Objective:

To understand the concepts of Data Structures and Algorithms.

Unit I

Arrays and sequential representations – ordered lists – Stacks and Queues – Evaluation of Expressions – Multiple Stacks and Queues – Singly Linked List – Linked Stacks and queues – Polynomial addition.

Unit II

Trees – Binary tree representations – Tree Traversal – Threaded Binary Trees – Binary Tree Representation of Trees – Graphs and Representations – Traversals, Connected Components and Spanning Trees – Shortest Paths and Transitive closure – Activity Networks – Topological Sort and Critical Paths.

Unit III

Algorithms – Priority Queues - Heaps – Heap Sort – Merge Sort – Quick Sort – Binary Search – Finding the Maximum and Minimum.

Unit IV

Greedy Method : The General Method – Optimal Storage on Tapes – Knapsack Problem – Job Sequencing with Deadlines – Optimal Merge Patterns.

Unit V

Back tracking: The General Method – The 8-Queens Problem – Sum of Subsets – Graph Coloring.

Text Books:

1. Fundamentals of Data Structure – Ellis Horowiz, SartajSahni, Galgotia Publications, 2008.
2. Computer Algorithms – Ellis Horowiz, SartajSahni and Sanguthevar Rajasekaran, University Press, 2008.

1. What is a Data Structure?

A data structure is a way of organizing the data so that the data can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

2. What are linear and non linear data Structures?

- **Linear**: A data structure is said to be linear if its elements form a sequence or a linear list. Examples: Array. Linked List, Stacks and Queues
- **Non-Linear**: A data structure is said to be non-linear if traversal of nodes is nonlinear in nature. Example: Graph and Trees.

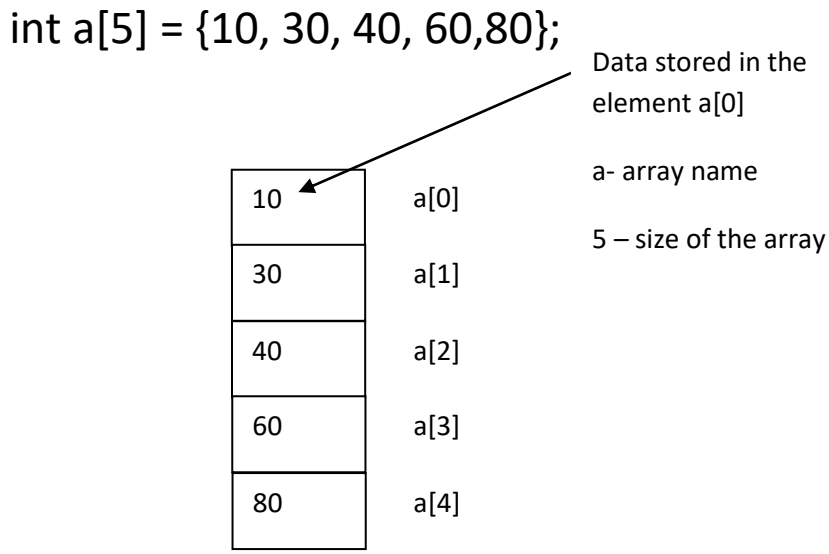
3. What are the various operations that can be performed on different Data Structures?

- **Insertion**: Add a new data item in the given collection of data items.
- **Deletion**: Delete an existing data item from the given collection of data items.
- **Traversal**: Access each data item exactly once so that it can be processed.
- **Searching**: Find out the location of the data item if it exists in the given collection of data items.
- **Sorting**: Arranging the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

4. Explain array with example (5 marks)

An array is a linear data structure that contains a collection of data items of the same type. Each value is called an element of the array. The elements of the array share the same variable name but each element has its own unique index number (also known as a subscript).

Example



Features/characteristics of array

- Array is a linear data structure
- It contains fixed number elements of same data type
- The elements of array are stored in consecutive memory locations
- Array provides random access.
- It provides constant time access
- Any element of an array can be accessed using its index.
- Array provides fast performance

5. Describe memory representation of array/describe memory representation of one dimensional array

An array is a linear data structure that contains a collection of data items of the same type. Each value is called an element of the array. The elements of the array share the same variable name but each element has its own unique index number (also known as a subscript).

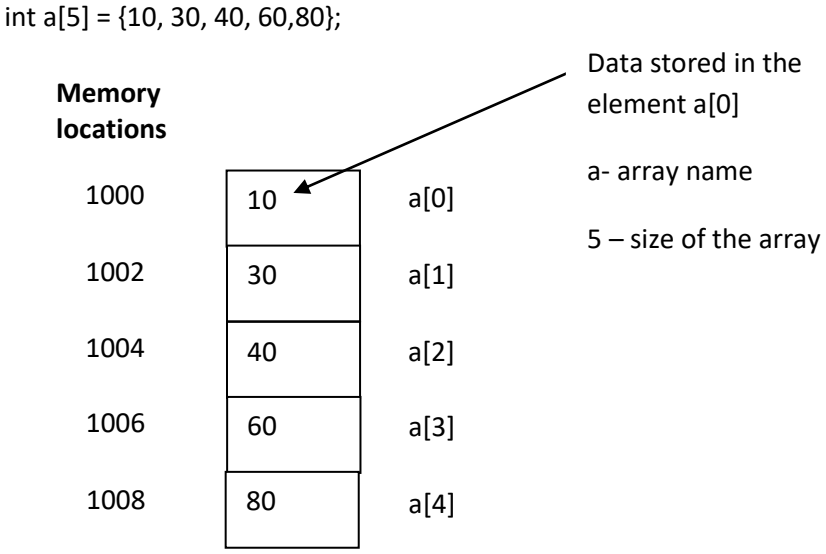
One dimensional array is an array in which each element is represented using one index

MEMORY REPRSENTATION

In memory, the elements are array are represented or stored in consecutive memory locations.

Consider an integer array, say 'a'. Consider that each integer occupies two bytes. The size of array is say 5.

Now this array is represented in memory as shown in the diagram.



The starting address of the array is called Base address. In this example, Base address=1000

The memory location k^{th} element is calculated using the formula

LOC(a[k])=Base address + (k ×width of each element)

Example:

K=3; base address = 1000 and width of each element = 2 bytes

Now, LOC(a[3])=1000+(3×2)=1006

6. Explain Two dimensional array and its representation (10 marks)

Two dimensional array is an array in which each element is represented using two indices. Example. Consider a two dimensional array, A

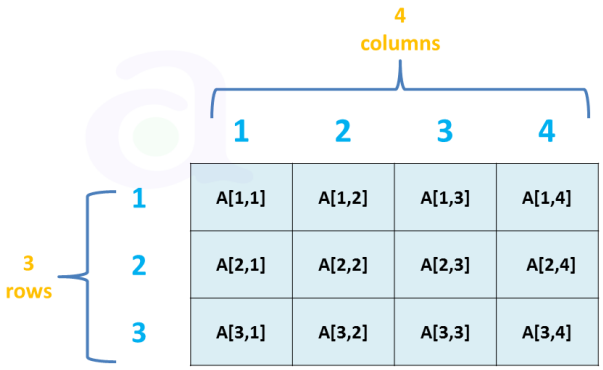


Fig: Two-dimension 3x4 array

The two dimensional array is represented with two indices or dimensions. Here A[3,2]. Size=6
Memory representation of two dimensional array.
In memory, a two dimensional array is represented in two ways
(i) column major order – here elements are stored column by column
(ii) row major order- here elements are stored row by row

Representation in column major order

In column major order, the elements are represented column by column in sequential memory locations.

Two dimensional array

A[1,1]	A[1,2]
A[2,1]	A[2,2]
A[3,1]	A[3,2]

Memory representation - Column major order

A[1,1]	1000
A[2,1]	1002
A[3,1]	1004
A[1,2]	1006
A[2,2]	1008
A[3,2]	1010

The memory location of (i,j)th element is:

LOC(A[i,j]) = BASE ADDRESS(A) + W[M(j-1) + (i-1)]

Where w=width of an element. M- Total number of rows
find the address of A[3,2]. Say for example, the Base address=1000; w=2 bytes.
In the example, M=3; w=2 bytes. Base address=1000

LOC(A[3,2])=1000+2[3(2-1)+(3-1)]
LOC(A[3,2])=1000+2[3(1)+(2)]
LOC(A[3,2])=1000+2[3+2]=1000+10=1010

Row major order- in row major order representation, the elements of the array represented row by row in sequential memory locations.

Two dimensional array

A[1,1]	A[1,2]
A[2,1]	A[2,2]
A[3,1]	A[3,2]

Memory representation - Row major order

A[1,1]	1000
A[1,2]	1002
A[2,1]	1004
A[2,2]	1006
A[3,1]	1008
A[3,2]	1010

The memory location of (i,j)th element is:

LOC(A[i,j]) = BASE ADDRESS(A) + W[N(i-1) + (j-1)]

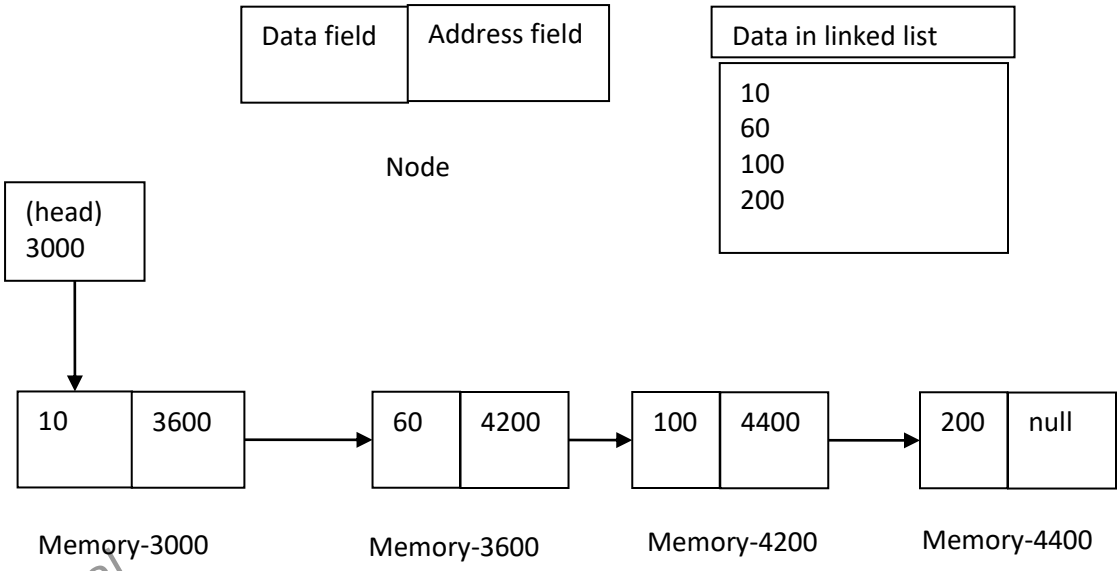
Here, w- width of each element, N-number of columns
Here, base address=1000, w=2, N=2
Example- LOC(A[3,1])=1000+2[2(3-1)+(1-1)]
=1000+2[2(2)+0]=1000+2[4+0]=1000+8=1008

7. Explain linked list with example

- A linked list is a linear data structure where elements are not stored at contiguous or consecutive memory locations. The elements are linked using pointers.
- Each node of a list is made up of two items - the data and a reference to the next node.
- The last node has a reference to null.

