MC5301: ADVANCED DATA STRUCTURES AND ALGORITHMS

COURSE OBJECTIVES

- Understand and apply linear data structures-List, Stack and Queue.
- Understand the graph algorithms.
- Learn different algorithms analysis techniques.
- Apply data structures and algorithms in real time applications
- Able to analyze the efficiency of algorithm.

SYLLABUS

UNIT I LINEAR DATA STRUCTURES 9

Introduction - Abstract Data Types (ADT) – Stack – Queue – Circular Queue - Double Ended Queue - Applications of stack – Evaluating Arithmetic Expressions - Other Applications - Applications of Queue - Linked Lists - Singly Linked List - Circularly Linked List - Doubly Linked lists – Applications of linked list – Polynomial Manipulation.

UNIT II NON-LINEAR TREE STRUCTURES 9

Binary Tree – expression trees – Binary tree traversals – applications of trees – Huffman Algorithm - Binary search tree - Balanced Trees - AVL Tree - B-Tree - Splay Trees – Heap-Heap operations- -Binomial Heaps - Fibonacci Heaps- Hash set.

UNIT III GRAPHS 9

Representation of graph - Graph Traversals - Depth-first and breadth-first traversal - Applications of graphs - Topological sort – shortest-path algorithms - Dijkstra"s algorithm – Bellman-Ford algorithm – Floyd's Algorithm - minimum spanning tree – Prim's and Kruskal's algorithms.

UNIT IV ALGORITHM DESIGN AND ANALYSIS 9

Algorithm Analysis – Asymptotic Notations - Divide and Conquer – Merge Sort – Quick Sort - Binary Search - Greedy Algorithms – Knapsack Problem – Dynamic Programming – Optimal Binary Search Tree - Warshall"s Algorithm for Finding Transitive Closure.

UNIT V ADVANCED ALGORITHM DESIGN AND ANALYSIS 9

Backtracking – N-Queen's Problem - Branch and Bound – Assignment Problem - P & NP problems – NP-complete problems – Approximation algorithms for NP-hard problems – Traveling salesman problem-Amortized Analysis.

TOTAL : 45 PERIODS

REFERENCES:

1. Anany Levitin "Introduction to the Design and Analysis of Algorithms" Pearson Education, 2015
2. E. Horowitz, S.Sahni and Dinesh Mehta, "Fundamentals of Data structures in C++", University Press, 2007
3. E. Horowitz, S. Sahni and S. Rajasekaran, "Computer Algorithms/C++", Second Edition, University Press, 2007
4. Gilles Brassard, "Fundamentals of Algorithms", Pearson Education 2015
5. Harsh Bhasin, "Algorithms Design and Analysis", Oxford University Press 2015
6. John R.Hubbard, "Data Structures with Java", Pearson Education, 2015
7. M. A. Weiss, "Data Structures and Algorithm Analysis in Java", Pearson Education Asia, 2013

8. Peter Drake, “Data Structures and Algorithms in Java”, Pearson Education 2014
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms", Thrid Edition, PHI Learning Private Ltd, 2012
10. Tanaenbaum A.S.,Langram Y. Augestein M.J, “Data Structures using C” Pearson Education , 2004.
11. V. Aho, J. E. Hopcroft, and J. D. Ullman, “Data Structures and Algorithms”, Pearson Education, 1983

COURSE OUTCOMES (COs)

C201.1: Describe, explain and use abstract data types including stacks, queues and lists
C201.2: Design and Implement Tree data structures and Sets
C201.3: Able to understand and implement non linear data structures - graphs
C201.4: Able to understand various algorithm design and implementation

MAPPING BETWEEN COs, POs AND PSOs

COs	**PROGRAMME OUTCOMES (POs)**												**PSOs**	
	PO1	**P02**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**
C201.1	1	2	1	2	1	-	-	-	1	1	1	1	2	2
C201.2	2	2	1	1	1	-	-	-	1	1	1	1	2	1
C201.3	2	2	1	2	1	-	-	-	1	1	1	1	2	1
C201.4	2	2	2	2	1	-	1	-	1	2	1	1	2	2

RELATION BETWEEN COURSE CONTENTS WITH CO’s

S.No	**Knowledge level**	**COURSE CONTENT**	**Course Outcomes**
UNIT I		**LINEAR DATA STRUCTURES - 9 hrs**	
1	U,R	Introduction - Abstract Data Types (ADT)	C201.1
2	U, An,AP,C	Stack – Queue – Circular Queue - Ended Queue	
3	U, An, Ap	Applications of stack	
4	U, An, Ap	Evaluating Arithmetic Expressions	
5	An, Ap, E	Applications of Queue - Linked Lists - Singly Linked List - Circularly Linked List - Doubly Linked lists	
6	U, Ap, E, C	Applications of linked list	
7	U, An,AP,C	Polynomial Manipulation	
UNIT II		**NON-LINEAR TREE STRUCTURES - 9hrs**	
1	U, An,AP,C	Binary Tree – expression trees – Binary tree traversals	C201.2
2	U, An, Ap	Applications of trees	
3	U, R, C	Huffman Algorithm	
4	U, Ap, C	Binary search tree - Balanced Trees	
5	U, Ap, An	AVL Tree - B-Tree - Splay Trees	
6	U, Ap, C, E	Heap- Heap operations- -Binomial Heaps - Fibonacci Heaps	
7	U,R	Hash set	

UNIT III	**GRAPHS - 9 hrs**		
1	U, C	Representation of graph	C201.3
2	U, Ap	Graph Traversals - Depth-first and breadth-first traversal	
3	U, Ap, E, C	Applications of graphs	
4	U, Ap, C	Topological sort	
5	U, Ap, E, C	shortest-path algorithms - Dijkstra‟s algorithm – Bellman-Ford algorithm – Floyd's Algorithm	
6	U, Ap, E, C	Minimum Spanning Tree	
7	U, Ap, E, C	Prim's and Kruskal's Algorithms.	
UNIT IV	**ALGORITHM DESIGN AND ANALYSIS - 9 hrs**		
1	U,Ap, An, E	Algorithm Analysis – Asymptotic Notations	C201.4
2	U, Ap, R	Divide and Conquer – Merge Sort – Quick Sort	
3	U, Ap, C, E	Binary Search	
4	U,Ap, An, E	Greedy Algorithms – Knapsack Problem	
5	U,Ap, An, E	Dynamic Programming	
6	U,Ap, An, E	Optimal Binary Search Tree	
7	U,Ap, An, E	Warshall‟s Algorithm for Finding Transitive Closure	
UNIT V	**ADVANCED ALGORITHM DESIGN AND ANALYSIS - 9 hrs**		
1	U,Ap, An, E	Backtracking	C201.4
2	U,Ap, An, E	N-Queen's Problem	
3	U,Ap, An, E	Branch and Bound	
4	U,Ap, An, E	Assignment Problem	
5	U,Ap, An, E	P & NP problems – NP-complete problems - Approximation algorithms for NP-hard problems	
6	U,Ap, An, E	Traveling salesman problem	
6	U,Ap, An, E	Amortized Analysis	
ADDITIONAL TOPICS			
The Knight Problem using backtracking			C201.4
Finding a Hamiltonian circuit or disprove its existence in the graph			C201.3

R – Remember; Ap – Apply; An – Analyze; U – Understand, E- Evaluate ,C-Create

PART - A

UNIT – I

1. **Define data structure. What is the main advantage of data structure?**
   A data structure is a logical or mathematical way of organizing data. It is the way of organizing, storing and retrieving data and the set of operations that can be performed on that data.
   Eg.: Arrays, structures, stack, queue, linked list, trees, graphs.
2. **What are the different types of data structures.**
   Primitive Data Structure- It is basic data structure which is defined by the language and can be accessed directly by the computer.

