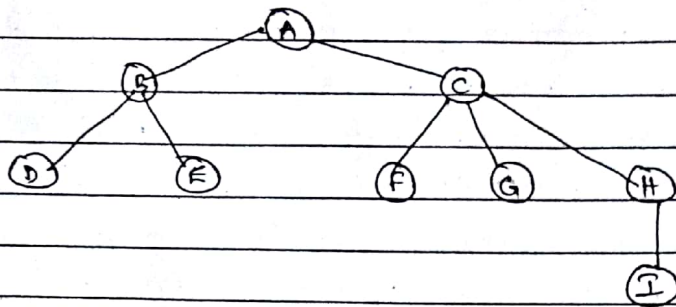
degree of a tree:- The total number of levels present in a node is known as degree of a tree

leaf nodes:- The number of the degree of the node is "zero" Such node is known as leaf node.

Siblings:- The children of same parent are said to be siblings.

Ancestor nodes:- All the nodes along the path from the root to that node.

Example:-



A is the root node

B is the parent node of D, E

B and C are siblings

D and E are children of B

D, E, F, G and I are leaf nodes.

A, B, C, H are internal nodes

The level of E is 3

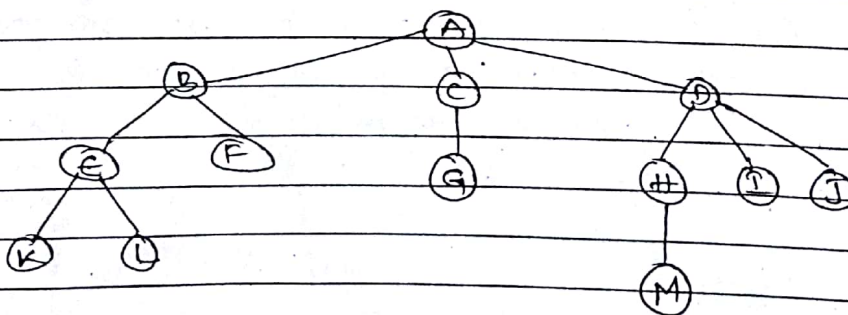
Height/depth of tree is 4.

The ancestors of nodes I is A, C and H.

The descendants of nodes C is F, G, H and I

Representation of Trees:-

→ List representation:- The root comes first followed by a list of subtrees.



The generalized representation of a tree

$(A (B (E (K, L), F), C (G), D (H (M), I, J)))$

node structure of a tree

| data | |
|------------|-------------|
| left child | right child |

2

Binary Tree :-

Binary trees are characterized by the fact that any node can have at most two sub nodes.

* Definition :- A binary tree is a finite set of nodes that is either empty or consists of root and two disjoint binary trees called "left subtree" and the "right subtree".

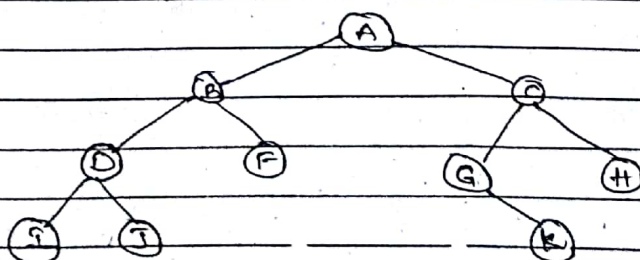
Properties of a Binary tree :-

- * Level n has at most 2^n nodes
- * A full binary tree of height h has $[2^{h+1} - 1]$ nodes
- * In a binary tree, the number of external nodes is "1" more than the number of internal nodes.
- * The number of external nodes satisfies $(h+1) \leq e \leq 2^h$
- * The no. of internal nodes satisfies $h \leq i \leq 2^{h-1}$
- * The total no. of nodes (n) satisfies $2^{h+1} \leq n \leq 2^{h+1} - 1$

Binary tree representation :-

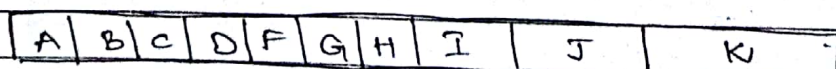
A binary tree data structure is represented using two methods. These representations are

1. Array representation
2. Linked List representation



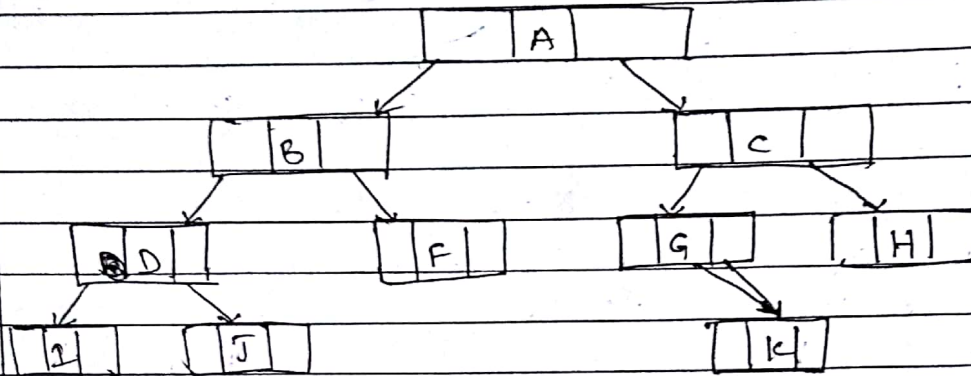
Array representation :-

We basically use one dimensional Array for representation



Linked List Representation :-

Here we use double linked list to represent the binary tree



Binary tree traversal :-

traversing means visiting all the nodes in the tree in certain order. to traverse a tree we basically use three type of traversings they are

- ① In order
- ② pre Order
- ③ post Order

In Order :-

1. Traverse the left subtree
2. Visit the root
3. Traverse the right subtree

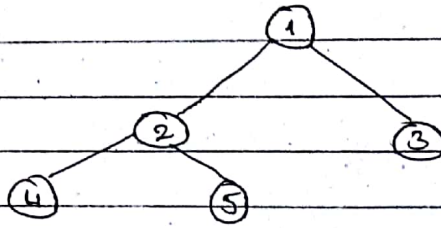
Pre Order :-

1. Visit the root node
2. Traverse the left subtree
3. Traverse the right subtree

Post order :-

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root node

Example:-



Inorder:- 4 2 5 1 3

Pre Order:- 1 2 4 5 3

Post Order:- 4 5 2 3 1

Threaded Binary Tree:-

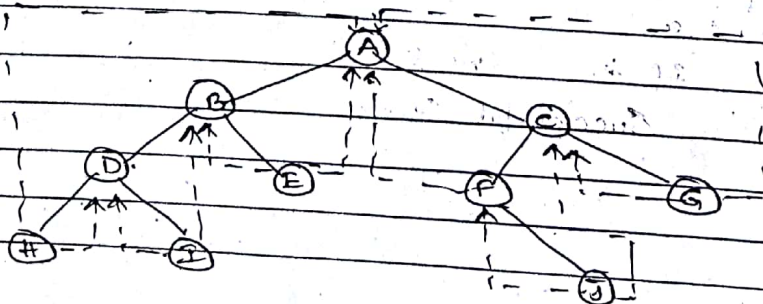
A binary tree is represented using array representation (8) linked list representation. When a binary tree is represented using linked list, if any node is not having a child we use "NULL" pointer in that position. In any binary tree representation if there are n nodes then there are $n+1$ number of Null nodes.

AJ Perlis and C. Thornton have proposed new binary tree called "Threaded Binary tree", which make use of NULL pointer to improve its traversal process. In threaded binary tree NULL pointers are replaced by references to other nodes called threads.

→ All the left child pointers that are NULL point to its in-order predecessor.

→ All the right child pointers that are NULL point to its in-order successor.

If there is no in-order predecessor or in-order successor, then it may point to root node.



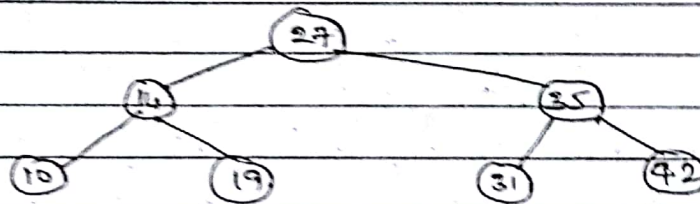
Binary Search Tree :-

A BST is a tree in which all the nodes follow the below mentioned properties

1. A BST Left Subtree has a Key less than or equal to its parent node's key
2. The Right Subtree of a node has a Key greater than to its parent node's Key

$$\Rightarrow \boxed{\text{Left node} < \text{Root} < \text{Right node}}$$

The BST is represented below



The basic operations of BST is

- * Insertion of elements
- * deletion of elements
- * Searching of elements

Searching of element :-

Step 1:- Select the key element to search

Step 2:- if the tree contains ~~any~~ no. nodes then the tree does not exist and search fails

Step 3:- if (key < root) to search in ^{left} ~~right~~ subtree

Step 4:- if (key > root) to search continues in right subtree

Step 5:- if key element is found in the tree then to display Successful search else the key not exist

```
def search_recursively (key, node)
    if node is None or node.key == key:
        return node
    elif key < node.key:
        return search_recursively (key, node.left);
    else
        return search_recursively (key, node.right);
```

Insertion:-

```
Node * insert (Node *& root, int key, int value)
{
    if (!root)
        root = newNode (key, value);
    else if (key < root->key)
        root->left = insert (root->left, key, value);
    else
        root->right = insert (root->right, key, value);
    return root;
}
```

Deletion:-

When removing a node it is mandatory to maintain in order sequence of the nodes.