As in the diagram, each node contains, data field and address of the next node (pointer). The first node of the list is called head which contains the address of the first element of the list. The last node of the list is called tail which contains null if there no further element. In the example, there are 4 elements. The elements are not stored in consecutive memory locations. They are linked using pointers. The first element contains the address of 2nd element. The 2nd element contains

the address of 3rd element and so on. Here, the first element is at memory 3000. 2nd element is at memory location 3600. 3rd element is stored at memory 4000 and 4th element is at 4400.



Linked list data structure

8. Explain types of linked lists (10 marks)

There are three types of Linked List.

- Singly Linked List
- Doubly Linked List
- Circular Linked List

1. Singly Linked List

- Each node has a single link to another node is called Singly Linked List.
- Singly Linked List does not store any pointer any reference to the previous node.
- Each node stores the contents of the node and a reference to the next node in the list.
- In a singly linked list, last node has a pointer which indicates that it is the last node. It requires a reference to the first node to store a single linked list.
- It has two successive nodes linked together in linear way and contains address of the next node to be followed.

- It has successor and predecessor. First node does not have predecessor while last node does not have successor. Last node have successor reference as NULL.
- It has only single link for the next node.
- ***In this type of linked list, only forward sequential movement is possible, no direct access is allowed***.

2. Doubly Linked List

- Doubly linked list is a sequence of elements in which **every node has link to its previous node and next node.**
- **Traversing can be done in both directions** and displays the contents in the whole list.

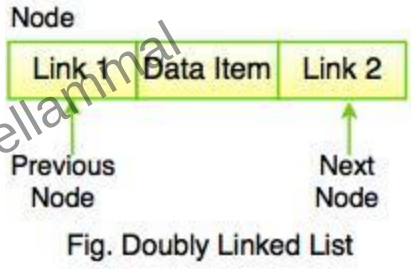


Fig. Doubly Linked List

Advantages of Doubly Linked List

- Doubly linked list can be traversed in both forward and backward directions.
- To delete a node in singly linked list, the previous node is required, while in doubly linked list, we can get the previous node using previous pointer.
- It is very convenient than singly linked list. Doubly linked list maintains the links for bidirectional traversing.

Disadvantages of Doubly Linked List

- In doubly linked list, each node requires extra space for previous pointer.

- All operations such as Insert, Delete, Traverse etc. require extra previous pointer to be maintained.

Circular Linked List

A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.

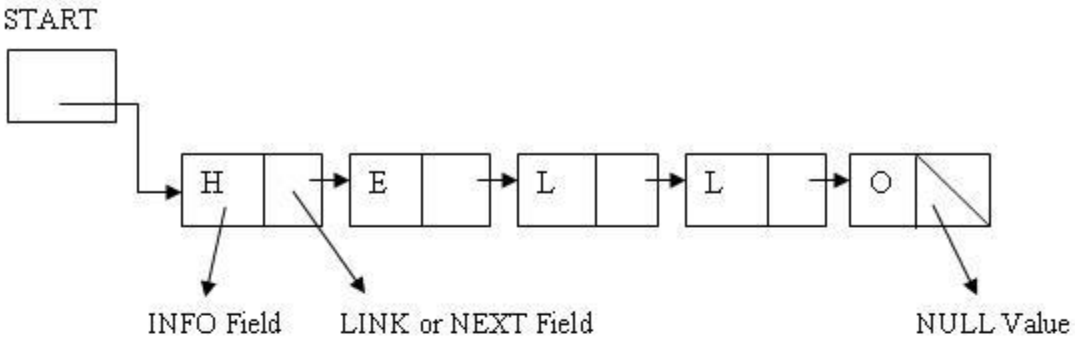
9. Give differences between array and linked list

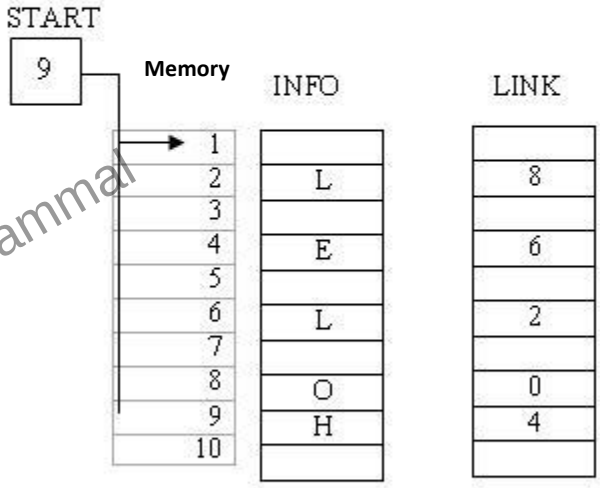
Array	Linked List
Array is collection of elements with same data type	Linked list is linear data structure in which nodes are connected using pointers
They can be accessed randomly	They need to traversed linearly
They are stored continuous	They are stored as per the memory location available.
Memory is allocated statically ie at compile time	Memory is allocated dynamically ie. At run time
Insertion and deletion operation is long process as rest of the elements need to be shifted to make space for new element	In this for insertion only pointers needed to be changed.
Elements are independent of each other	In this element are connected with each other using pointers as they are linked using addresses.
No need of pointers so no extra memory required	Extra memory overhead for storing pointers
Types of array are single and multi dimensional	Linked list are singly, doubly and circular linked list

10. Explain memory representation of linked list (10 marks)

Let LIST is linear linked list. It needs two linear arrays for memory representation. Let these linear arrays are INFO and LINK. INFO[K] contains the data or information part and LINK[K] contains the next pointer field of node K.

A variable START is used to store the location of the beginning of the LIST and NULL is used as next pointer sentinel which indicates the end of LIST. It is shown below:



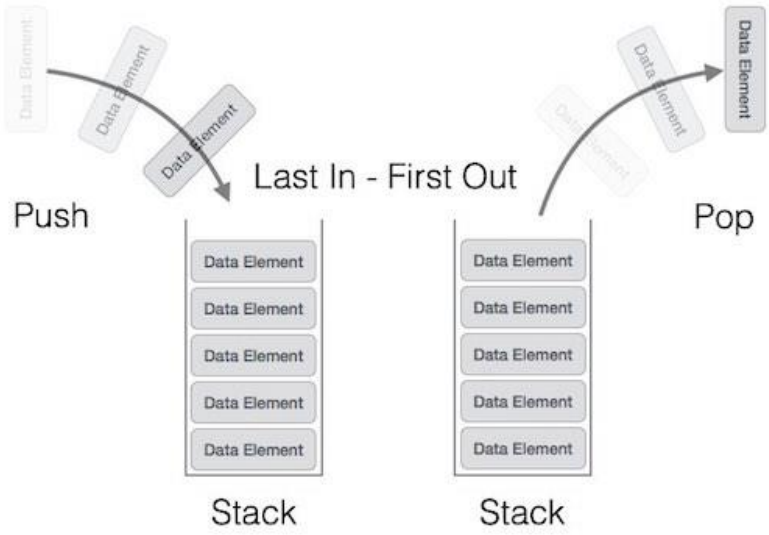


Here

START = 9	=>	INFO[9] = H is the first character.
LINK[9] = 4	=>	INFO[4] = E is the second character.
LINK[4] = 6	=>	INFO[6] = L is the third character.
LINK[6] = 2	=>	INFO[2] = L is the fourth character.
LINK[2] = 8	=>	INFO[8] = O is the fifth character.
LINK[8] = 0	=>	The NULL value, so the LIST ends here.

11. Describe in detail stack data structure with its operations (10 marks)

Stack – Stack is a linear data structure in which elements can be inserted or removed only at the top of the stack according to Last In First Out(LIFO) principle.



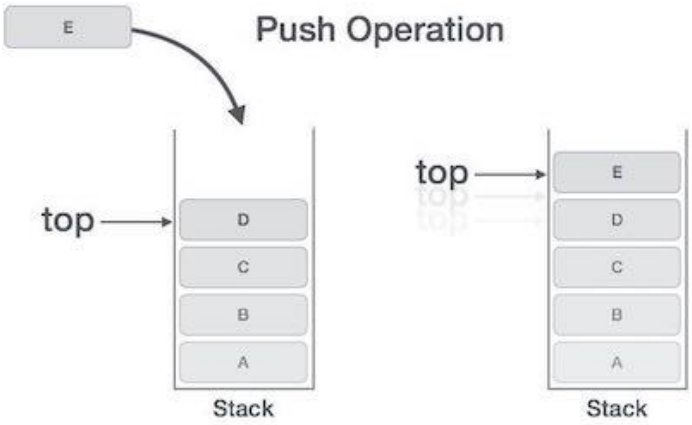
Push Operation

The process of inserting a new data element onto stack is known as a Push Operation.