Non Primitive Data Structure- Data structure emphasize on structuring of a group of homogenous or heterogeneous data item.

Linear Data Structure- A data structure which contains a linear arrangement of elements in the memory.

Non-Linear Data Structure- A data structure which represents a hierarchical arrangement of elements.

3. **Define Abstract Data Type.**

An Abstract data type is a data type that is organized in such a way that the specification of the objects and the specification of operations on the objects is separated from the representation of the objects and the implementation of the operations. In other words, ADT is a collection of values and a set of operations on those values. ADT is a mathematical tool
for specifying the logical properties of a datatype.

4. **Define an array. Mention the different kinds of arrays with which you can manipulate and represent data.**

An array is a group of related data items that shares a common name. In other words, we can say it is a collection of data items which are of same data type. The data items are stored in contiguous memory locations. There are three kinds of arrays present for the manipulation and representation of data.They are

1. One dimensional array.
2. Two dimentional array.
3. Multi dimentional array.

5. **A two dimensional array consisting of 8 rows and 3 columns is stored in a row major order. Compute the address of element A(4, 2). Base address is 1000 and word length is 2. Find the address of the same element in the column major representation.**

In Row major representation

Address(a[i, j]) = base address + [(i-l1) * (u2-l2+1) + (j-l2)] * element size
Here we can represent the array as a[0..7, 0..2]
Base address = 1000
l1=0, u1=7, l2=0, u2=2
I= 4, j=2
Element size = 2
Address(a[4,2]) = 1000 + [(4-0)*(2-0+1)+(2-0)] * 2
= 1000 + [4*3 + 2] *2
= 1028

In Column major representation

Address(a[i, j]) = base address + [(j-l2) * (u1-l1+1) + (i-l1)] * element size
Here we can represent the array as a[0..7, 0..2]
Base address = 1000
L1=0, u1=7, l2=0, u2=2
I= 4, j=2
Element size = 2
Address(a[4,2]) = 1000 + [(2-0) * (7-0+1) + (4-0)] * 2
= 1000 + [2 * 8 + 4] *2
= 1040

6. **How much memory is required for storing two matrices A(10,15,20) and B(11,16,21) where each element requires 16 bit for storage.**
   Number of elements in array A = 10*15*20 =3000
   Element Size = 16 bits.
   Memory required for storing A = 3000*16=48,000
   Number of elements in array A = 11*16*21=3696
   Element Size = 16 bits
   Memory Required for storing A = 3696 *16 = 59,136
   Total = 107136 bits = 107136/8 = 13,392 bytes.
7. **What are the differences between arrays and structures? (JAN 2012)**

ARRAYS	STRUCTURES
1.Array size should be mentioned during the declaration.	Declared using the keyword "struct".
2. Array uses static memory location.	Each member has its own memory location.
3. Each array element has only one part.	Only one member can be handled at a time.

8. **Define stack. Give some applications of stack.**
   A stack is an ordered list in which insertions and deletions are made at one end called the top. Stack is called as a Last In First Out(LIFO) data structure. Stack is used in Function call, Recursion and evaluation of expression.
9. **How do you check the stack full and stack empty condition?**

```
Void StackFull()
{
        If (top == maxsize-1)
            Printf("Stack is Full");
}

Void StackEmpty()
{
        If (top == -1)
            Printf("Stack is Empty");
}
```

10. **Define the terms: Infix, postfx and prefix.**
    - INFIX: It is a conventional way of writing an expression.The notation is
      <Operand><Operator><Operand>
      This is called infix because the operators are in between the operands.
      EXAMPLE: A+B
    - POSTFIX: In this notation the operator is suffixed by operands.
      <Operand><Operand><Operator>
      EXAMPLE: AB+
    - PREFIX: In this notation the operator preceeds the two operands.
      <Operator><Operands><Operand>
      EXAMPLE: +AB

11. What are the advantages in reverse polish (prefix and postfix notation) over polish (infix) notation?

The advantages in prefix & postfix notation over infix notation is:

The scanning of the expression is required in only one direction viz. from left to right and only once; where as for the infix expression the scanning has to be done in both directions.

For example, to evaluate the postfix expression abc*+, we scan from left to right until we encounter *. The two operands which appear immediately to the left of this operator are its operands and the expression bc* is replaced by its value.

12. Define queue and give its applications

A Queue is an ordered list in which all insertions take place at one end called the rear and all deletions take place at the opposite end called the front. The Queue is called as the FIFO data structure.

Applications of Queue:

1. It is used in batch processing of O.S
2. It is used in simulation
3. It is used in queuing theory
4. It is used in computer networks where the server takes the jobs of the clients using queuing strategy.

13. What is a circular queue? How do you check the queue full condition?

In circular queue, the elements are arranged in a circular fashion. Circular queue is a data structure which efficiently utilizes the memory space & the elements Q[0], Q[1], …, Q[n-1] are arranged in circular fashion such that Q[n-1] is followed by Q[0].

It returns queue full condition only when the queue does not have any space to insert new values. But ordinary queue returns queue full condition when the rear reaches the last position.

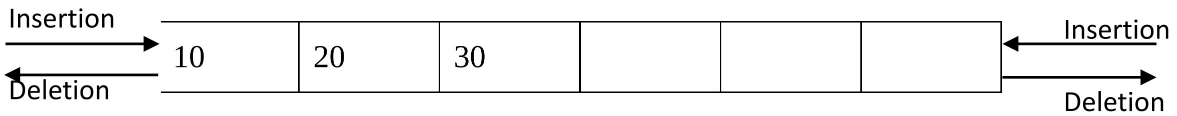
```
Void CircularQFull()
{
    if (front == (rear+1)%maxsize)
        printf("Circular Queue is Full");
}
```

14. Write an algorithm to count the nodes in a circular queue

```
int countcq()
{
        Count = 0;
        If (front = -1)
                Printf (" Queue is empty");
        Else
        {       i = front
                while (i !=rear)
                {
                      Count++;
                       i = (i+1)%maxsize;
                 }
                Count++;
        }
        Return(count);
    }
```

15. Define Dequeue.

Dequeue is a queue in which insertion and deletion can happen in both the ends (front & rear) of the queue.



16. What are the two kinds of dequeue?

Input restricted dequeue -- restricts the insertion of elements at one end (rear) only, but the deletion of elements can be done at both the ends of a queue.

Output restricted dequeue --Restricts the deletion of elements at one end (front) only, and allows insertion to be done at both the ends of a deque.