- * Deleting a node with no childrens
- * Deleting a node with one childrens
- * Deleting a node with two childrens

Heap:-

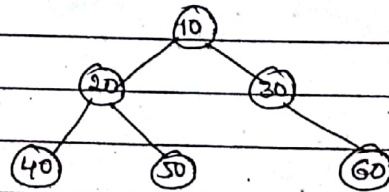
A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types

1. The min-heap property: The value of each node is greater than or equal to the value of its parent, with the

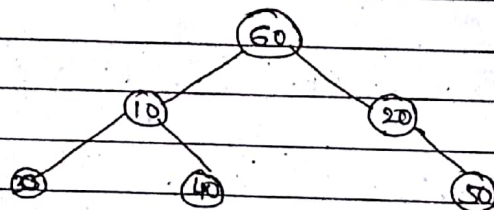
minimum-value element at the root

2. The max-heap property: The value of each node is less than or equal to the value of its parent, with the maximum value element at the root.

Examples:-



Min-Heap



Max-Heap

Inserting a node in a heap:-

The node is basically added at the end of the heap. The heap is repaired by comparing the added element with its parent and moving the added element up a level. This process is called "percolation up".

public void insert (Comparable x)

{

if (Size == heap.length - 1) double Size();

int pos == ++Size;

for (pos > 1 && x.compareTo(heap[pos/2]) < 0; pos = pos/2)

heap[pos] = heap[pos/2];

heap[pos] = x;

}

Deleting a node from heap:-

Consider a minimum heap tree, then minimum element can be found at the root, which is the first element of the array. We remove the root

and replace it with the last element of the heap and then restore the heap property by percolating down.

Priority Queue:-

This is Useful for processing elements based on some priority. The elements of priority queue must be comparable to each other, The priority Queue operations are done by binary heap.

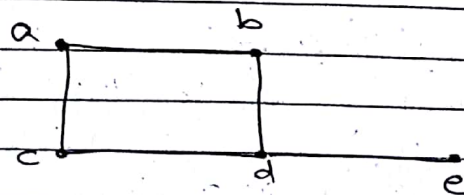
== x = x ==

UNIT - 5

GRAPHS

A Graph is a non-linear data structure which consist of set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as "vertices" and the links that connected the vertices are called "edges".

Formally a graph is a pair of sets (V, E) , where V is set of vertices and E is a set of edges.



$$V = \{a, b, c, d, e\}$$

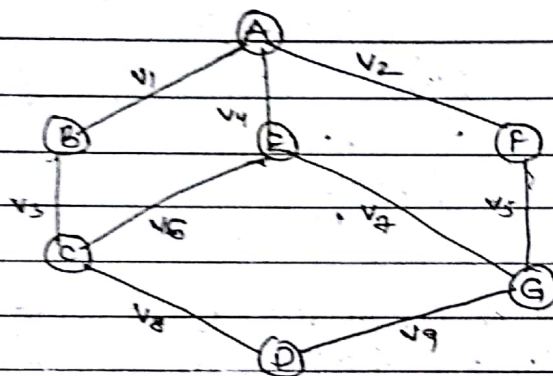
$$E = \{ab, ac, bd, cd, de\}$$

Vertex:- Each node of a graph is represented as a vertex. In the below example the circle represents the vertex. Here A, B, C, D, E, F, G are the vertices of A Graph.

Edge :- The edge is the connection in between two vertices. In the below example vertices A and B are joined together with an edge e_1 .

Adjacency :-

Two nodes or vertices are adjacent if they are connected to each other through the edge. In the below example B is adjacent to A, C is adjacent to B and so on.



Basic Operations :-

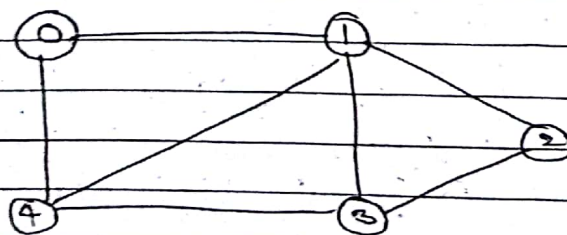
The following are the basic operations on a graph

- Add vertex
- Add edge
- Delete vertex

representations of a graph :-

The following are the most common representations of a graph

- Adjacency matrix
- Adjacency list



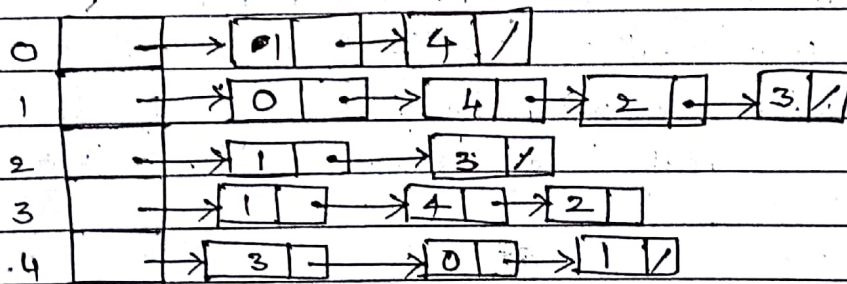
Adjacency Matrix:-

This is an 2D array of size $V \times V$ is the number of vertices in a graph. The slot $a[i][j] = 1$ indicates that there is an edge from vertex i to vertex j .

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

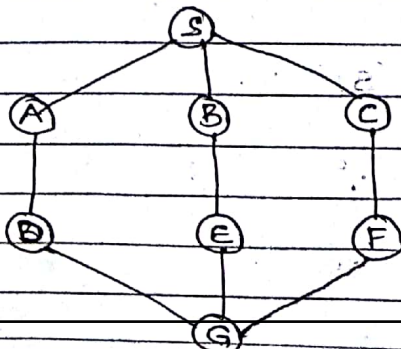
Adjacency List:-

An array of linked list is used. Size of an array is equal to no. of vertices. This representation is also be used to represent a weighted graph. The weight of edge can be stored in nodes of linked list.



Graph Traversal:-

DFS (Depth First Search) :- This algorithm traverses a graph in a depth first motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

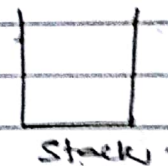
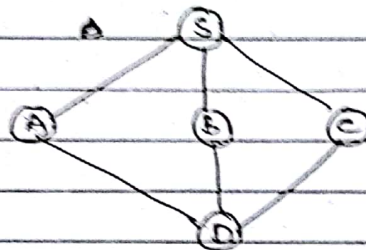


Rule 1:- Visit the adjacent unvisited vertex. Mark it as visited. Display. push it in a stack.

Rule 2:- If no adjacent vertex is found, pop up a vertex from the stack.

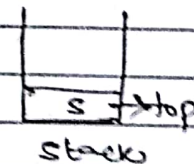
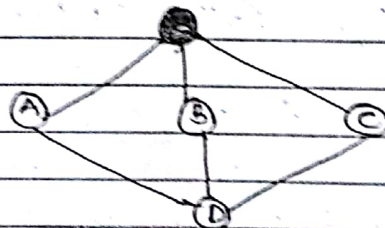
Rule 3:- Repeat Rule 1 and Rule 2 until the stack is empty.

Step 1:-

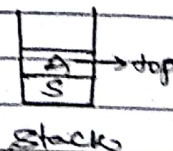
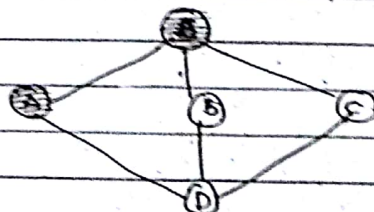


Initialize the stack

Step 2:- Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. we have three nodes and we can pick any of them. For this example we next have to traverse A or B or C.

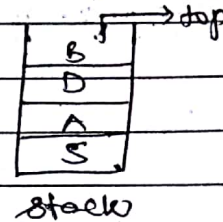
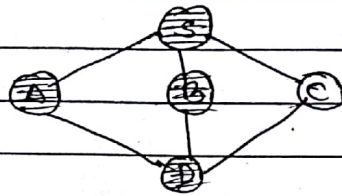


Step 3:- Then visit vertex 'A' and put it onto the stack. Explore any unvisited vertex 'D'.