Push operation involves a series of steps −

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

```
begin procedure push: stack, data
   if stack is full
      return “stack is full”
   endif

   top=top+1
   stack[top]=data

end procedure
```

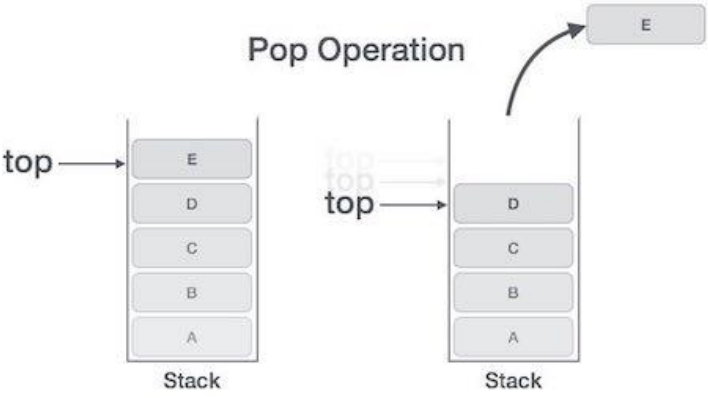


Pop Operation

The process of removing an element from the stack is called PoP Operation.

A Pop operation may involve the following steps −

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




```
begin procedure pop: stack
   if stack is empty
      return “Stack is empty”
   endif
    data = stack[top]
   top = top - 1
   return data
end procedure
```

12. List out the basic operations of stack

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

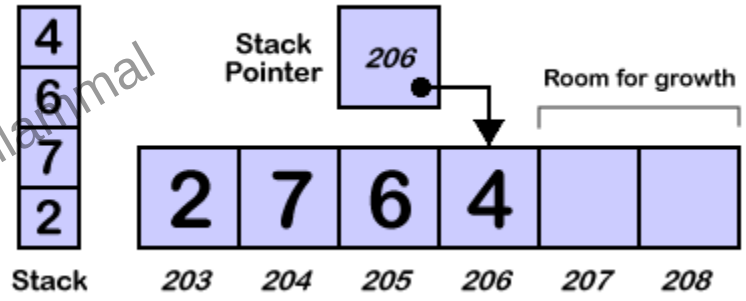
13. Explain the memory representation of stack (10 marks)

Stack can be represented in memory in two ways (**i) using array (ii) using linked list**

Array representation of stack (5 marks)

A stack is a data structure that can be represented as an array. An array is used to store an ordered list of elements. an array can be used to represent a stack by taking an array of maximum size; big enough to manage a stack.

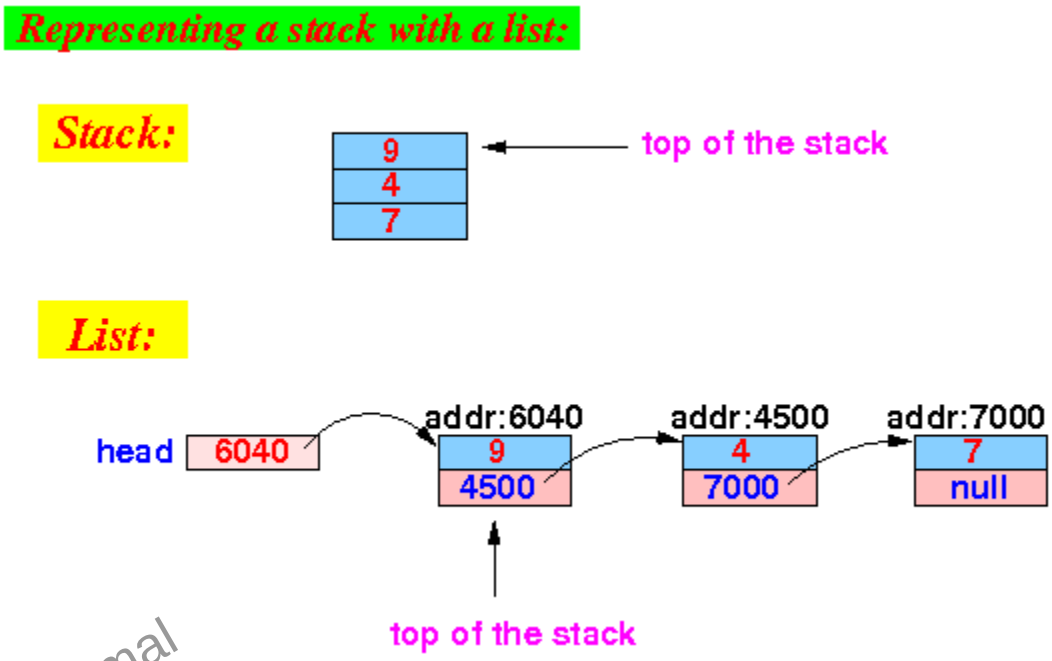
Example



- A stack can be maintained by a linear array
- Stack pointer holds the address of the TOP of the stack
- The condition TOP = 0 or TOP = NULL will indicate that the stack is empty.

Linked list representation of stack (5 marks)

A single linked list data structure is sufficient to represent any stack. It is shown.



- In linked list represent, the first node on the list is the current item at the top of the stack. In the above example, node 9 is at the top of the stack and head of linked list is pointing the top element. The last node contains the element 7.
- The push operation adds the element in the front of list.
- Similarly, pop operation removes the element from the front of the list.

14. What is Stack and where it can be used?

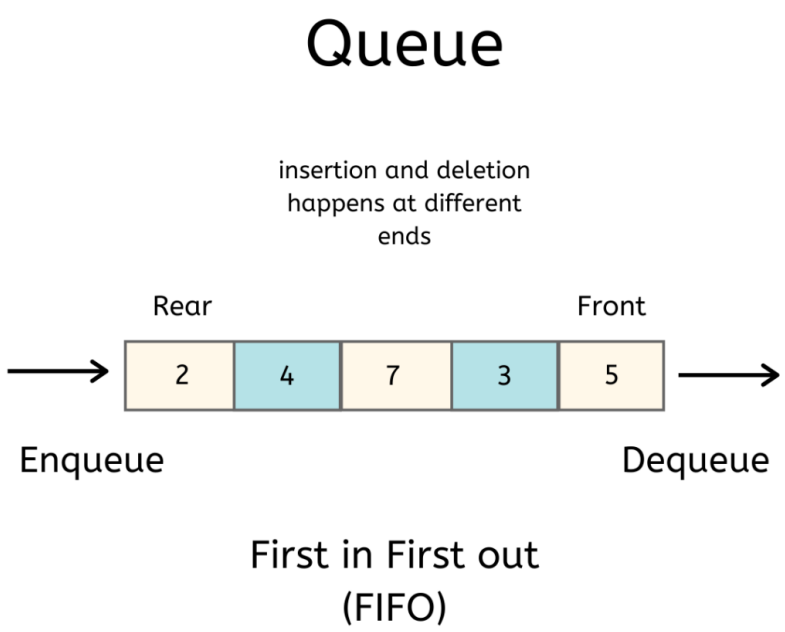
Stack is a linear data structure which the order LIFO(Last In First Out) or FILO(First In Last Out) for accessing elements. Basic operations of stack are : Push, Pop , Peek.

Applications of Stack:

1. Infix to Postfix Conversion using Stack
2. Evaluation of Postfix Expression
3. Reverse a String using Stack
4. Implement two stacks in an array
5. Check for balanced parentheses in an expression

15. Explain Queue in detail. (10 marks)

Queue is a linear list of elements in which an element can be inserted at 'rear' end and an element can be removed from the other end, called 'front' end. Queue follows the order is First In First Out (FIFO) to access elements.

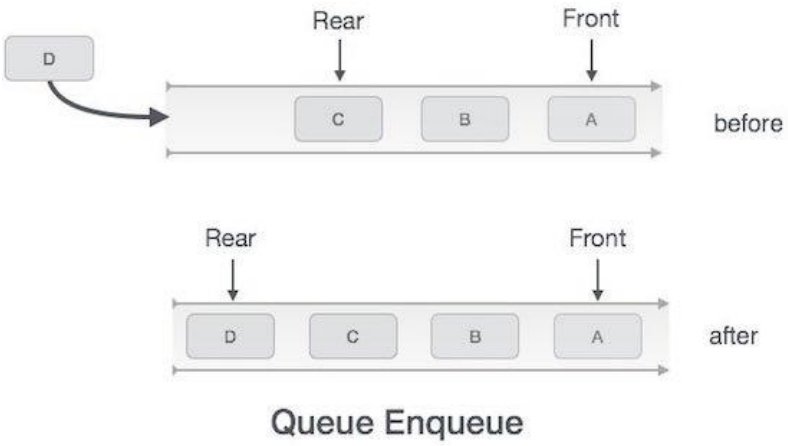


Enqueue Operation

The process of inserting an element in the rear end of the queue is called enqueue

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.
The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Algorithm for enqueue operation