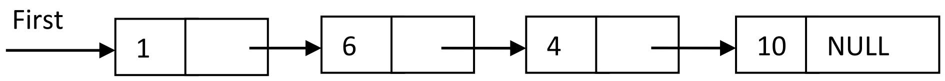
17. What is a priority queue?

A queue in which we are able to insert or remove items from any position based on some priority is referred to as priority queue.

18. Define Linked list and give its applications.

It is an ordered collection of homogeneous data elements. The elements of the linked list are stored in non contiguous memory locations. So each element contains the address of the next element in the list. The last node contains the NULL pointer which represents the end of the list.

Example:



Applications of Linked List:

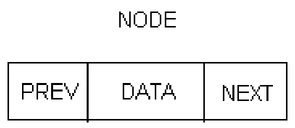
- It is used in polynomial manipulation.
- It is used for sparse matrix representation.

19. Compare array and linked list.

Array	Linked List
1. In an array, the successive elements are in contiguous memory locations	1. Successive elements in the list can be stored any where in the memory
2. Insertion & deletion operation requires lot of data movement.	2. No data movement during insertion & deletion.
3. The amount of memory needed to store the list is less.	3. More storage is needed because with each data item the link is also stored.
4. Follows static memory allocation	4. Follows dynamic memory allocation.

20. Define Doubly Linked List.

The Doubly linked list is a collection of nodes each of which consists of three parts namely the data part, prev pointer and the next pointer. The data part stores the value of the element, the prev pointer has the address of the previous node and the next pointer has the value of the next node.



In a doubly linked list, the head always points to the first node. The prev pointer of the first node points to NULL and the next pointer of the last node points to NULL.

21. What are the advantages of using doubly linked list over singly linked list?

The advantage of using doubly linked list is,it uses the double set of pointers.One pointing to the next item and other pointing to the preceeding item.This allows us to traverse the list in either direction.

22. List the advantages of linked list

Since linked list follows dynamic memory allocation, the list can grow dynamically, the insertion and deletion of elements into the list requires no data movement

UNIT-II

1. **Define tree.**

A tree is a finite set of one or more nodes such that there is a specially designated node called the root. The remaining nodes are partitioned into n>=0 disjoint sets T1, T2, …, Tn, where each of these sets is a tree. T1, …,Tn are called the subtrees of the root.

2. **Define the following terms: node, leaf node, ancestors, siblings of a node**

Node: Each element of a binary tree is called node of a tree. Each node may be a root of a tree with zero or more sub trees.

Leaf node: A node with no children (successor) is called leaf node or terminal node.

Ancestor: Node n1 is an ancestor of node n2 if n1 is either a father of n2 or father of some ancestor of n2.

Siblings: Two nodes are siblings if they are the children of the same parent.

3. **Define level of a node, degree of a node, degree of a tree, height and depth of a tree.**

Level of a node: The root node is at level 1. If a node is at level l, then its children are at level i+1.

Degree of a node: The number of sub trees of a node is called as degree of a node.

The degree of a tree is the maximum of the degree of the nodes in the tree.

The height or depth of a tree is defined to be the maximum level of any node in the tree.

4. **What are the ways to represent Binary trees in memory?**

1. Array representation (or) Sequential Representation.
2. Linked List representation (or) Node representation.

5. **Define binary tree.**

Binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains the single element called the root of tree. The other two subsets are themselves binary tree called the left and right sub tree of original tree. In other words, a binary tree is a tree in which each node can have a maximum of two children.

6. Define Full binary tree (or) Complete binary tree

A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes. In other words, all the levels in the binary tree should contain the maximum number of nodes.

7. Define strictly binary tree

If every non leaf node in a binary tree has non empty left and right sub trees then the tree is termed as strictly binary tree. In other words, a strictly binary tree contains leaf nodes and non leaf nodes of degree

8. List out few of the Application of tree data-structure?

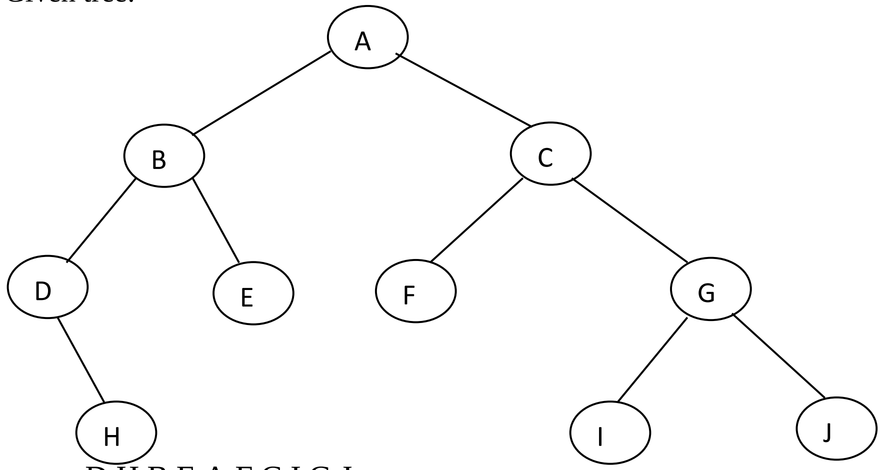
The applications of tree data-structure are the manipulation of Arithmetic expression, Symbol Table construction, Syntax analysis.

9. Define expression tree

An expression tree is built up from the simple operands and operators of an(arithmetic or logical) expression by placing the simple operands as the leaves of a binary tree and the operators as the interior nodes.

10. Traverse the given tree using Inorder, Preorder and Postorder traversals.

Given tree:

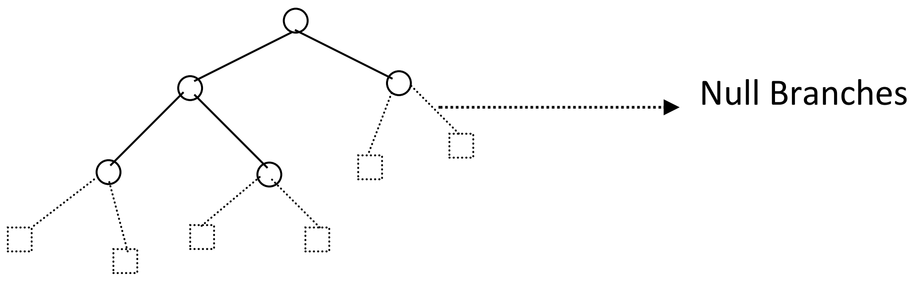


- Inorder : D H B E A F C I G J
- Preorder: A B D H E C F G I J
- Postorder: H D E B F I J G C A

11. How many null branches can a binary tree have with 20 node?

21 null branches

Let us take a tree with 5 nodes (n=5)



It will have only 6 (ie,5+1) null branches. In general, a binary tree with n nodes has exactly n+ 1 null node. Thus a binary tree with 20 nodes will have 21 null branches.

12. What is a binary search tree?

A binary search tree is a binary tree. It may be empty. If it is not empty then, it satisfies the following properties.