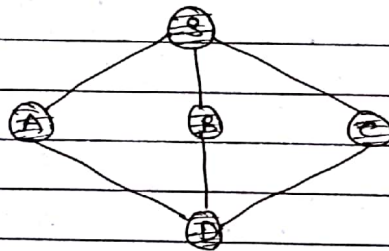
Step 4:-

Visit D and mark it as visited and put it on the stack. next we have B and C as unvisited vertex then visit B



Step 5:-

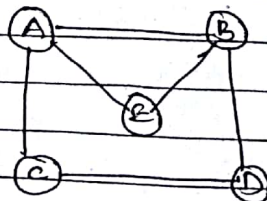
Only the unvisited vertex is 'C' then visit the node and put it on to the stack



The visited vertex are popped from the stack and make the stack empty $C \rightarrow B \rightarrow D \rightarrow A \rightarrow S$

BFS (Breadth First Search)

Here to trace this algorithm we basically use Queue data structure. the below example shows how the bfs algorithm is used for traversing a graph. The Queue follows the mechanism of "FIFO"



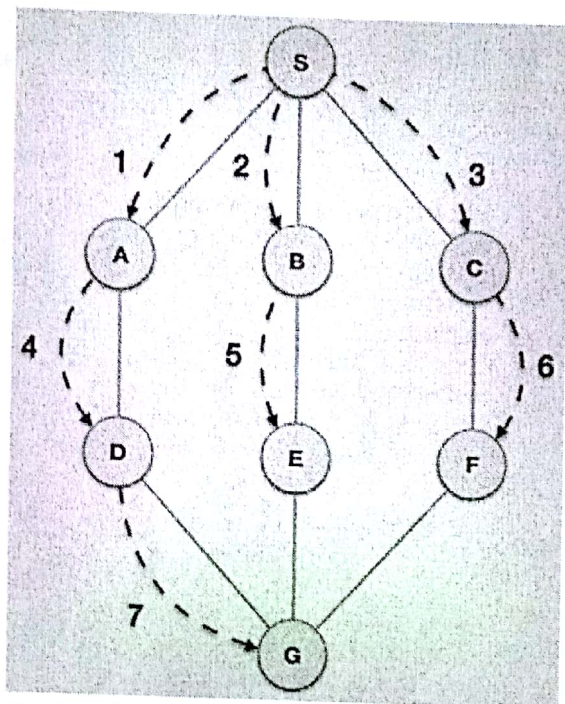
First of all start from any of the vertex

and mark that vertex as starting vertex.

step 1 - first select the vertex A as starting one.
the adjacency vertex

Breadth First Search

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

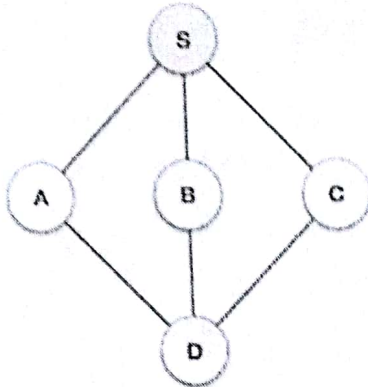


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

| Step | Traversal | Description |
|------|-----------|-----------------------|
| 1. | | Initialize the queue. |

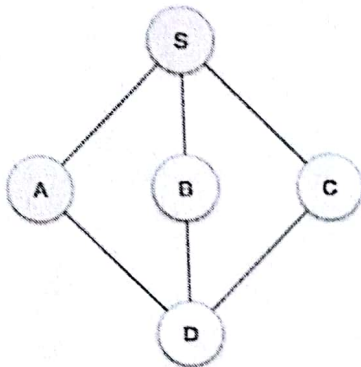
2.



Queue

We start from visiting **S**(starting node), and mark it as visited.

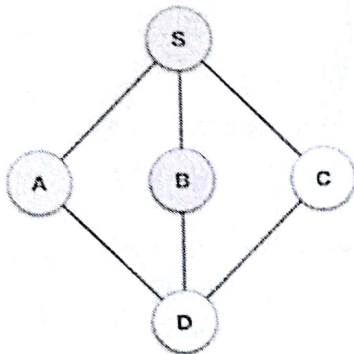
3.



A _____
Queue

We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

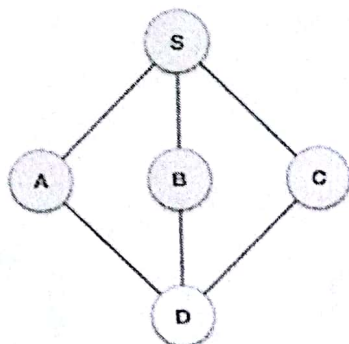
4.



B **A** _____
Queue

Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

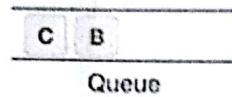
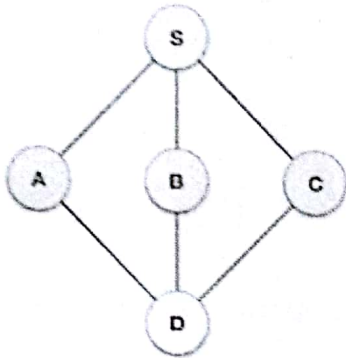
5.



C **B** **A** _____
Queue

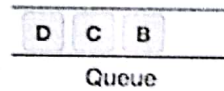
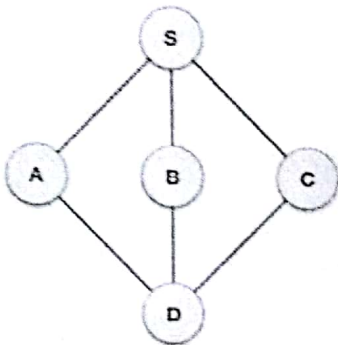
Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6.



Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

7.



From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

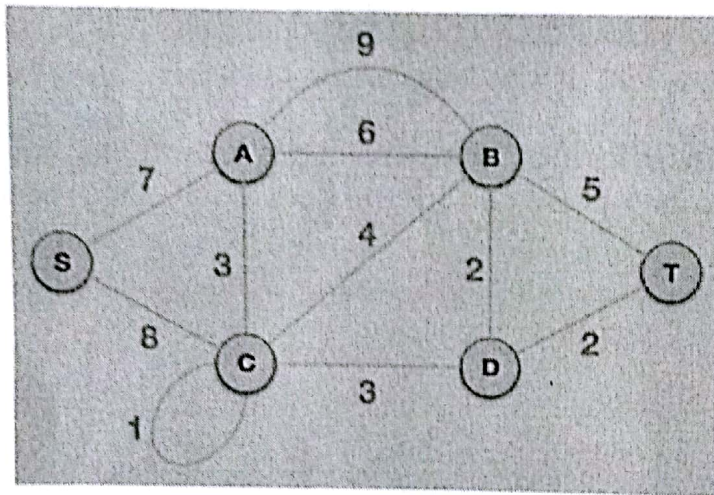
The implementation of this algorithm in C programming language can be seen here

Prim's Spanning Tree Algorithm

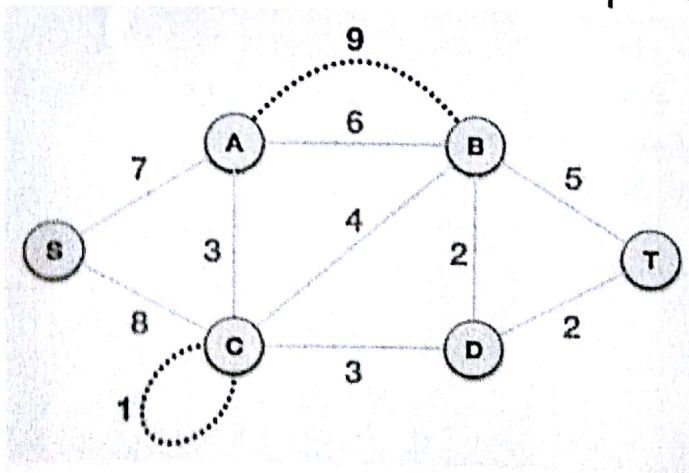
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

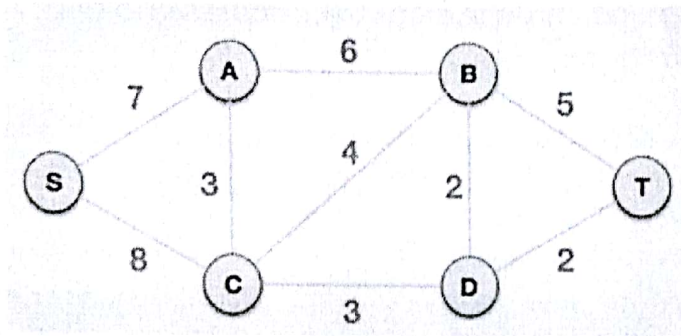
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