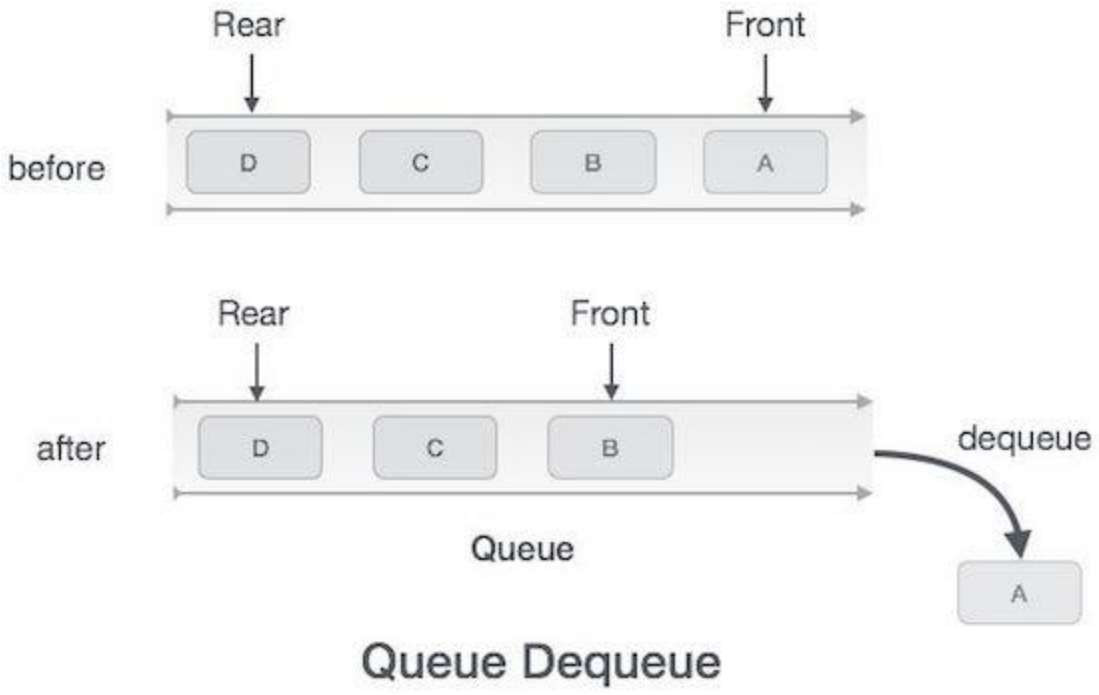
```
procedure enqueue(data)

   if queue is full
      return overflow
   endif

   rear = rear + 1
   data = queue[rear]
   return true

end procedure
```


Dequeue Operation

The process of removing an element from the front end of queue is called dequeue

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation


```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front = front + 1
   return true

end procedure
```

16.What are Infix, prefix, Postfix notations? Describe in detail how to evaluate expression using stack.

Infix notation

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as

A * (B + C) / D

Postfix notation (also known as "Reverse Polish notation"): Operators are written after their operands. The infix expression given above is equivalent to

A B C + * D/

(A(BC+)*)D/ = ABC+*D/

Prefix notation (also known as “Polish notation”): Operators are written before their operands. The expressions given above are equivalent to

/(*A(+BC))D

/ * A + B C D

Consider an infix expression

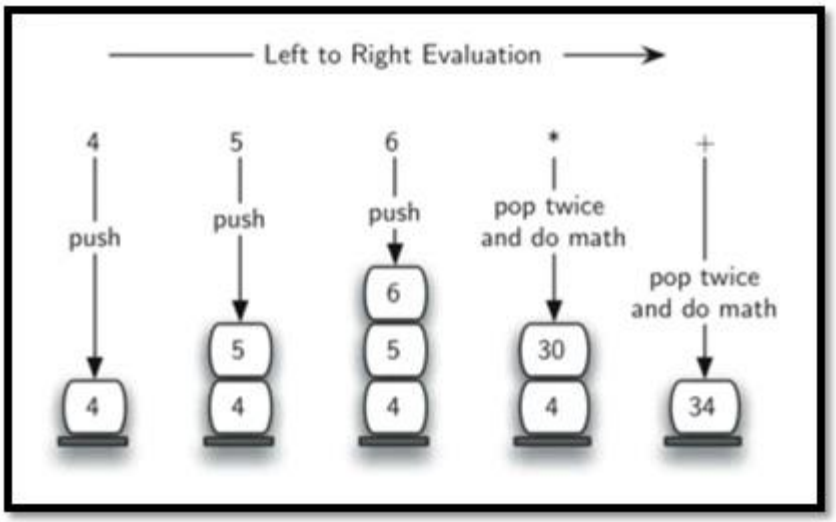
4+5*6

Postfix 4+(56*) = 4(56*)+=456*+

As **Postfix expression** is without parenthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle.

Evaluation rule of a Postfix Expression states:

1. While reading the expression from left to right, push the element in the stack if it is an operand.
2. Pop the two operands from the stack, if the element is an operator and then evaluate it.
3. Push back the result of the evaluation. Repeat it till the end of the expression.



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Evaluation of prefix expressions using stack

In prefix expression evaluation we have to scan the string from right to left. If we encounter any operand then we will push it into the stack and if we encounter any operator then we will pop two elements from the stack and perform the operation using current operator and the result will get pushed in stack.

Evaluate below prefix expression +-927

Accept above string in variable exp.

Read string from right to left.

Step No.	Value of i	Operation	Stack
1	7	Push 7 in stack	7
2	2	Push 2 in stack	**2** 7
3	9	Push 9 in stack	**9** **2** 7
4	-	Pop 2 elements form stack, and perform subtraction operation, and push result back in stack. i.e. 9 - 2 = 7	 7 7
5	+	Pop 2 elements form stack, and perform addition operation, and push result back in stack. i.e. 7 + 7 = 14	14
6		Pop result and display it	

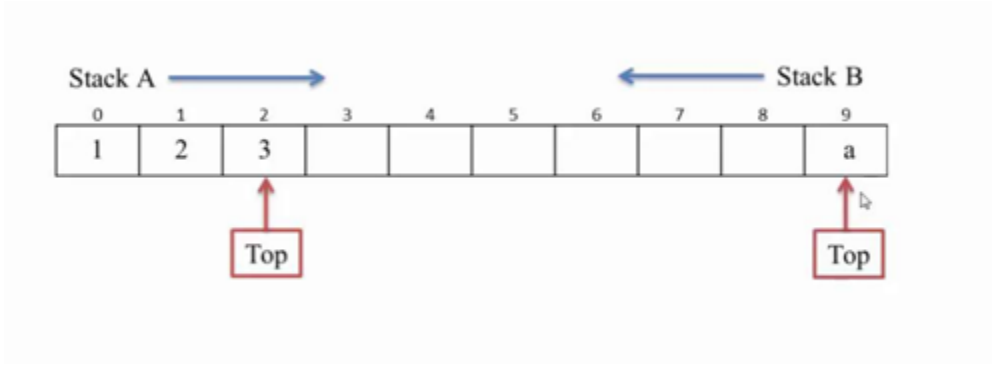
17. Explain multiple stacks and multiple queues. (5 marks)

Multiple stacks

When a stack is created using single array, we can not able to store large amount of data, thus this problem is rectified using more than one stack in the same array of sufficient array. This technique is called as Multiple Stack.

Example

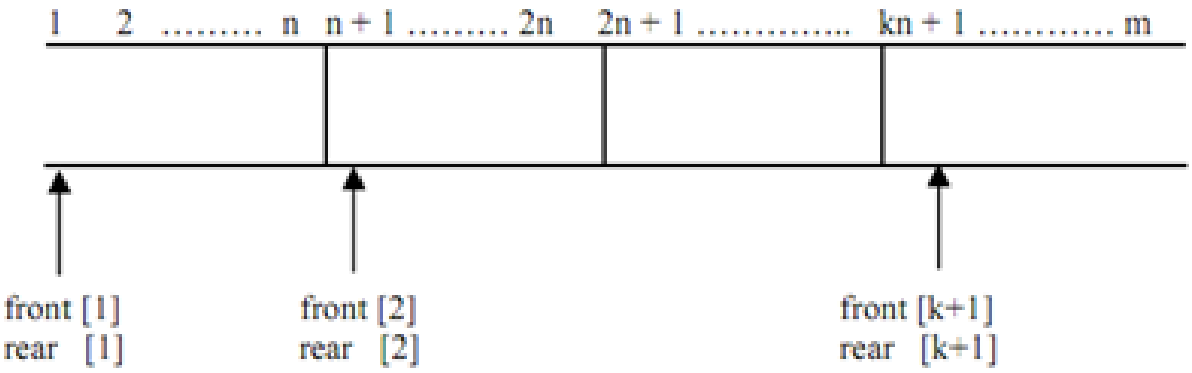
In the following diagram, two stacks are implemented in single array



Multi Queue

Multi queue is data structure in which multiple queues are maintained. This type of data structures are utilized for process scheduling. We might use one dimensional array or multidimensional array to illustrate a multiple queue. A multi queue implementation by using a single dimensional array

along m elements is illustrated in following figure. Each of queues contains n elements that are mapped to a liner array of m elements



18. List out the basic operations of queue

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

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

19. Give the applications of Queue

1.When data is transferred asynchronously between two processes. eg. IO Buffers.
2.When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
3.In recognizing palindrome.
4.In shared resources management.
5.Keyboard buffer.
6.Round robin scheduling.
7.Job scheduling.
8.Simulation