1. Every element has a key & the keys are distinct.
2. The keys in the left sub tree is smaller than the key in the root.
3. Keys in the right sub tree is larger than the key in the root.
4. The left & right sub trees are also BST.

13. How will you construct binary search tree?

a. Make the first node as the root node.

b. To insert the next node into the BST, search for the value in the BST. If the value is found in the BST, then a duplicate value cannot be inserted into the BST.

c. If the element is not found, add the element at the point where the search becomes unsuccessful.

14. Define the term skewed tree?

In skewed tree all the nodes are skewed in one direction either left or right.

Left Skewed Tree: A tree in which all nodes are skewed in left direction.

Right Skewed Tree: A tree in which all nodes are skewed in right direction.

15. What is the maximum number of nodes in level i of a binary tree and what is the maximum number of nodes in a binary tree of depth k?

The maximum number of nodes in level i of a binary tree = 2^{i-1}

The maximum number of nodes in a binary tree of depth k = 2^k-1, where $k>0$

16.What are the non-linear data structures? (JAN 2014)

Non-Linear Data Structure- A data structure which represents a hierarchical arrangement of elements. Examples: Graphs and trees.

17. Define balanced search tree.

Balanced search tree have the structure of binary tree and obey binary search tree properties with that it always maintains the height as O(log n) by means of a special kind of rotations. Eg. AVL, Splay, B-tree.

18.What are the drawbacks of AVL trees?

The drawbacks of AVL trees are

- Frequent rotations
- The need to maintain balances for the tree's nodes
- Overall complexity, especially of the deletion operation.

19. Define B-tree?

A B-tree of order m in an m-way search tree that is either empty or is of height ≥ 1 and

1. The root node has at least 2 children
2. All nodes other than the root node and failure nodes have at least m/2 children.
3. All failure nodes are at same level.

20. Explain AVL rotation.

Manipulation of tree pointers is centered at the pivot node to bring the tree back into height balance. The visual effect of this pointer manipulation so to rotate the sub tree whose root is the pivot node. This operation is referred as AVL rotation.

21. What are the different types of Rotation in AVL Tree?

Two types of rotation are

1. single rotation
2. double rotation.

22. Explain Hashing.

Hashing is a technique used to identify the location of an identifier 'x' in the memory by some arithmetic functions like f(x), which gives address of 'x' in the table.

23. Explain Hash Function. Mention Different types of popular hash function.

Hash Function takes an identifier and computes the address of that identifier in the hash table.

1.Division method
2.Square method
3.Folding method

24..Define Splay Tree.

A splay tree is a self-adjusting binary search treewith the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time.

25. What are the different rotations in splay tree?

- ❖ Zig Rotation.
- ❖ Zag Rotation
- ❖ Zig-Zag Rotation.
- ❖ Zag-Zig Rotation
- ❖ Zig-Zig Rotation
- ❖ Zag-Zag- Rotation

26.Write short notes on Heap.

Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then −
key(α) ≥ key(β)

27.Define Binomial Heap.

A Binomial Heap is a collection of Binomial Trees A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1, and making one as leftmost child of other.
A Binomial Tree of order k has following properties.
a) It has exactly 2k nodes.
b) It has depth as k.
c) There are exactly kCi nodes at depth i for i = 0, 1, . . . , k.
d) The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

28.Define Fibonacci Heaps.

Fibonacci heap is a data structure for priority queue operations, consisting of a collection of heap-ordered trees. It has a better amortized running time than many other priority queue data structures including the binary heap and binomialheap.

29.Write notes on Hash Set.

- ❖ Implements Set Interface.
- ❖ Underlying data structure for HashSet is hashtable.
- ❖ As it implements the Set Interface, duplicate values are not allowed.
- ❖ Objects that you insert in HashSet are not guaranteed to be inserted in same order.
- ❖ Objects are inserted based on their hash code.
- ❖ NULL elements are allowed in HashSet.
- ❖ HashSet also implements Searlizable and Cloneable interfaces.

UNIT-III

1. **Write the concept of Prim's spanning tree.**
   Prim's algorithm constructs a minimum spanning tree through a sequence of expanding sub trees. The initial sub tree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices.
   On each iteration, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed
2. **What is the purpose of Dijikstra's Algorithm?**
   Dijikstra's algorithm is used to find the shortest path between sources to every vertex. This algorithm is applicable to undirected and directed graphs with nonnegative weights only.
3. **How efficient is prim's algorithm?**
   It depends on the data structures chosen for the graph itself and for the priority queue of the set $V\text{-}V_T$ whose vertex priorities are the distances to the nearest tree vertices.
4. **Mention the two classic algorithms for the minimum spanning tree problem.**
   - ✓ Prim's algorithm
   - ✓ Kruskal's algorithm
5. **What is the Purpose of the Floyd algorithm?**
   The Floyd's algorithm is used to find the shortest distance between every pair of vertices in a graph.
6. **What are the conditions involved in the Floyd's algorithm?**
   - ❖ Construct the adjacency matrix.
   - ❖ Set the diagonal elements to zero
   - ❖ $A_k[i,j] = \min \begin{cases} A_{k-1}[i,j] \\ A_{k-1}[i,k] \text{ and } A_{k-1}[k,j] \end{cases}$
7. **Write the concept of kruskal's algorithm**.
   Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph G=(V,E) as an acyclic sub graph with |V|-1 edges for which the sum of the edge weights is the smallest. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of sub graphs, which are always acyclic but are not necessarily connected on the intermediate stages of the algorithm. The algorithm begins by sorting the graph's edges in non decreasing order of their weights. Then, starting with the empty sub graph, it scans this sorted list, adding the next edge on the list to the current sub graph if such an inclusion does not create a cycle and simply skipping the edge otherwise.
8. **What is the difference between dynamic programming with divide and conquer method?**
   Divide and conquer divides an instance into smaller instances with no intersections whereas dynamic programming deals with problems in which smaller instances overlap. Consequently divide and conquer algorithm do not explicitly store solutions to smaller instances and dynamic programming algorithms do.
9. **State two obstacles for constructing minimum spanning tree using exhaustive-search approach.**
   - ❖ The number spanning tree grows exponentially with the graph size

- ❖ Generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a minimum spanning tree for a weighted graph by using one of several efficient algorithms available for this problem

10. Define spanning tree and minimum spanning tree problem.

A spanning tree of a connected graph is its connected acyclic sub graph that contains all the vertices of the graph. A minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

11. Define the single source shortest paths problem.

Dijkstra's algorithm solves the single-source shortest-path problem of finding shortest paths from a given vertex (the source) to all the other vertices of a weighted graph or digraph. It works as Prim's algorithm but compares path lengths rather than edge lengths. Dijkstra's algorithm always yields a correct solution for a graph with nonnegative weights

12. Mention the methods for generating transitive closure of digraph.

- ✓ Depth First Search (DFS)
- ✓ Breadth First Search (BFS)

13. What do you meant by graph traversals?

Graph traversal (also known asgraph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversalsare classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal

14. Define Depth First Search DFS

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

15. Write down the steps involved in DFS

Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty

16. Define Breadth First Search (BFS)

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.