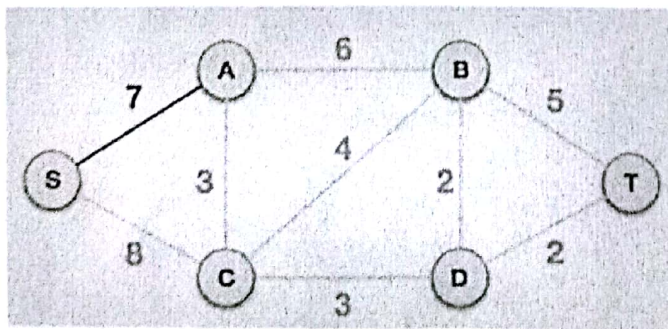


Step 2 - Choose any arbitrary node as root node

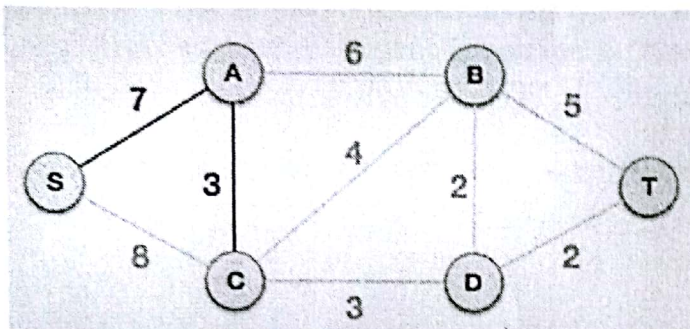
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

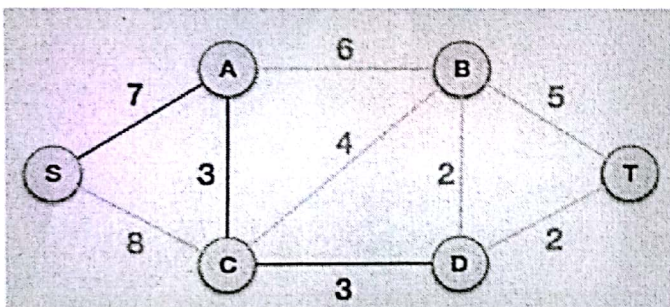
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



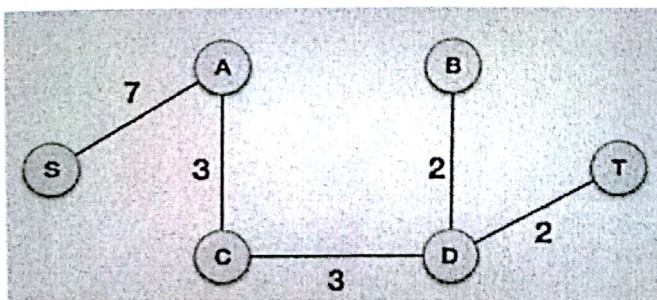
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

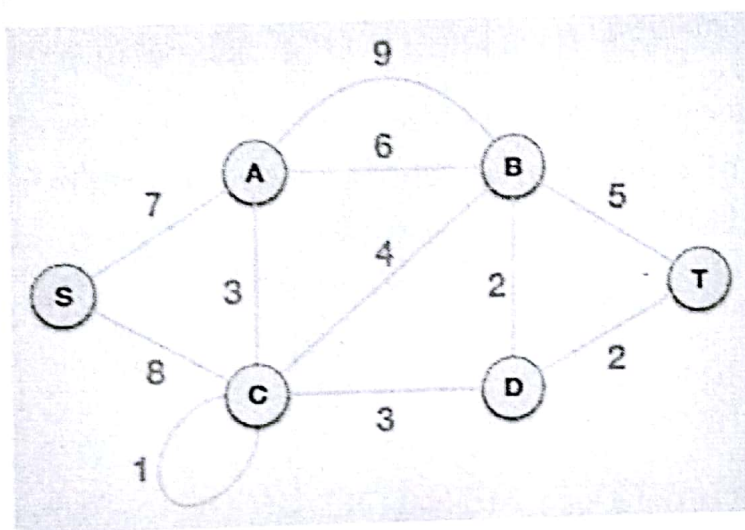


We may find that the output spanning tree of the same graph using two different algorithms is same.

Kruskal's Spanning Tree Algorithm

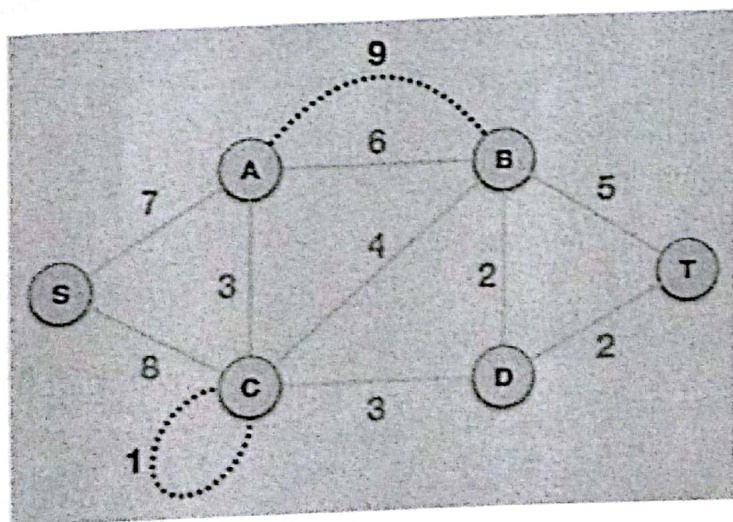
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example –

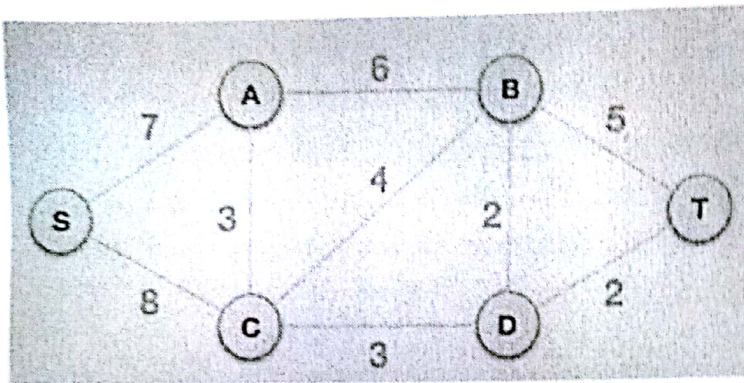


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



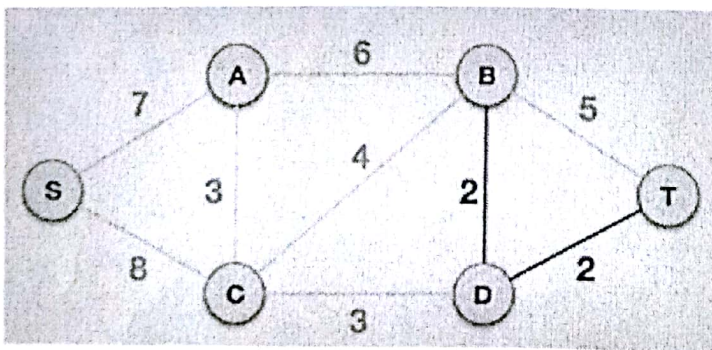
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

| B, D | D, T | A, C | C, D | C, B | B, T | A, B | S, A | S, C |
|------|------|------|------|------|------|------|------|------|
| 2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 |

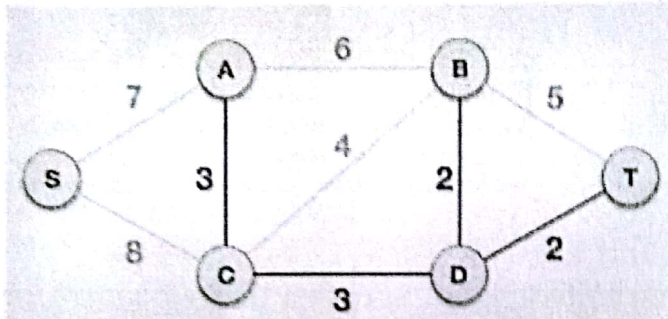
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