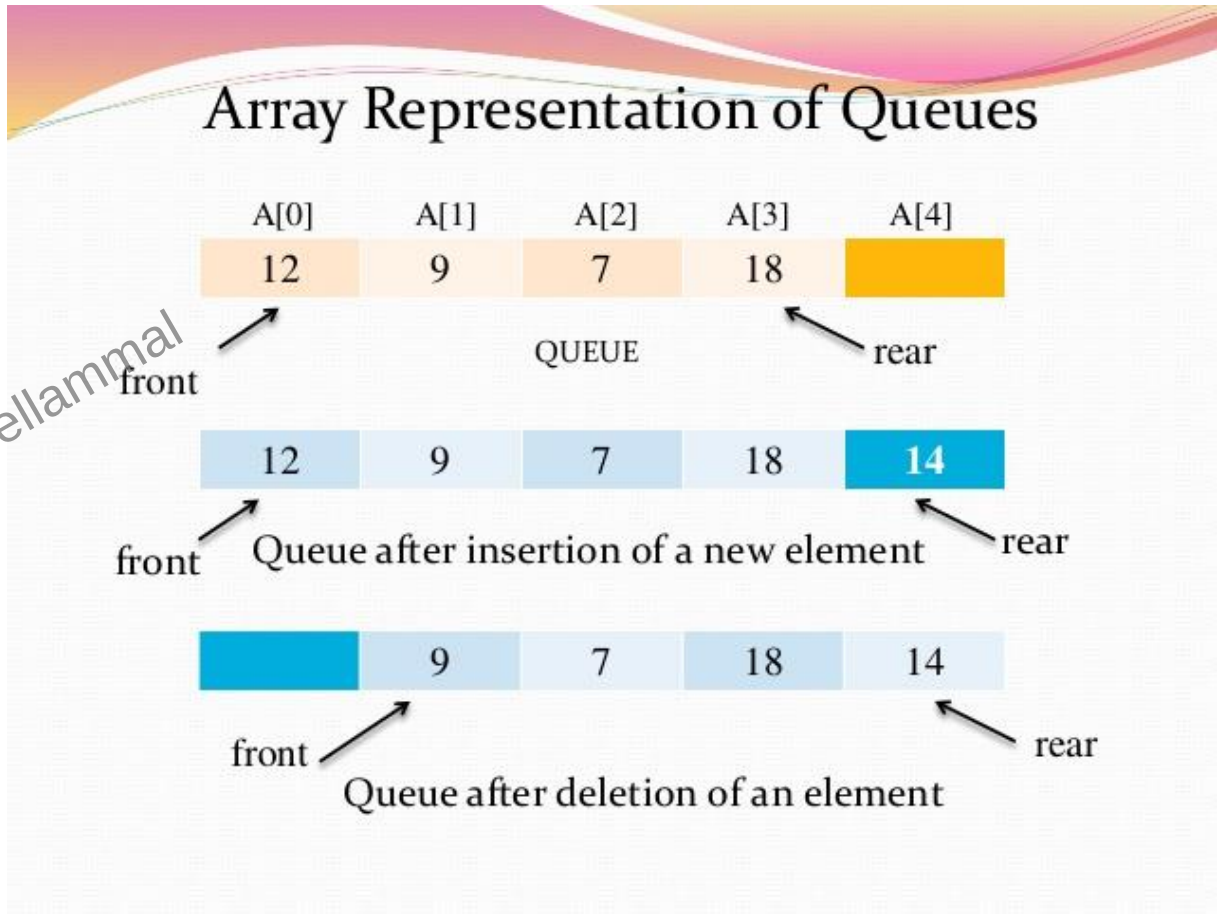
20. Explain in detail the memory representation of queue

A queue can be represented in memory using (i) array (ii) linked list

Array representation of queue in memory

To implement a queue using array, create an array *arr* of size *n* and take two variables *front* and *rear* both of which will be initialized to *0* which means the queue is currently empty. Element *rear* is the index upto which the elements are stored in the array and *front* is the index of the first element of the array

The array representation of a queue is shown in the following diagram



1. **Enqueue:** Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If *rear* < *n* which indicates that the array is not full then store the element at *arr[rear]* =item; rear=rear+1

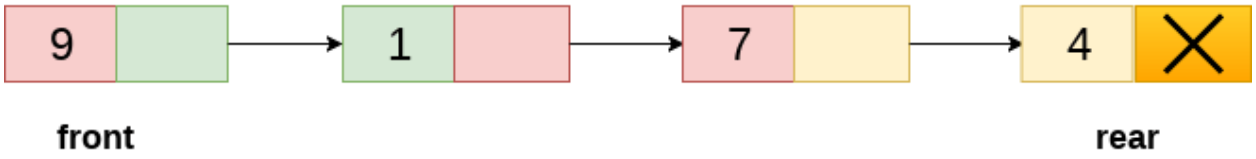
2. **Dequeue:** Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. *rear > 0. Item=arr[front]; front=front+1*

A queue can be represented using linked list as shown . In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

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

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

The linked representation of queue is shown in the following figure.



Linked Queue

Enqueue operation

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

Step – 1 Allocate memory for new node and name it as **ptr**

Step – 2 **ptr[data]=data**

ptr[link]=null

Step – 3 rear[link]= ptr

Dequeue operation

ptr=front
item=front[data]
front=front[link]

21. List out various applications of stack

- Expression Evaluation and Conversion
- Backtracking
- Function Call
- Parenthesis Checking
- String Reversal
- Syntax Parsing
- Memory Management

22. Explain in detail evaluation of polynomial addition using linked list

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

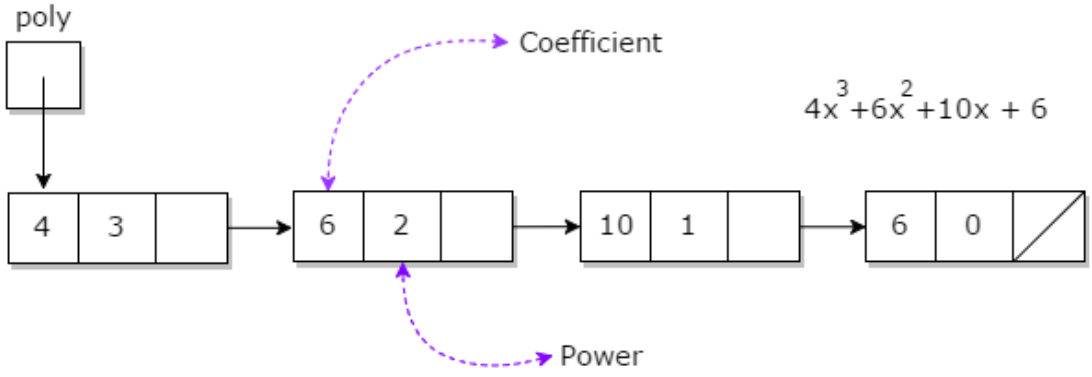
- one is the coefficient
- other is the exponent

Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

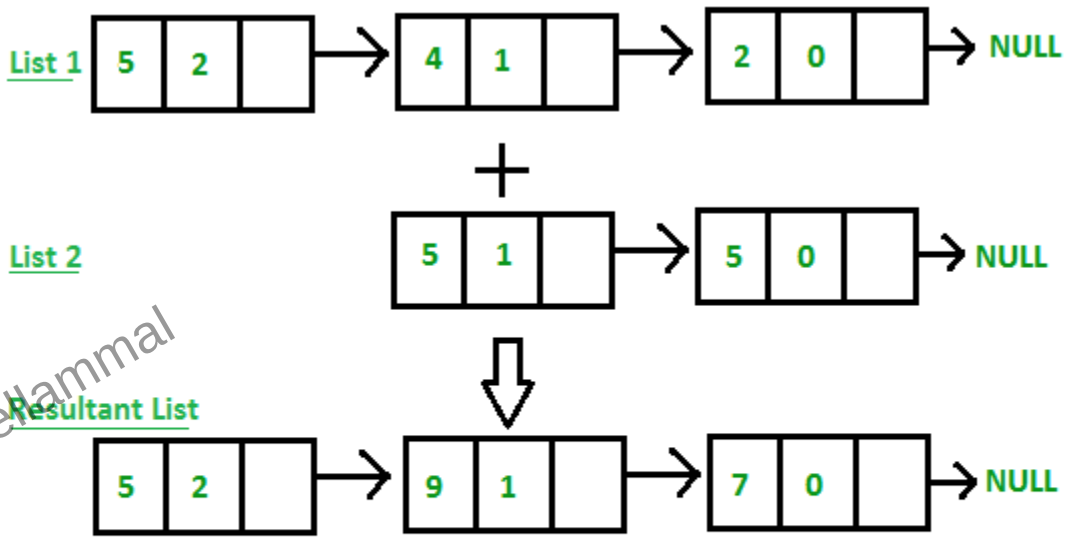
Input:

1st number = 5x^2 + 4x^1 + 2x^0

2nd number = 5x^1 + 5x^0

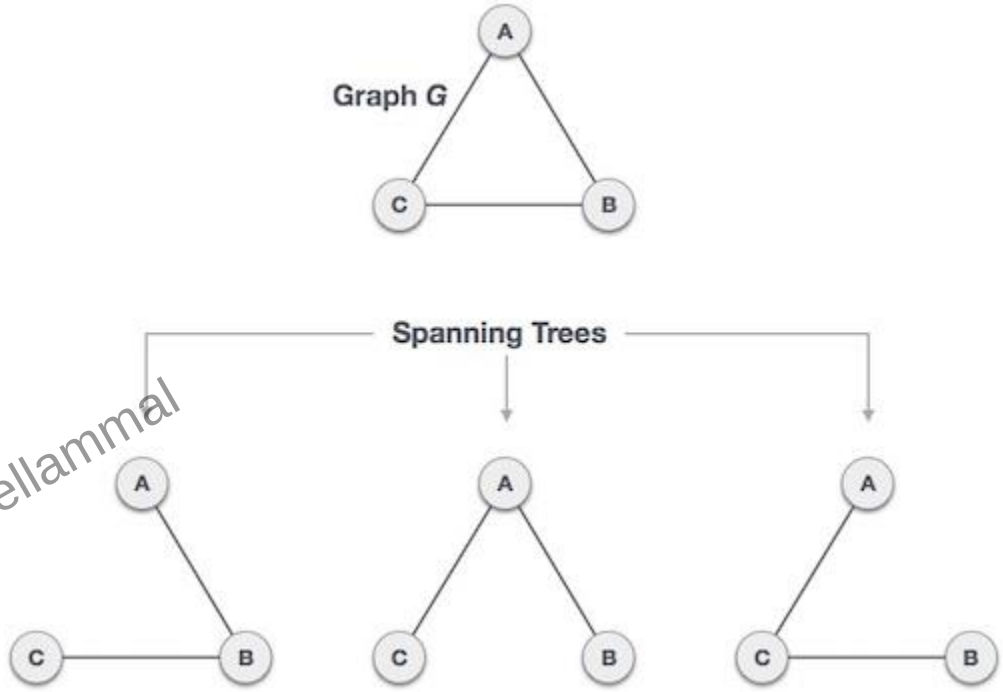
Output:

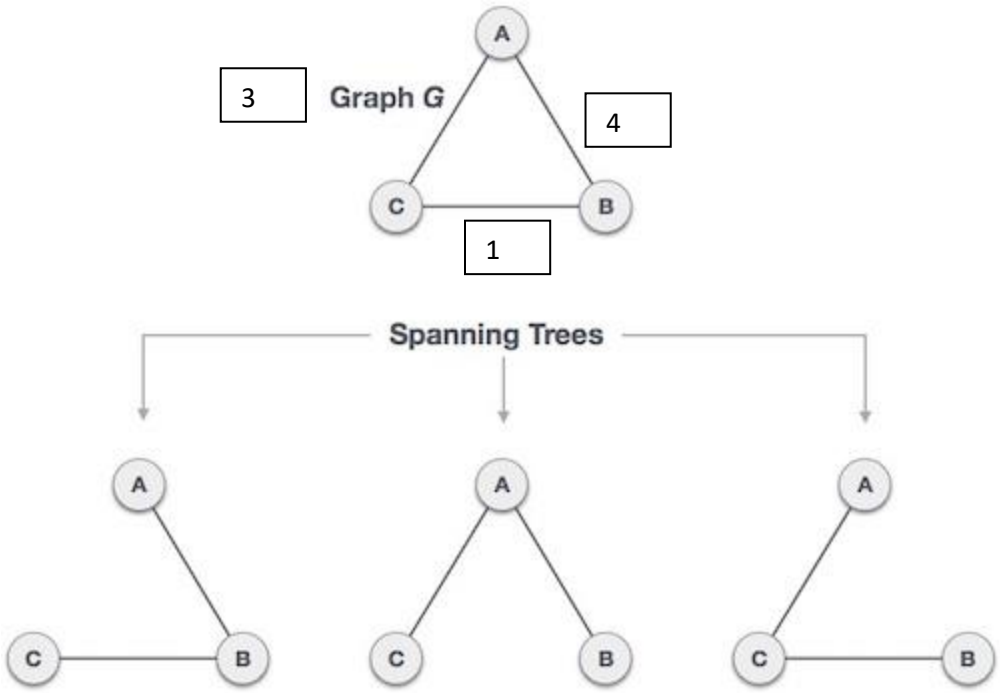
5x^2 + 9x^1 + 7x^0



Spanning tree

A spanning tree is a subset of Graph G, which has **all the vertices covered with minimum possible number of edges**. Hence, a spanning tree does not have cycles and it cannot be disconnected..





We found three spanning trees off one complete graph. A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the **same number of edges and vertices.**
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- **Civil Network Planning**

- **Computer Network Routing Protocol**
- **Cluster Analysis**

Minimum Spanning Tree (MST)

In a **weighted graph**, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

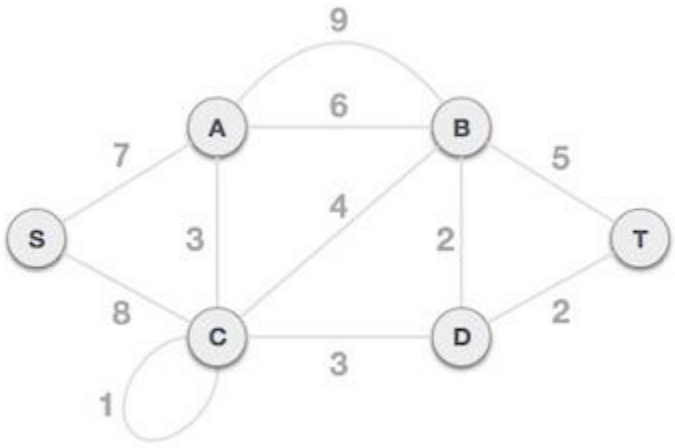
Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here −

- Kruskal's Algorithm
- Prim's Algorithm

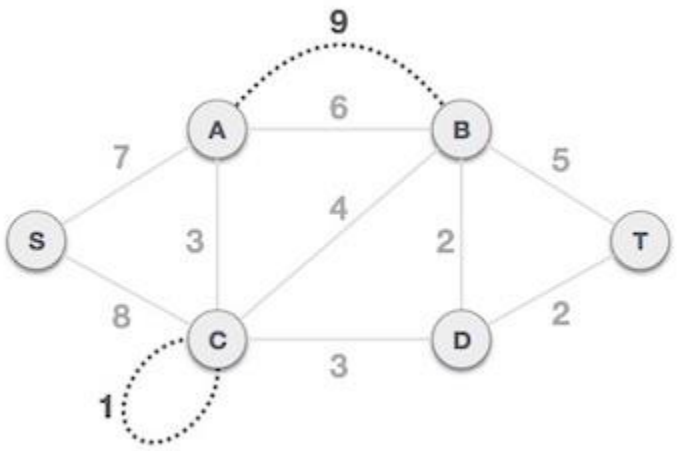
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

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

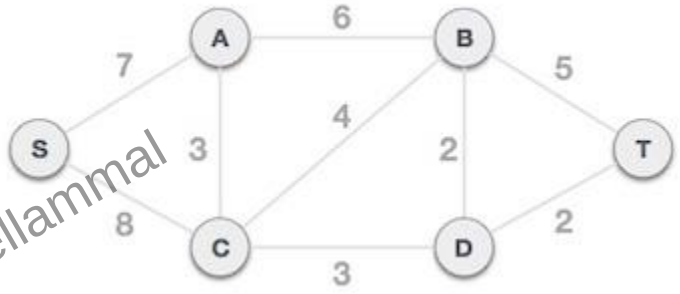


Step 1 - **Remove all loops** and **Parallel Edges**

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.



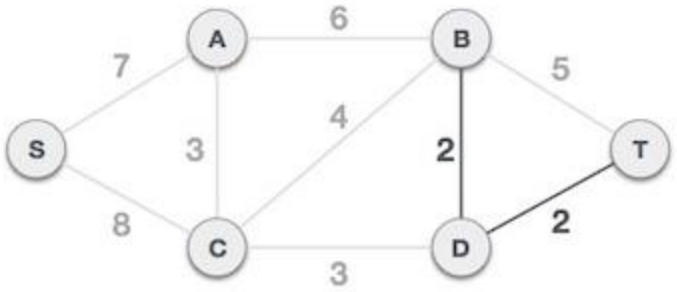
Step 2 - Arrange all edges in their increasing order of weight

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

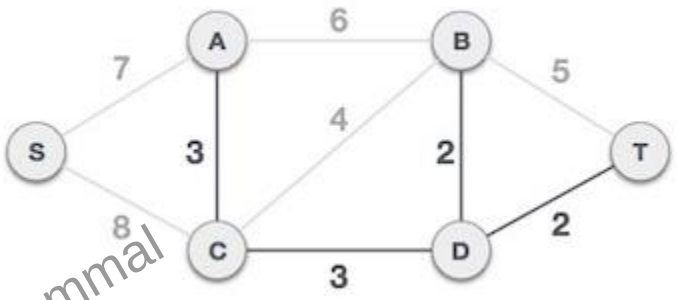
Step 3 - Add the edge which has the least weightage

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.

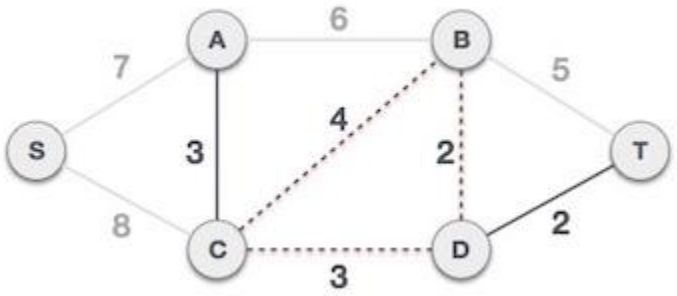


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

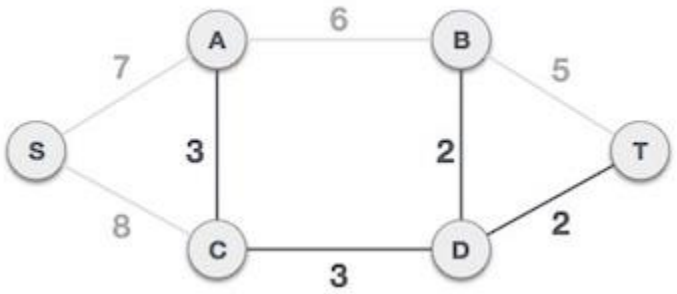
Next cost is 3, and associated edges are A,C and C,D. We add them again −



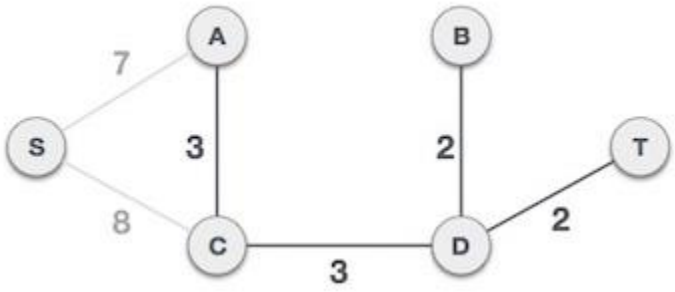
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



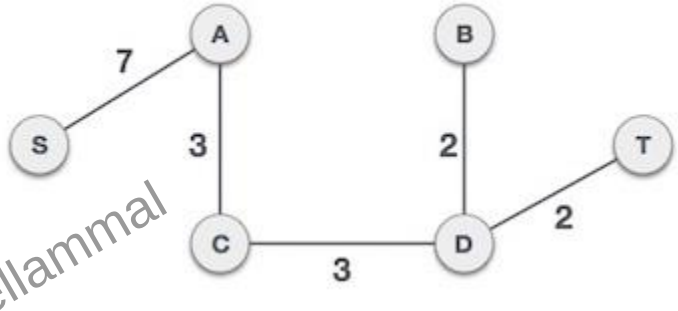
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



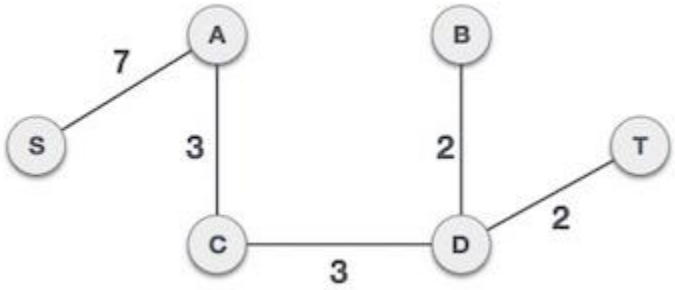
We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



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.