17. Write down the steps involved in Breadth First Search (BFS)

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

18. Define graph data structure

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges. Formally, a graph is a pair of sets (V, E), where V is the set of vertices and Eis the set of edges, connecting the pairs of vertices.

19. **Define topological sorting**
    Topological sorting of vertices of a **Directed Acyclic Graph** is an ordering of the vertices v1,v2,...vn in such a way, that if there is an edge directed towards vertex vj from vertex vi, then vi comes before vj.
20. **Define Memory function techniques**
    The memory function technique seeks to combine strengths of the top-down and bottom-up approaches to solving problems with overlapping sub problems. It does this by solving, in the top-down fashion but only once, just necessary sub problems of a given problem and recording their solutions in a table.

UNIT-IV

1. **Define Algorithm.**
   An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
2. **Define order of an algorithm.**
   The order of an algorithm is a standard notation of an algorithm that has been developed to represent function that bound the computing time for algorithms. The order of an algorithm is a way of defining its efficiency. It is usually referred as Big O notation.
3. **What are the features of efficient algorithm?**
   - ✓ Free of ambiguity
   - ✓ Efficient in execution time
   - ✓ Concise and compact
   - ✓ Completeness
   - ✓ Definiteness
   - ✓ Finiteness
4. **Define Asymptotic Notations.**
   The notation that will enable us to make meaningful statements about the time and space complexities of a program. This notation is called asymptotic notation. Some of the asymptotic notation are 1. Big Oh notation, 2. Theta notation, 3. Omega notation, 4. Little Oh notation.
5. **What is best-case efficiency?**
   The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input or inputs for which the algorithm runs the fastest among all possible inputs of that size.
6. **Define divide and conquer design technique**

- ❖ A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size
- ❖ The smaller instances are solved
- ❖ If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

7. **List out some of the stable and unstable sorting techniques.**
   Stable sorting techniques includes Bubble sort, Insertion sort, Selection sort, Merge sort and Unstable sorting techniques includes Shell sort, Quick sort, Radix sort, Heap sort

8. **Define Knapsack problem**
   Given n items of known weights w1…wn and values v1…vn and knapsack of capacity W. The aims is to find the most valuable subset if the items that fit into the knapsack. The exhaustive search approach to knapsack problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset to identify feasible subsets and finding a subset of the largest value among them.
9. **Define merge sort.**
   The merge sort algorithm divides a given array A[0..n-1] by dividing it into two halves A[0.. n/2-1] and A[n/2-..n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.
10. **Define quick sort**
    Quick sort employs a divide-and-conquer strategy. It starts by picking an element from the list to be the "pivot." It then reorders the list so that all elements with values less than the pivot come before the pivot, and all elements with values greater than the pivot come after it (a process often called "partitioning"). It then sorts the sub-lists to the left and the right of the pivot using the same strategy, continuing this process recursively until the whole list is sorted
11. **What is a pivot element?**
    The pivot element is the chosen number which is used to divide the unsorted data into two halves. The lower half contains less than value of the chosen number i.e. pivot element. The upper half contains greater than value of the chosen number i.e. pivot element. So the chosen number is now sorted.
12. **Define Binary Search**
    Binary search is a efficient algorithm for searching in a sorted array. It works by comparing a search key K with the array"s middle element A[m]. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if K<a[m] and for the second half if K>A[m]
13. **Define dynamic programming.**
    Dynamic programming is a technique for solving problems with overlapping sub problems. Rather than solving overlapping sub problems again and again, dynamic programming suggests solving each of the smaller sub problems only once and recording the results in a table from which a solution to the original problem can then be obtained.
14. **Define Optimal Binary Search Tree (OBST).** ***(June 06)***
    Dynamic programming can be used for constructing an optimal binary search tree for a given set of keys and known probabilities of searching for them. If probabilities of searching for an element of a set are known, the average number of comparisons in a search will have smallest possible value in an OBST.
15. **State greedy technique.**
    The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
16. **Write down the optimization technique used for Warshall's algorithm. State the rules and assumptions which are implied behind that.**
    Optimization technique used in Warshall's algorithm is Dynamic programming. Dynamic programming is a technique for solving problems with overlapping sub problems. Typically, these sub problems arise from a recurrence relating a solution to a given

problem with solutions to its smaller sub problems of the same type. Dynamic programming suggests solving each smaller sub problem once and recording the results in a table from which a solution to the original problem can be then obtained.

17. Define objective function and optimal solution

To find a feasible solution that either maximizes or minimizes a given objective function. It has to be the best choice among all feasible solution available on that step.

18. Define knapsack problem using dynamic programming.

Designing a dynamic programming algorithm for the knapsack problem: given n items of known weights $w_1. . . w_n$ and values $v_1, . . . , v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. We assume here that all the weights and the knapsack capacity are positive integers; the item values do not have to be integers

19. Mention different algorithm design techniques

- ❖ Methods of specifying an algorithm
- ❖ Proving an algorithms correctness
- ❖ Analyzing an algorithm
- ❖ Coding an algorithm

20. Mention the two properties of sorting algorithms

- ❖ A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
- ❖ An algorithm is said to be in place if it does not require extra memory

UNIT-V

1. On what basis problems are classified?

Problems are classified into two types based on time complexity. They are

- ❖ Polynomial (P) Problem
- ❖ Non-Polynomial (NP) Problem

2. Define Polynomial (P) problem

Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called polynomial.

3. Define Non Polynomial (NP) problem

Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called nondeterministic polynomial

4. Give some examples of Polynomial problem

- ❖ Selection sort
- ❖ Bubble Sort
- ❖ String Editing
- ❖ Factorial
- ❖ Graph Coloring

5. Give some examples of Non-Polynomial problem

- ▪ Travelling Salesman Problem
- ▪ Knapsack Problem.

6. Define backtracking

The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed.

7. Define state space tree

It is convenient to implement this kind of processing by constructing a tree of choices being made, called the state-space tree. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution; the nodes of the second level represent the choices for the second component, and so on

8. When a node in a state space tree is said to promising and non promising?

A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called non promising. Leaves represent either non promising dead ends or complete solutions found by the algorithm

9. Define n-queens problem

The problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal

10. Define branch and bound method

- Branch and bound is an algorithm that enhances the idea of generating a state space tree with idea of estimating the best value obtainable from a current node of the decision tree
- If such an estimate is not superior to the best solution seen up to that point in the processing, the node is eliminated from further consideration

11. How NP-hard problems are different from NP-Complete?

NP-hard : If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.

NP-Complete: A problem that is NP-complete has the property that it can be solved in polynomial time if all other NP-complete problems can also be solved in polynomial time

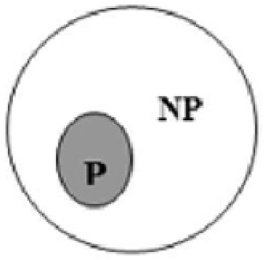
12. Define decision problem

Any problem for which the answer is either zero or one is called a decision problem. An algorithm for a decision problem is termed a Decision algorithm

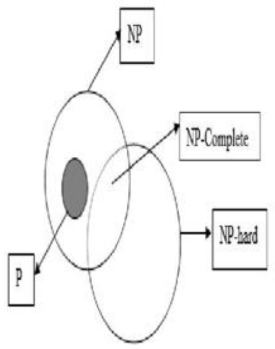
13. Define optimization Problem

Any problem that involves the identification of an optimal (maximum or minimum) value of a given cost function is known as an optimization problem. An Optimization algorithm is used to solve an optimization problem

14. Mention the relation between P and NP



15. Mention the relation between P, NP, NP-Hard and NP Complete Problem



16. Define NP hard problems

If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.