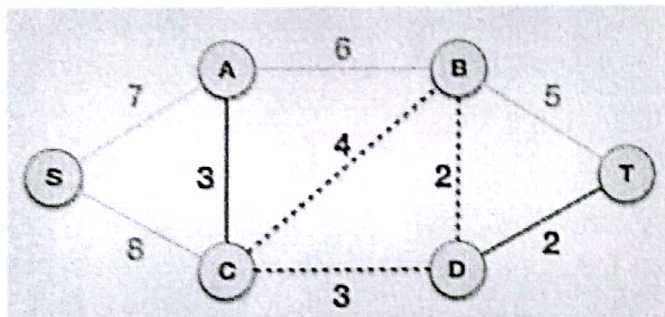


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

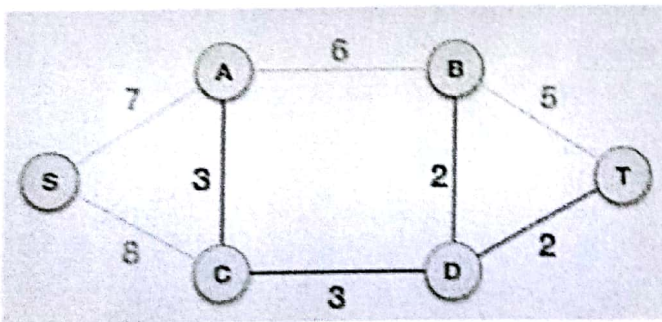
Next cost is 3, and associated edges are A,C and C,D. We add them again –



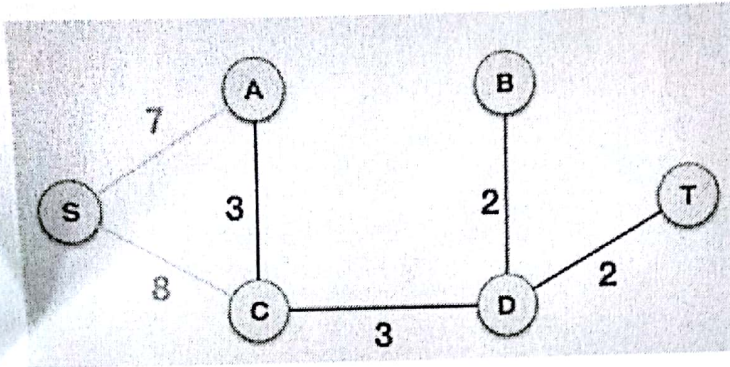
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



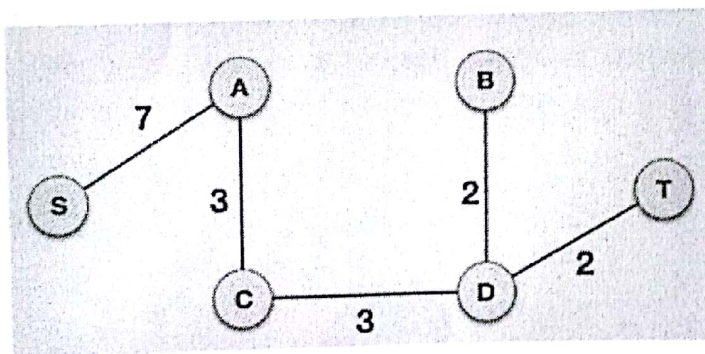
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

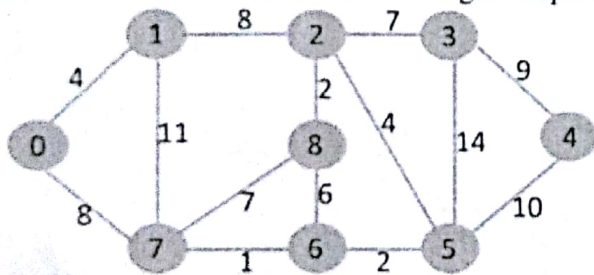
Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

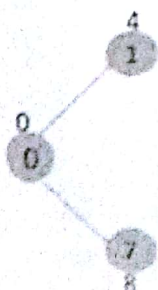
Algorithm

- 1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While *sptSet* doesn't include all vertices
 -a) Pick a vertex *u* which is not there in *sptSet* and has minimum distance value.
 -b) Include *u* to *sptSet*.
 -c) Update distance value of all adjacent vertices of *u*. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex *v*, if sum of distance value of *u* (from source) and weight of edge *u-v*, is less than the distance value of *v*, then update the distance value of *v*.

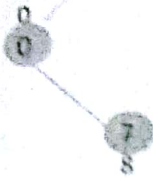
Let us understand with the following example:



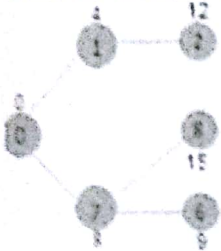
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



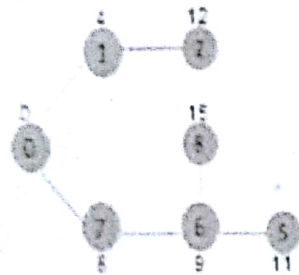
Pick the vertex with minimum distance value and not already included in SPT (not in *sptSet*). The vertex 1 is picked and added to *sptSet*. So *sptSet* now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



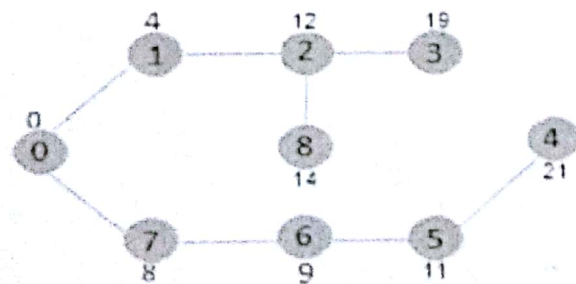
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until sptSet doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



How to implement the above algorithm?

Recommended: Please solve it on "**PRACTICE**" first, before moving on to the solution.

We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store shortest distance values of all vertices.

SOLINS ALGORITHM

Solins algorithm is an algorithm for finding a minimum spanning tree in a graph for which all edge weights are distinct, or a minimum spanning forest in the case of a graph that is not connected.

It was first published in 1926 by Otakar Borůvka as a method of constructing an efficient electricity network for Moravia. The algorithm was rediscovered by Choquet in 1938 again by Florek, Łukasiewicz, Perkal, Steinhaus, and Zubrzycki in 1951; and again by Sollin in 1965. Because Sollin was the only computer scientist in this list living in an English speaking country, this algorithm is frequently called **Sollin's algorithm**, especially in the parallel computing literature.

The algorithm begins by finding the minimum-weight edge incident to each vertex of the graph, and adding all of those edges to the forest. Then, it repeats a similar process of finding the minimum-weight edge from each tree constructed so far to a different tree, and adding all of those edges to the forest. Each repetition of this process reduces the number of trees, within each connected component of the graph, to at most half of this former value, so after logarithmically many repetitions the process finishes. When it does, the set of edges it has added forms the minimum spanning forest.

Designating each vertex or set of connected vertices a "component", pseudocode Solions algorithm is:

Input: A graph G whose edges have distinct weights

Initialize a forest F to be a set of one-vertex trees, one for each vertex of the graph.

While F has more than one component:

Find the connected components of F and label each vertex of G by its component

Initialize the cheapest edge for each component to "None"

For each edge uv of G :

If u and v have different component labels:

If uv is cheaper than the cheapest edge for the component of u :

Set uv as the cheapest edge for the component of u

If uv is cheaper than the cheapest edge for the component of v :

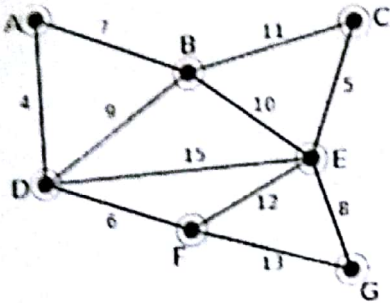
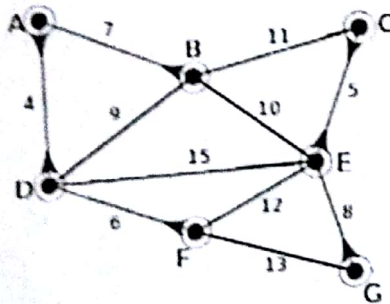
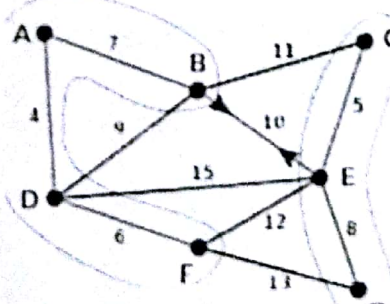
Set uv as the cheapest edge for the component of v

For each component whose cheapest edge is not "None":

Add its cheapest edge to F

Output: F is the minimum spanning forest of G .