Prim’s algorithm

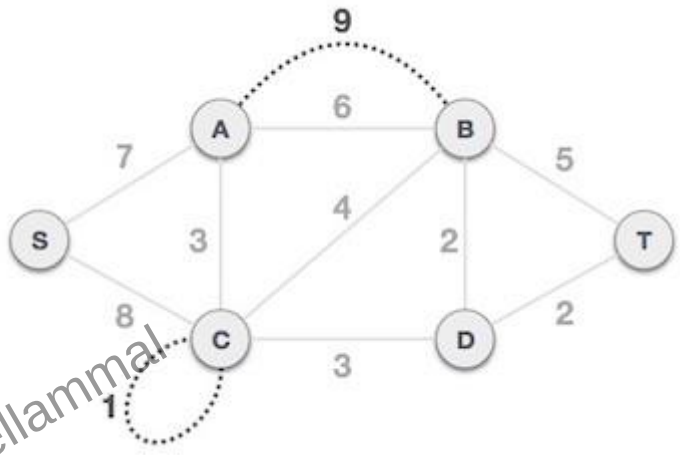
Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

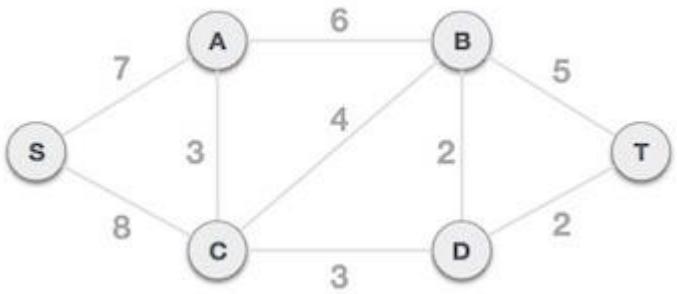
Algorithm

- **Step 1:** Select a starting vertex
- **Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
- **Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
- **Step 4:** Add the selected edge and the vertex to the minimum spanning tree T
  [END OF LOOP]
- **Step 5:** EXIT

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.

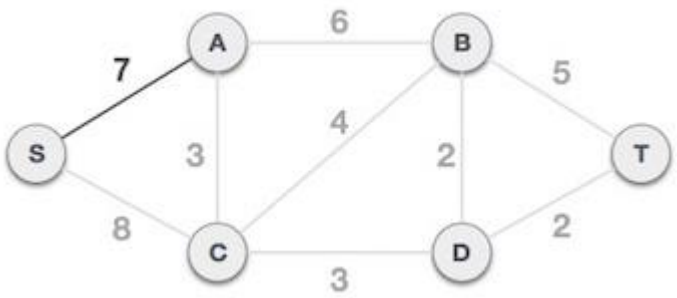


Step 2 - Choose any arbitrary node as root node

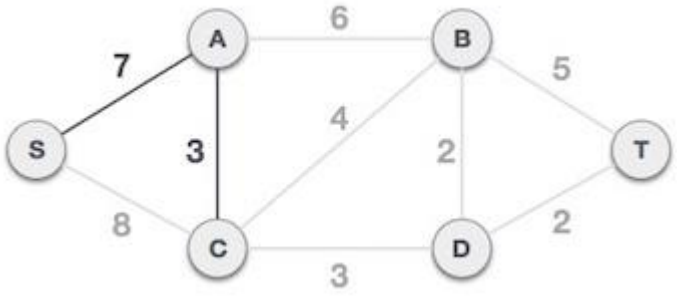
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then 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(or weight)

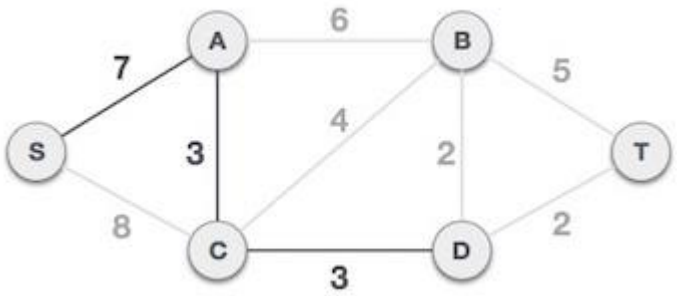
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



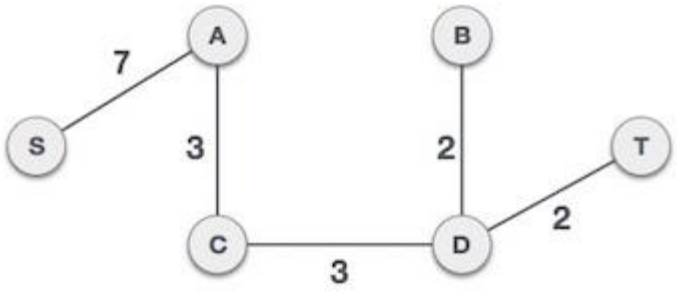
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



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.

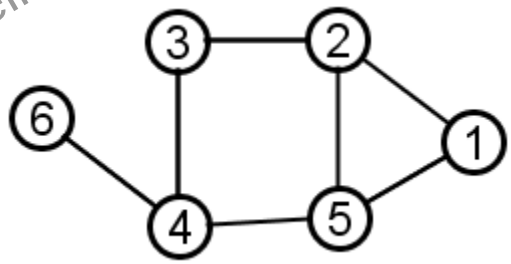
Connected graph

Connected has several distinct formal definitions, each of which is important. We introduce three of them here and elaborate the details later.

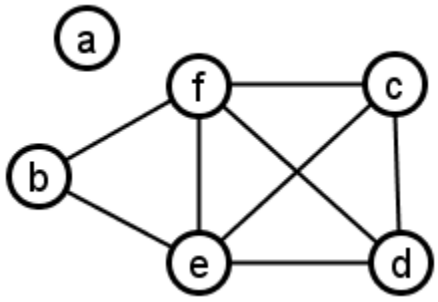
- A path connects its origin and destination nodes
- Two nodes are connected if there is at least one path that connects them
- A graph is connected if each of its nodes is connected to all the others

Examples:

- Nodes 6 and 2 are connected in graph G1 below
- Nodes a and b are not connected in graph G2 below
- Graph G1 is connected; graph G2 is not connected



Graph G1 = (V1,E1)



Graph G2 = (V2,E2)

Shortest Path algorithm

- The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the **edges weights** is minimum.
- **Dijkstra's algorithm** has many variants but the most common one is to find the **shortest paths from the source vertex to all other vertices in the graph.**
- Works on both **directed and undirected** graphs. However, all edges must have positive weights.

- **Input**: Weighted graph G={E,V} and source vertex v∈V, such that all edge weights are positive

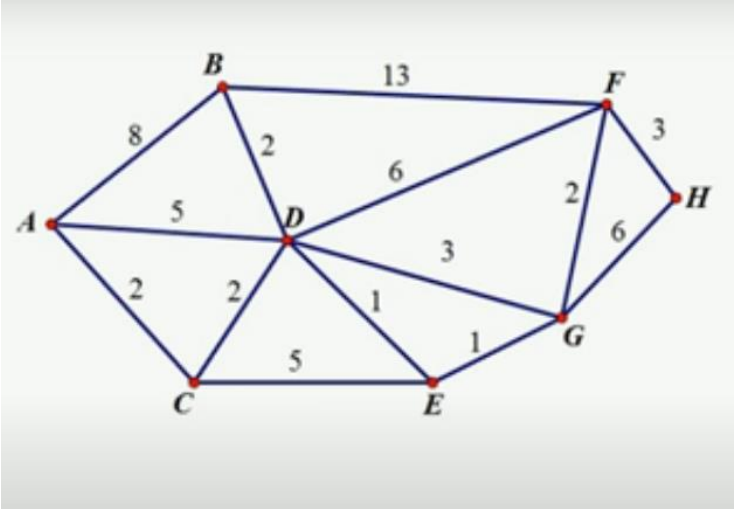
- **Output**: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex v∈V to all other vertices

- Dijkstra's Algorithm as a greedy algorithm

  ALGORITHM- STEPS

1. Initialize distances according to the **algorithm**.
2. Pick first node and calculate distances to adjacent nodes.
3. Pick next node with minimal **distance**; repeat adjacent node **distance** calculations.

4. Final result of **shortest**-**path** tree.



From the above table, the shortest path between A and E is calculated as

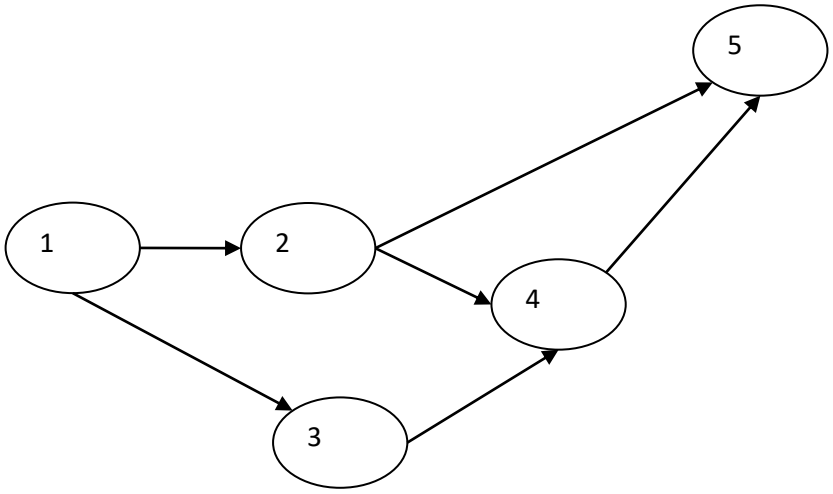
A->C, C->D, D->E=2+2+1=5

Topological sorting

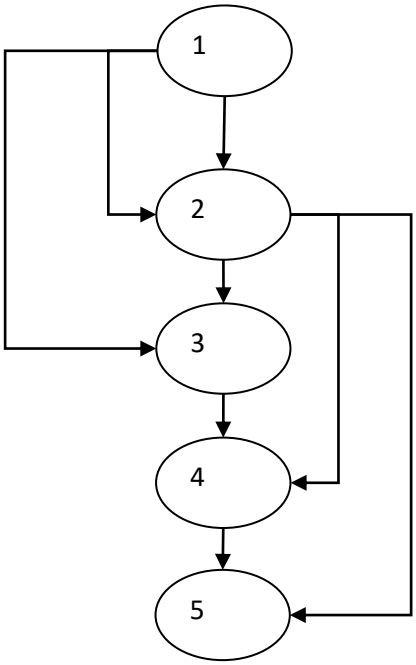
The **topological sort** algorithm takes a directed graph and returns an array of the nodes where each node appears *before* all the nodes it points to.