17. Define NP complete problems

A problem that is NP-complete has the property that it can be solved in polynomial time if all other NP-complete problems can also be solved in polynomial time

18. Define assignment problem

Assignment problem is the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible

19. What do you meant by amortized analysis ?

Amortized analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation

20. What are the examples for amortized analysis?

The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

PART-B

UNIT-I

1. Write the algorithm for performing operations in a stack. Trace your algorithm with suitable example

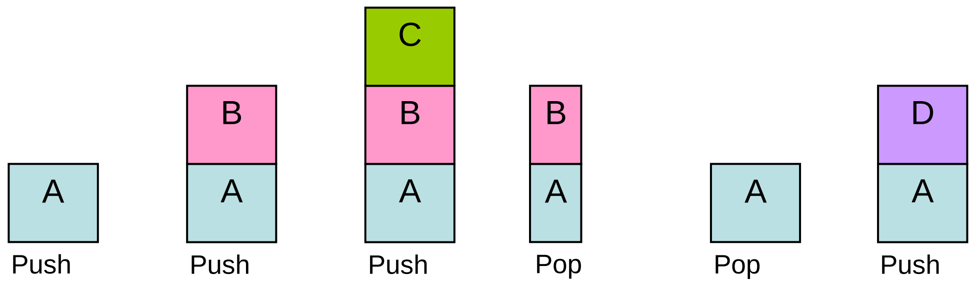
Stack

- A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end called the top of the stack
- Stack is a linear data structure which follows Last-in First-out principle, in which both insertion and deletion occur at only one end of the list called the top.
- The insertions operations is called push and deletion operations is called pop operation. Every insertion stack pointer is incremented by one, every deletion stack pointer will be decremented by one

Operations on stack

Push:- the process of inserting a new element to the top of the stack. For every push operations the top is incremented by one.

Pop:- Pop removes the item in the top of the stack



Exceptional conditions

Overflow:- Attempt to insert an element when the stack is full is said to be overflow.

Underflow:- Attempt to delete an element ,when the stack is said to be underflow.

Algorithm for insertion(Push):

```
Void push( int s[10], int top, int x)
{
// s-> stack, // top -> points the top
//x-> element to be inserted into the stack
If (top == MAXSIZE -1)
        Printf("Stack Full");
Else
        {top = top +1;s[top] = x;      }}
```

Algorithm for deletion(pop):

```
int pop(int s[10], int top, int x)
        {       // s-> stack
                //top-> points the top of the stack
                // x->int x;
        If (top== -1)
        {       printf(" Stack Empty");
                return (-1);
        }Else
        {x = s[top];
                top = top-1;    }
```

Algorithm for display:

```
void display(int s[10], int top)
{
        Int i;
        If ( top ==-1)
        Printf("Stack Empty");
else
        {
                For( i = 0; i<=top; i++)
                Printf("%d", s[i]);
        }
}
```

2. **Evaluate the postfix expression that is obtained in (i) for the values A = 5, B =3, C= 2, D= 2, E = 4, F = 3, G = 8, H=6**

Evaluation of postfix expression:-

Postfix : ABC↑D↑+EFGH−/+*

A=5, B=3, C=2, D=2, E=4, F=3, G=8, H=6.

I/P char	Stack	Remarks.
A	5	Push the value (operand)
B	3 5	operand — push.
C	2 3 5	operand — push

↑	9
	5
		operator — pop & perform operation
		op2 = 2 ; op1 = 3
		val= 3 ↑ 2 = 9
		Push val.

~~↑~~		~~op2=9,~~
D	2 9 5	Push

↑	81
	5
		op2=2 ; op1=9
		val = 9 ↑ 2 = 81
		push val.

+	86	op2= 81; op1=5
		val = 5+81=86
		push val.

I/p char	Stack	Remarks.
E	4 86	push
F	3 4 86	push
G	8 3 4 86	push
H	6 8 3 4 86	push

–	2
	3
	4
	86
		op2 = 6, op1 = 8
		val = 8 – 6 = 2 . push val.

/	1.5
	4
	86
		op2 = 2, op1 = 3
		val = 3/2 = 1.5 . push val.

+	5.5
	86
		op2 = 1.5 ; op = 4
		val = 4 + 1.5 = 5.5 , push

*	473	op2 = 5.5; op1 = 86
		val = 86 * 5.5 , push
		= 473

#	pop the result.	$\boxed{\text{Result} = 473}$

3. Write the algorithms for PUSH, POP and change operations on stack. Using these algorithms, how do you check whether the given string is a palindrome?

Algorithm for insertion(Push):

```
Void push( int s[10], int top, int x)
{
// s-> stack
// top -> points the top
//x-> element to be inserted into the stack
If (top == MAXSIZE -1)
        Printf("Stack Full");
Else
        {
                top = top +1;
                s[top] = x;
        }
}
```

Algorithm for deletion(pop):

```
int pop(int s[10], int top, int x)
        {
                // s-> stack
                //top-> points the top of the stack
                // x->int x;
        If (top== -1)
        {
                printf(" Stack Empty");
                return (-1);
        }
        Else
        {
                x = s[top];
                top = top-1;
        }
```

Algorithm for change operation:

```
void change(int s[10], intpos, intval)
        {
                // s-> stack
                //top-> points the top of the stack
                // x->int x;
        If (pos>top)
        {
                printf(" Change operation is not possible");
        }
        Else
        {
                S[pos] = val;
        }
```

Algorithm for Checking for Palindrome

```
Void palindrome(char str[])
{
        charrevstr[20];
        i = 0;
        While (str[i] != '\0')
        {
        Push(str[i])
        i++;
        }
        i=0;
        While (!stackempty())
        {
                revstr[i] = pop()
                i++;
        }
        if (strcmp(str, revstr) == 0)
        printf("The given string is a palindrome");
        else
printf("The given string is not a palindrome")
```

4. Write the algorithm for converting infix expression to postfix expression with the suitable example

Infix to postfix Conversion:

1. Fully parenthesise the expression according to the priority.
2. Move all operators so that they replace their corresponding to right paranthesis
3. Delete all the paranthesis

Priority

Operator	**Priority**
Brackets	1
Unary -	2
*,/,%	3
+,-	4

<,>,<=,>=	5
==, !=	6
&&	7
||	8

Algorithm from infix to postfix:

```
1. Read the infix expression 1 character at a time and repeat the steps 2 to 5 until it
   encounters the delimiter.
2. If the ( character is an operand)
   Append it to the postfix string
3. else if the ( character is '(')
   push it into the stack
4. else if ( character is ')')
   pop all the elements from the stack and append it to the postfix string till it encounter '('.
   Discard both paranthesis in the output.
5. else if ( the character is an operator)
   {
       While ( stack not empty and priority of(top element in the stack is higher =
   priority of the input character))
       Pop the operator from the stack and append  it to the postfix string
   }
   Push the operator into the stack
   }
6. While(Stack is not empty)
   Pop the symbols from the stack and append it to the postfix expression
```