Example

| Image | components | Description |
|---|--|--|
|  | <p>{A} {B} {C} {D} {E} {F} {G}</p> | <p>This is our original weighted graph. The numbers near the edges indicate their weight. Initially, every vertex by itself is a component (blue circles).</p> |
|  | <p>{A,B,D,F} {C,E,G}</p> | <p>In the first iteration of the outer loop, the minimum weight edge out of every component is added. Some edges are selected twice (AD, CE). Two components remain.</p> |
|  | <p>{A,B,C,D,E,F,G}</p> | <p>In the second and final iteration, the minimum weight edge out of each of the two remaining components is added. These happen to be the same edge. One component remains and we are done. The edge BD is not considered because both endpoints are in the same component.</p> |

UNIT- VI: Sorting: Bubble sort, Merge sort, Insertion Sort, Selection Sort, Quick Sort. Searching: Linear Search, Binary Search.
Introduction to Data Structures: Basics of Linear and Non-Linear Data structures.

UNIT VI:

1. Explain in detail about sorting and different types of sorting techniques

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order. Sorting can be classified in two types;

Internal Sorts:- This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting. There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints. There are 3 types of internal sorts.

(i) SELECTION SORT :- Ex:- Selection sort algorithm, Heap Sort algorithm

(ii) INSERTION SORT :- Ex:- Insertion sort algorithm, Shell Sort algorithm

(iii) EXCHANGE SORT :- Ex:- Bubble Sort Algorithm, Quick sort algorithm

External Sorts:- Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.

Ex:- Merge Sort

2. Write a program to explain bubble sort. Which type of technique does it belong. (b) What is the worst case and best case time complexity of bubble sort?

/* bubble sort implementation */

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,n,temp,j,arr[25];
    clrscr();
    printf("Enter the number of elements in the Array: ");
    scanf("%d",&n);
    printf("\nEnter the elements:\n\n");
    for(i=0 ; i<n ; i++)
    {
        printf(" Array[%d] = ",i);
        scanf("%d",&arr[i]);
    }
}
```



```

for(i=0 ; i<=n ; i++)
{
    for(j=0 ; j<=n-i-1 ; j++)
    {
        if(arr[j]>arr[j+1]) //Swapping Condition is Checked
        { temp=arr[j];
          arr[j]=arr[j+1];
          arr[j+1]=temp;
        }
    }
}
printf("\nThe Sorted Array is:\n\n");
for(i=0 ; i<=n ; i++)
{
    printf(" %4d",arr[i]);
}
getch();
}

```

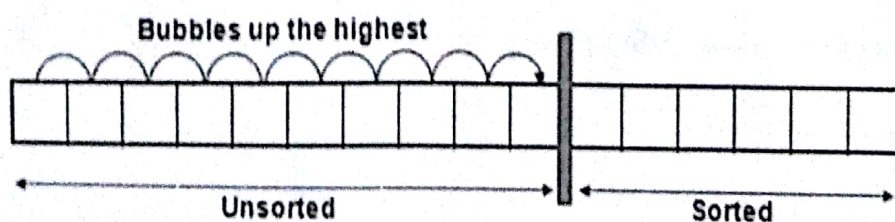
Time Complexity of Bubble Sort :

The complexity of sorting algorithm is depends upon the number of comparisons that are made. Total comparisons in Bubble sort is: $n(n-1)/2 \approx n^2 - n$

Best case : $O(n^2)$
 Average case : $O(n^2)$
 Worst case : $O(n^2)$

3. Explain the algorithm for bubble sort and give a suitable example. (OR) Explain the algorithm for exchange sort with a suitable example.

In bubble sort method the list is divided into two sub-lists sorted and unsorted. The smallest element is bubbled from unsorted sub-list. After moving the smallest element the imaginary wall moves one element ahead. The bubble sort was originally written to bubble up the highest element in the list. But there is no difference whether highest / lowest element is bubbled. This method is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. Given a list of 'n' elements the bubble sort requires up to n-1 passes to sort the data.



Algorithm for Bubble Sort: Bubble Sort (A [], N)

Step 1 : Repeat For P = 1 to N - 1 Begin

Step 2 : Repeat For J = 1 to N - P Begin

Step 3 : If (A [J] < A [J - 1])

 Swap (A [J], A [J - 1]) End For

 End For

Step 4 : Exit

Example:

Ex:- A list of unsorted elements are: 10 47 12 54 19 23

(Bubble up for highest value shown here)

| | | | | | |
|---------------|--------------|--------------|--------------|--------------|--------------|
| 10 | 54 | 54 | 54 | 54 | 54 |
| 47 | 10 | 47 | 47 | 47 | 47 |
| 12 | 47 | 10 | 23 | 23 | 23 |
| 54 | 12 | 23 | 10 | 19 | 19 |
| 19 | 23 | 12 | 19 | 10 | 12 |
| 23 | 19 | 19 | 12 | 12 | 10 |
| Original List | After Pass 1 | After Pass 2 | After Pass 3 | After Pass 4 | After Pass 5 |

A list of sorted elements now : 54 47 23 19 12 10

4. Show the bubble sort results for each pass for the following initial array of elements.
35 18 7 12 5 23 16 3 1

```

enter number of elements to be sorted:8
enter elements of array:35 18 7 12 5 23 16 31

The given list
35 18 7 12 5 23 16 31

iteration 1 18 7 12 5 23 16 31 35
iteration 2 7 12 5 18 16 23 31 35
iteration 3 7 5 12 16 18 23 31 35
iteration 4 5 7 12 16 18 23 31 35
iteration 5 5 7 12 16 18 23 31 35
iteration 6 5 7 12 16 18 23 31 35
iteration 7 5 7 12 16 18 23 31 35
iteration 8 5 7 12 16 18 23 31 35

The final sorted list
5 7 12 16 18 23 31 35
  
```

5. Write a program to explain insertion sort . Which type of technique does it belong.
 (or)

Write a C-program for sorting integers in ascending order using insertion sort.

/*Program to sort elements of an array using insertion sort method*/

```

#include<stdio.h>

#include<conio.h>

void main( )
{
    int a[10],i,j,k,n;

    clrscr( );

    printf("How many elements you want to sort?\n");

    scanf("%d",&n);

    printf("\nEnter the Elements into an array:\n");

    for (i=0;i<n;i++)
        scanf("%d",&a[i]);

    for(i=1;i<n;i++)
    {
        k=a[i];

        for(j= i-1; j>=0 && k<a[j]; j--)

            a[j+1]=a[j];
    }
  
```



```
a[j+1]=k;

} printf("\n\n Elements after sorting: \n");

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

    printf("%d\n", a[i]);

    getch( );