The ordering of the nodes in the array is called a *topological ordering*

Here's an example:



Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.



So [**1, 2, 3, 4, 5**] would be a topological ordering of the graph.

UNIT – III

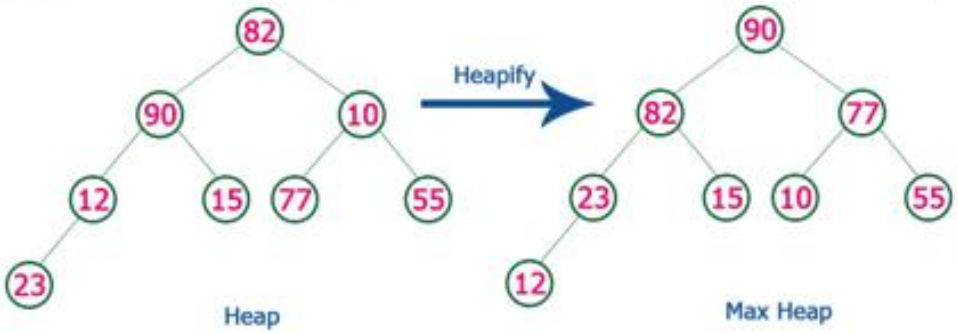
The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Min Heap.**
- **Step 3 -** Delete the root element from Min Heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Min Heap becomes empty.
- **Step 6 -** Display the sorted list.

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

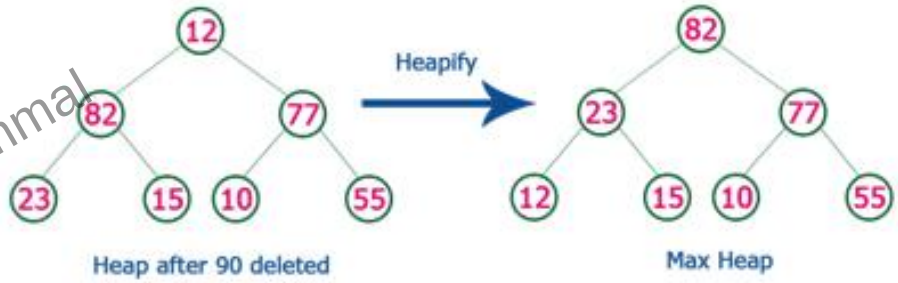
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82, 90**

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77, 82, 90**

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 55 with 15.

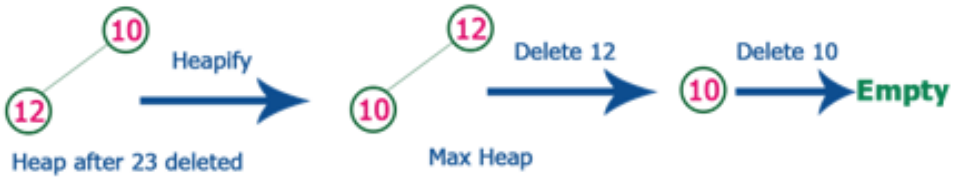
12, 15, 10, 23, **55, 77, 82, 90**

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, **23, 55, 77, 82, 90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Merge sort

Merge sort is based on divide and conquer algorithm

Algorithmic steps

Merge sort uses a divide-and-conquer approach:

1) Divide the array repeatedly into two halves

2) Stop dividing when there is single element left. By fact, single element is already sorted.

3) Merges two already sorted sub arrays into one.

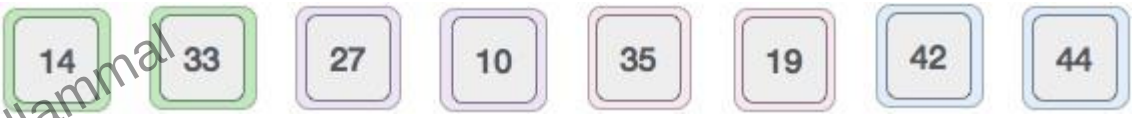
Divide steps

To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.



Sort and merge steps

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Binary search

- Binary search is a fast search algorithm with run-time complexity of O(log n). (n-number of items)
- This search algorithm works on the principle of divide and conquer.
- For this algorithm to work properly, the data collection should be in the **sorted form**

Steps of binary search

- Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. (location is returned)
- If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item.
- This process continues on the sub-array as well until the size of the subarray reduces to zero.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← item to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2


      if A[midPoint] > x     //  search in the left sub array
         set upperBound = midPoint - 1


      if A[midPoint] < x   // search in the right sub array)
         set lowerBound = midPoint + 1


      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while

end procedure
```

Example

For a binary search to work, it is mandatory for the target array to be sorted.

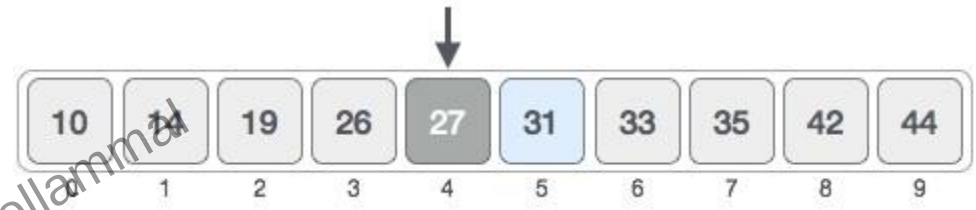
The following is our sorted array

Let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.(right sub array)

lowerBound = midPoint + 1

We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
```

```
mid = low + (high - low) / 2; 5+(9-5)/2=7=5+2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.(left sub array)

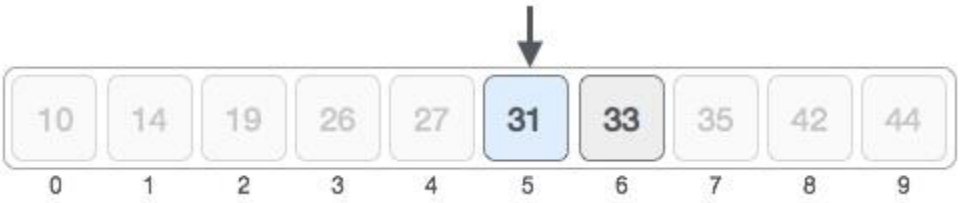
upperBound = midPoint - 1

High=mid-1

High=6


```
mid = low + (high - low) / 2   ; 5+(6-5)/2=5.5
```

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Finding minimum and maximum

Problem Statement

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array or a set of elements using divide and conquer algorithm.

Divide and conquer algorithms solves a problem using the following steps

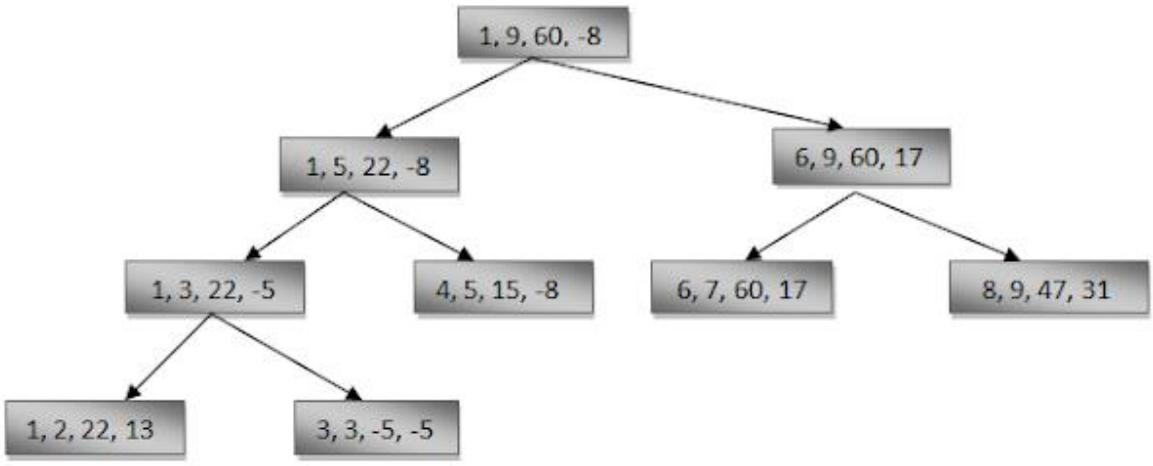
1.Divide: Break the given problem into sub-problems of same type.

2. Conquer: Recursively solve these sub-problems

3. Combine: Appropriately combine the answers

Example

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47



We see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new call to MaxMin, where i and j have the values 1, 5 and 6, 9, and thus split the set into two subsets of the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call).

Procedure


```
MaxMin(i, j, max, min)
{
//divide the problem into sub-problems and perform the procedure recursively
            // Find where to split the set.
            mid := ( i + j )/2;
            // Solve the sub-problems.
            MaxMin( i, mid, max, min );
            MaxMin( mid+1, j, max1, min1 );
            // Combine the solutions.
            if (max < max1) then max := max1;
            if (min > min1) then min := min1;

}
```