}
```

OUTPUT:

How many elements you want to sort ? : 6

Enter elements for an array : 78 23 45 8 32 36

After Sorting the elements are : 8 23 32 36 45 78

6. Explain the algorithm for insertion sort and give a suitable example.

Both the selection and bubble sorts exchange elements. But insertion sort does not exchange elements. In insertion sort the element is inserted at an appropriate place similar to card insertion. Here 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 position. Suppose we have 'n' elements, we need n-1 passes to sort the elements. Insertion sort works this way:

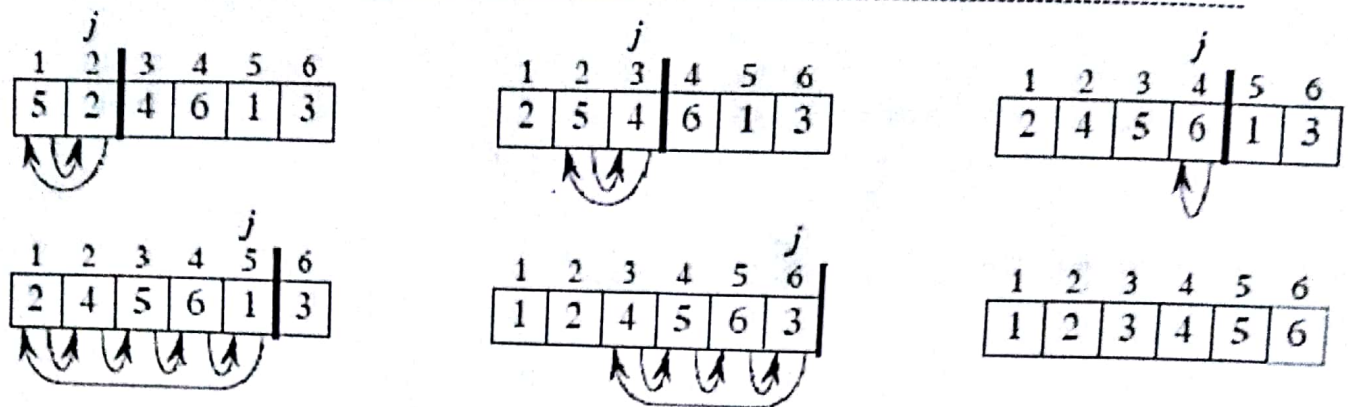
It works the way you might sort a hand of playing cards:

1. We start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].
2. Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].
3. To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.

INSERTION_SORT (A)

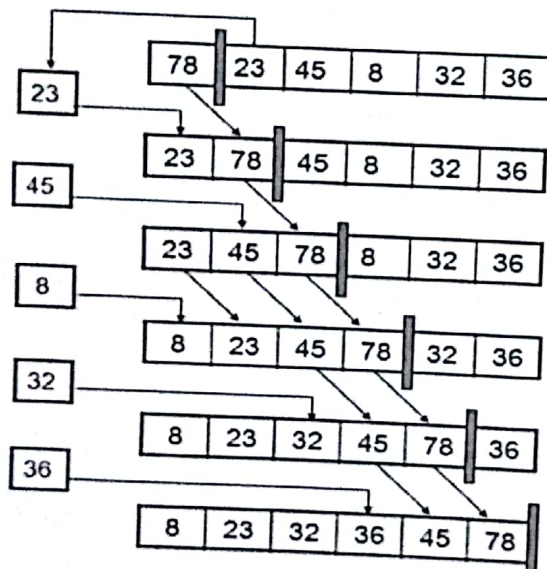
```
1.  FOR j ← 2 TO length[A]
2.      DO key ← A[j]
3.          {Put A[j] into the sorted sequence A[1 . . j - 1]}
4.          i ← j - 1
5.          WHILE i > 0 and A[i] > key
6.              DO A[i+1] ← A[i]
7.                  i ← i - 1
8.          A[i+1] ← key
```

Example: Following figure (from CLRS) shows the operation of INSERTION-SORT on the array $A = (5, 2, 4, 6, 1, 3)$. Each part shows what happens for a particular iteration with the value of j indicated. j indexes the "current card" being inserted into the hand.



Read the figure row by row. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right, and $A[j]$ moves into the evacuated position.

Ex:- A list of unsorted elements are: 78 23 45 8 32 36 . The results of insertion sort for each pass is as follows:-



A list of sorted elements now : 8 23 32 36 45 78

7. Demonstrate the insertion sort results for each insertion for the following initial array of elements
25 6 15 12 8 34 9 18 2

enter number of elements to be sorted:9

enter elements of array:25 6 15 12 8 34 9 18 2

The given list

25 6 15 12 8 34 9 18 2

| | | | | | | | | | |
|---------------|---|----|----|----|----|----|----|----|----|
| iteration 1 : | 6 | 25 | 15 | 12 | 8 | 34 | 9 | 18 | 2 |
| iteration 2 : | 6 | 15 | 25 | 12 | 8 | 34 | 9 | 18 | 2 |
| iteration 3 : | 6 | 12 | 15 | 25 | 8 | 34 | 9 | 18 | 2 |
| iteration 4 : | 6 | 8 | 12 | 15 | 25 | 34 | 9 | 18 | 2 |
| iteration 5 : | 6 | 8 | 12 | 15 | 25 | 34 | 9 | 18 | 2 |
| iteration 6 : | 6 | 8 | 9 | 12 | 15 | 25 | 34 | 18 | 2 |
| iteration 7 : | 6 | 8 | 9 | 12 | 15 | 18 | 25 | 34 | 2 |
| iteration 8 : | 2 | 6 | 8 | 9 | 12 | 15 | 18 | 25 | 34 |

The final sorted list

2 6 8 9 12 15 18 25 34

8. Demonstrate the selection sort results for each pass for the following initial array of elements
 . 21 6 3 57 13 9 14 18 2

enter number of elements to be sorted:9

enter elements of array:21 6 3 57 13 9 14 18 2

The given list

21 6 3 57 13 9 14 18 2

| | | | | | | | | | |
|---------------|----|---|---|----|----|----|----|----|----|
| iteration 1 : | 21 | 6 | 3 | 57 | 13 | 9 | 14 | 18 | 2 |
| iteration 2 : | 18 | 6 | 3 | 2 | 13 | 9 | 14 | 21 | 57 |
| iteration 3 : | 14 | 6 | 3 | 2 | 13 | 9 | 18 | 21 | 57 |
| iteration 4 : | 9 | 6 | 3 | 2 | 13 | 14 | 18 | 21 | 57 |
| iteration 5 : | 9 | 6 | 3 | 2 | 13 | 14 | 18 | 21 | 57 |
| iteration 6 : | 2 | 6 | 3 | 9 | 13 | 14 | 18 | 21 | 57 |
| iteration 7 : | 2 | 3 | 6 | 9 | 13 | 14 | 18 | 21 | 57 |
| iteration 8 : | 2 | 3 | 6 | 9 | 13 | 14 | 18 | 21 | 57 |

The final sorted list

2 3 6 9 13 14 18 21 57

How many elements you want to sort? : 5

Enter elements for an array : 2 6 4 8 5

After Sorting the elements are : 8 6 5 4 2

10. Explain the algorithm for selection sort and give a suitable example.

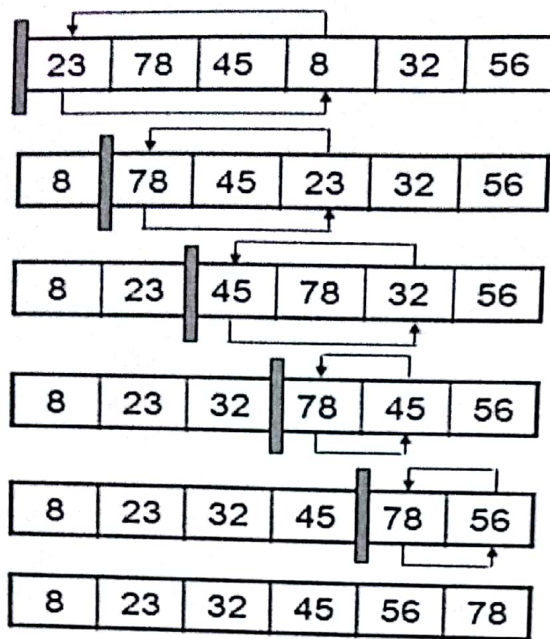
In selection sort the list is divided into two sub-lists sorted and unsorted. These two lists are divided by imaginary wall. We find a smallest element from unsorted sub-list and swap it to the beginning. And the wall moves one element ahead, as the sorted list is increases and unsorted list is decreases.

Assume that we have a list on n elements. By applying selection sort, the first element is compared with all remaining $(n-1)$ elements. The smallest element is placed at the first location. Again, the second element is compared with remaining $(n-1)$ elements. At the time of comparison, the smaller element is swapped with larger element. Similarly, entire array is checked for smallest element and then swapping is done accordingly. Here we need $n-1$ passes or iterations to completely rearrange the data.

Algorithm: Selection_Sort (A [], N)

```
Step 1 : Repeat For K = 0 to N - 2          Begin
Step 2 :   Set POS = K
Step 3 :   Repeat for J = K + 1 to N - 1      Begin
           If A[ J ] < A [ POS ]
           Set POS = J
           End For
Step 5 :   Swap A [ K ] with A [ POS ]
           End For
Step 6 :   Exit
```

Ex:- A list of unsorted elements are: 23 78 45 8 32 56



A list of sorted elements now : 8 23 32 45 56 78

11. Show the quick sort results for each exchange for the following initial array of elements
35 54 12 18 23 15 45 38

```
*****QUICK SORT*****
enter number of elements to be sorted:8
enter elements of array:35 54 12 18 23 15 45 38

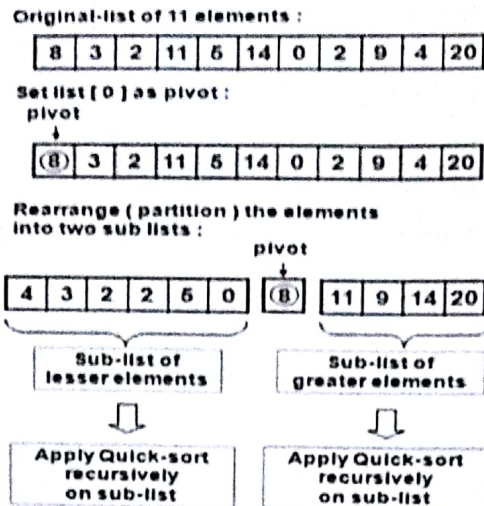
The given list
35  54  12  18  23  15  45  38
23  15  12  18  35  54  45  38
18  15  12  23  35  54  45  38
12  15  18  23  35  54  45  38
12  15  18  23  35  54  45  38
12  15  18  23  35  38  45  54

The final sorted list:
12  15  18  23  35  38  45  54
```

12. Explain the algorithm for QUICK sort (partition exchange sort) and give a suitable example.

Quick sort is based on partition. It is also known as partition exchange sorting. The basic concept of quick sort process is pick one element from an array and rearranges the remaining elements around it. This element divides the main list into two sub lists. This chosen element is called pivot. Once pivot is chosen, then it shifts all the elements less than pivot to left of value pivot and all the elements greater than pivot are shifted to the right side. This procedure of choosing pivot and partition the list is applied recursively until sub-lists consisting of only one element.

Ex:- A list of unsorted elements are: 8 3 2 11 5 14 0 2 9 4 20



Algorithm for quick sort:

It is also known as partition exchange sort. It was invented by CAR Hoare. It is based on partition. The basic concept of quick sort process is pick one element from an array and rearranges the remaining elements around it. This element divides the main list into two sub lists. This chosen element is called pivot. Once pivot is chosen, then it shifts all the elements less than pivot to left of value pivot and all the elements greater than pivot are shifted to the right side. This procedure of choosing pivot and partition the list is applied recursively until sub-lists consisting of only one element.

quicksort(q)

varlist less, pivotList, greater

if length(q) ≤ 1

return q

select a pivot value pivot from q

for each x in q except the pivot element

if x < pivot then add x to less

if x ≥ pivot then add x to greater

add pivot to pivotList

return concatenate(quicksort(less), pivotList, quicksort(greater))

Time Complexity of Quick sort:

Best case : $O(n \log n)$

Average case : $O(n \log n)$

Worst case : $O(n^2)$

Advantages of quick sort:

-
1. This is faster sorting method among all.
 2. Its efficiency is also relatively good.
 3. It requires relatively small amount of memory.

Disadvantages of quick sort:

1. It is complex method of sorting so, it is little hard to implement than other sorting methods.

13. Explain the algorithm for Merge sort and give a suitable example.

The basic concept of merge sort is divides the list into two smaller sub-lists of approximately equal size. Recursively repeat this procedure till only one element is left in the sub-list. After this, various sorted sub-lists are merged to form sorted parent list. This process goes on recursively till the original sorted list arrived.

Algorithm for merge sort:

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub-problems, we state each sub-problem as sorting a sub-array $A[p .. r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through sub-problems.

To sort $A[p .. r]$:

1. Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p .. r]$ into two sub-arrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

2. Conquer Step

Conquer by recursively sorting the two sub-arrays $A[p .. q]$ and $A[q + 1 .. r]$.

3. Combine Step

Combine the elements back in $A[p .. r]$ by merging the two sorted sub-arrays $A[p .. q]$ and $A[q + 1 .. r]$ into a sorted sequence. To accomplish this step, we will define a procedure $\text{MERGE}(A, p, q, r)$.

Note that the recursion bottoms out when the sub-array has just one element, so that it is trivially sorted.

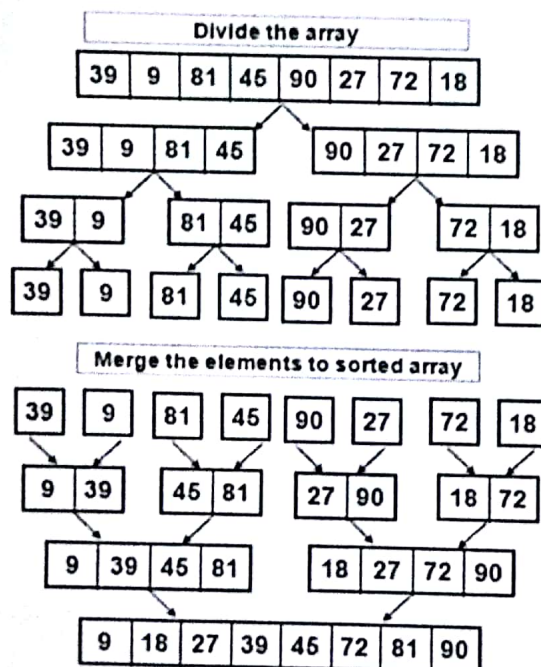
To sort the entire sequence $A[1 .. n]$, make the initial call to the procedure $\text{MERGE-SORT}(A, 1, n)$.

$\text{MERGE-SORT}(A, p, r)$

2)

1. IF $p < r$ // Check for base case
2. THEN $q = \text{FLOOR}[(p + r)/2]$ // Divide step
3. MERGE (A, p , q) // Conquer step.
4. MERGE (A, $q + 1$, r) // Conquer step.
5. MERGE (A, p , q , r) // Conquer step.

Ex:- A list of unsorted elements are: 39 9 81 45 90 27 72 18



Sorted elements are: 9 18 27 39 45 72 81 90

Time Complexity of merge sort:

Best case : $O(n \log n)$

Average case : $O(n \log n)$

Worst case : $O(n \log n)$

Program for Quick Sort in C++

```
#include <iostream>

using namespace std;

void quick_sort(int[],int,int);
int partition(int[],int,int);
int main()
{
    int a[50],n,i;
    cout<<"How many elements?";
    cin>>n;
    cout<<"\nEnter array elements:";

    for(i=0;i<n;i++)
        cin>>a[i];

    quick_sort(a,0,n-1);
    cout<<"\nArray after sorting:";

    for(i=0;i<n;i++)
        cout<<a[i]<<" ";

    return 0;
}

void quick_sort(int a[],int l,int u)
{
    int j;
    if(l<u)
    {
        j=partition(a,l,u);
        quick_sort(a,l,j-1);
        quick_sort(a,j+1,u);
    }
}

int partition(int a[],int l,int u)
{
    int v,i,j,temp;
    v=a[l];
    i=l;
    j=u+1;

    do
    {
        do
```

```
while(a[i]<v&& i<=u);

        do
            j--;
        while(v<a[j]);

        if(i<j)
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }while(i<j);

    a[l]=a[j];
    a[j]=v;

    return(j);
}
```

Output

How many elements?6

Enter array elements:9 15 6 7 10 12

Array after sorting:6 7 9 10 12 15

Program for merge Sort in C++

```
#include<iostream.h>
#include<conio.h>
```

```
void merge(int arr[], int start, int middle, int end)
```

```
{
    int i, j, k;
    int n1 = middle - start + 1;
    int n2 = end - middle;
```

```
    int L[30], R[30];
```

```
    for (i = 0; i < n1; i++)
        L[i] = arr[start + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
```

```
    i = 0;
    j = 0;
    k = start;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

```
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
```

```
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
    }
```

```
        k++;
    }
}
```

```
void mergeSort(int arr[], int start, int end)
```

```
{
    if (start < end)
    {
        int middle = start + (end - start) / 2;

        mergeSort(arr, start, middle);
        mergeSort(arr, middle + 1, end);

        merge(arr, start, middle, end);
    }
}
```

```
void main()
```

```
{ clrscr();
    int array[50], n;
    cout << "Enter the number of elements\n(MAX 50): "; cin >> n;
```

```
    for (int i = 0; i < n; i++) { cin >> array[i];
    }
```

```
    mergeSort(array, 0, n - 1);
```

```
    cout << "\n Array after sorting : ";
```

```
    for (i = 0; i < n; i++)
        cout << array[i] << " ";
```

```
    getch();
}
```


DATA STRUCTURES

Unit-4:

1. Define binary search tree. Show how to insert and delete an element from binary search tree.
2. Write algorithm to insert and delete an element from binary search tree.
3. Explain in-order traversal of threaded binary tree with an example.
4. What operations can be performed on binary trees? Discuss.
5. Define fully binary tree.
6. Define path in a tree.
7. Write in-order, pre-order and post-order traversal of a binary tree.
8. List the different tree traversals.
9. Explain binary tree ADT.
10. Discuss representation of binary tree using arrays and linked list.
11. Construct max heap for the following: 140, 80, 30, 20, 10, 40, 30, 60, 100, 70, 160, 50, 130, 110, 120.
12. What is heap. And heap as Priority Queues.
13. Explain definition of max heap. Explain Insertion into a Max Heap, Deletion from a Max Heap.

Unit-5:

1. What is a graph. Explain the properties of graphs.
2. What are connected components of graph. Is there a method to find out all the connected components of graph. Explain.
3. Write breadth first traversal algorithm. Explain with an example.
4. Write Depth First Search algorithm. With an example.
5. Explain Prim's algorithm with an example.
6. Discuss Kruskal's algorithm with an example.
7. Explain how to represent a graphs.
8. Explain All-Pairs Shortest Path.
9. What are Spanning Trees.
10. Explain Minimum Cost Spanning Trees.

Unit-6:

1. Rearrange following numbers using quick sort: 10, 6, 3, 7, 17, 26, 56, 32, 72.
2. Write a program to sort the elements using radix sort.
3. Write algorithm for merge sort.
4. Differentiate between iterative merge sort and recursive merge sort
5. Discuss how to sort elements using merge sort with suitable example.
6. State and explain heap sort with example.
7. Evaluate time complexity and space complexity of an algorithm.
8. State and explain insertion sort with example.
9. What is the best sorting technique.
10. Evaluate time complexity of insertion sort, quick sort, merge sort.