5. Write the algorithm for evaluating the postfix expression with the suitable example.

Evaluation of expression

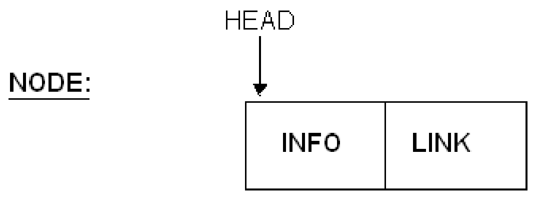
Algorithm

```
Step 1: Read the input postfix string one character at a time till the end of input
    While( not end of input)
        {
            Symbol = next input character
        If(symbol is an operand)
        Push symbol into the stack
        Else /* symbol is an operator */
        {
        Operand 2 = pop element from the stack;
        Operand 1 = pop element from the stack;
        Value = result of applying symbol to operand1 and operand2
        Push the value into the stack
        }
    }
```

Step 2: Pop the result from the stack.

6. Explain the algorithm for implementing Singly Linked list

A singly linked list is a linked list in which each node contains only one link pointing to the next node in the list.



In a singly linked list, the first node always pointed by a pointer called HEAD. If the link of the node points to NULL, then that indicates the end of the list.

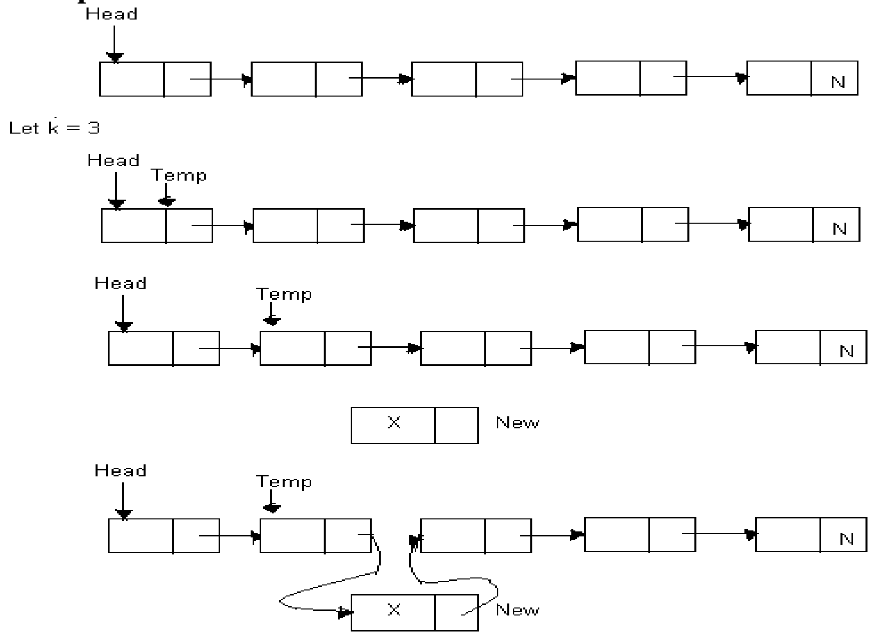
Algorithm for Creation:

```
void create()
{
node *t;
inti,n;
printf("\nenter the no. of elements in the list");
scanf("%d",&n);
first=NULL;
for(i=1;i<=n;i++)
{
t=(node*)malloc(sizeof(node));
scanf("%d",&t->data);
t->link=NULL;
if(first==NULL)
first=last=t;
else
{
last->link=t;
last=t;
}
}
}
```

Algorithm for insert operation:

```
void insert(intpos,intval)
{
int i;
node *curr,*prev,*t;
t=(node*)malloc(sizeof(node));
t->data=val;
t->link=NULL;
curr=first;
i=1;
while(curr!=NULL&&i<pos)
{
prev=curr;
curr=curr->link;
i++;
}
if(i==1)
{
t->link=first;
first=t;
}
else
{
prev->link=t;
t->link=curr;
}
}
```

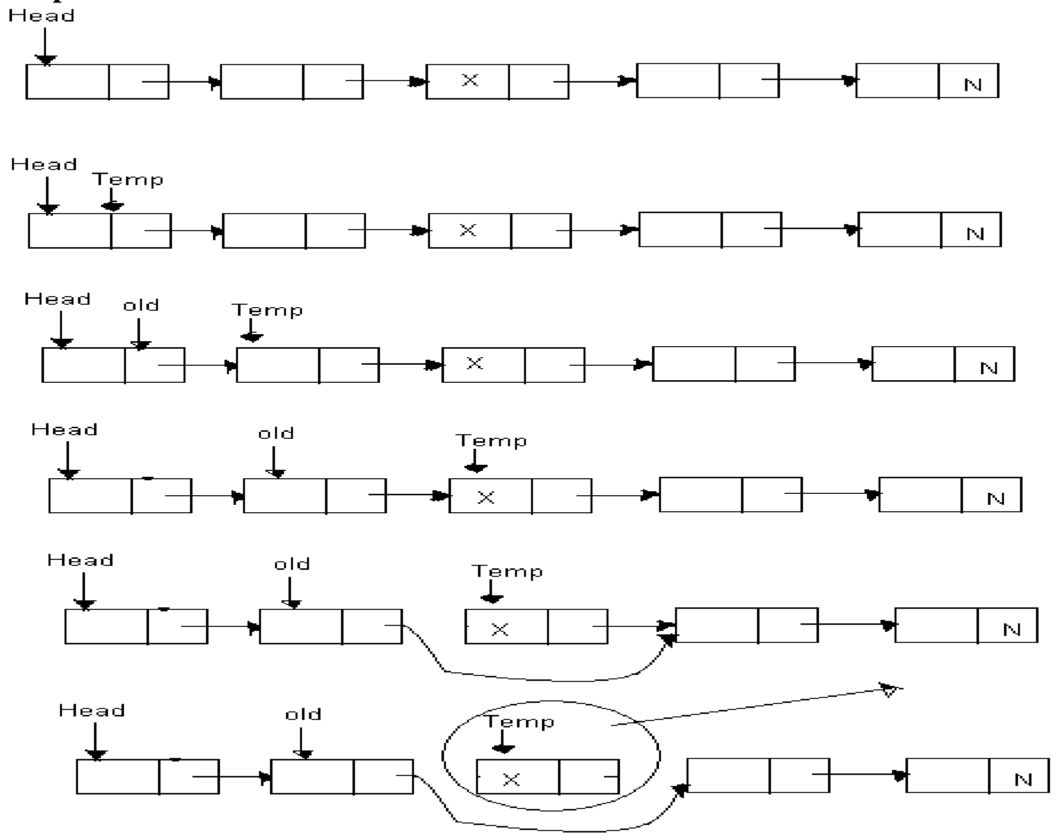
Example:



Algorithm for delete operation:

```
voiddelet(int x)
{
node *curr,*prev,*t;
curr=first; prev=NULL;
while(curr!=NULL&&curr->data!=x)
{
prev=curr;
curr=curr->link;
}
if(curr==NULL)
printf("\n elememt not found");
else
if(curr==first)
{
t=first;
first=first->link;
}
else
{
t=curr;
prev->link=curr->link;
}
free(t);}
```

Example:



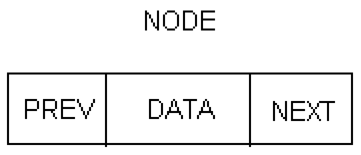
Algorithm for Traversing a Singly Linked List:

```
void traverse()
{
```

```
node *curr;
curr=first;
if(curr==NULL)
{
printf("\nlist is empty");
}
else
{
while(curr->link!=NULL)
{
printf("%d->",curr->data);
curr=curr->link;
}
}
printf("%d\n",curr->data);
}
```

7. Explain creation, insertion and deletion of doubly linked list with example

The Doubly linked list is a collection of nodes each of which consists of three parts namely the data part, prev pointer and the next pointer. The data part stores the value of the element, the prev pointer has the address of the previous node and the next pointer has the value of the next node.



In a doubly linked list, the head always points to the first node. The prev pointer of the first node points to NULL and the next pointer of the last node points to NULL.

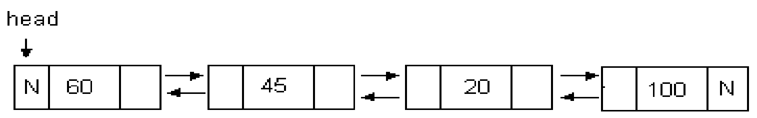
Algorithm for Creation:

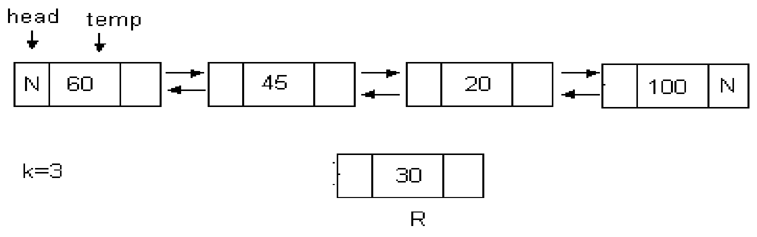
```
void create()
{
node *t;
inti,n;
printf("\nenter the no. of elements in the list");
scanf("%d",&n);
first=NULL;
for(i=1;i<=n;i++)
{
t=(node*)malloc(sizeof(node));
scanf("%d",&t->data);
t->llink=NULL;
t->rlink=NULL;
if(first==NULL)
first=last=t;
else
{
last->rlink=t;
t->llink=last;
last=t;
}
}
}
```

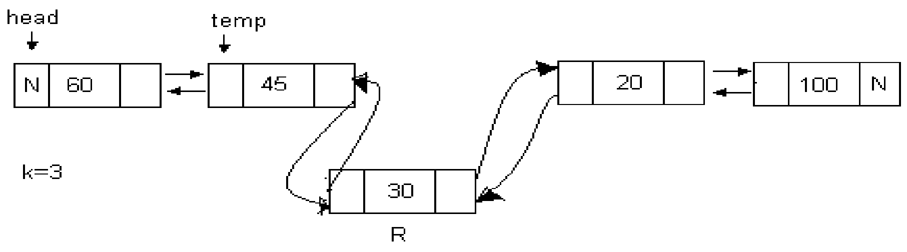
Algorithm for insertion:

```
void insert(intpos,intval)
{
int i;
node *t,*curr,*prev;
t=(node*)malloc(sizeof(node));
t->data=val;
t->llink=NULL;
t->rlink=NULL;
curr=first;
i=1;
while(curr!=NULL&&i<pos)
{
curr=curr->rlink;
i++;
}
if(curr==first)
{
t->rlink=first;
first->llink=t;
first=t;
}
else if(curr==NULL)
{
last->rlink=t;
t->llink=last;
last=t;
}
else
{
curr->llink->rlink=t;
t->llink=curr->rlink;
t->rlink=curr;
curr->llink=t;
}
}
```

Example:



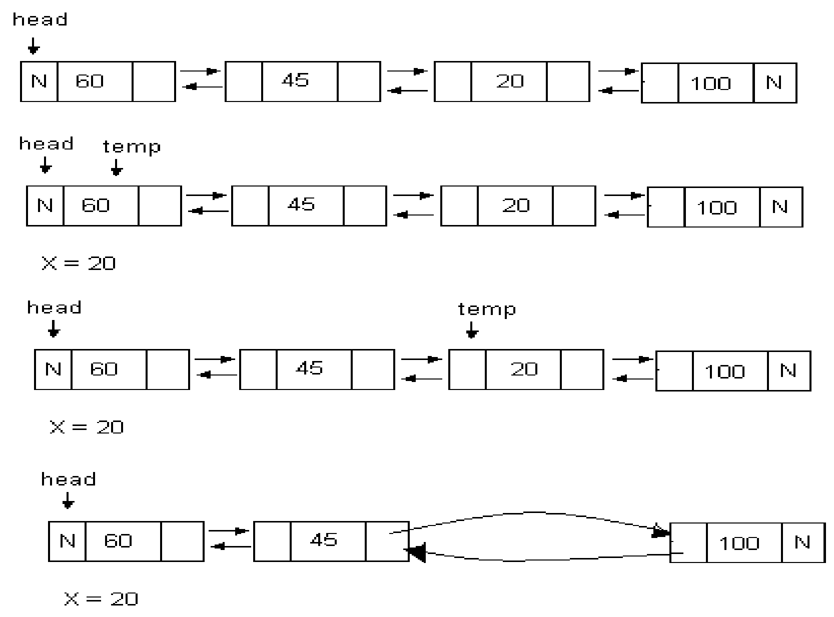




Algorithm for Deletion:

```
voiddelet(int x)
{
node *curr,*t;
curr=first;
while(curr!=NULL&&curr->data!=x)
curr=curr->rlink;
if(curr==NULL)
printf("\n element not found");
else if(curr==first)
{
t=first;
first=first->rlink;
first->llink=NULL;
}
else if(curr==last)
{
t=last;
last=last->llink;
last->rlink=NULL;
}
else
{
t=curr;
curr->llink->rlink=curr->rlink;
curr->rlink->llink=curr->llink;
}
free(t);
}
```

Example:



8. Implement a stack using doubly linked lists

Implementing Stack using Doubly Linked List:
Algorithm for Push:

```
void push(int x)
{
node*t;
t=(node*)malloc(sizeof(node));
t->data=x;
t->llink=NULL;
t->rlink=NULL;
if(top==NULL)
top=t;
else
{
t->rlink=top;
top->llink = t
top=t;
}
printf("\n");
printf("\n the element is pushed \n");
}
```

Algorithm for pop operation:

```
int pop()
{
node*t;
int x;
if (top==NULL)
{
printf("\n");
printf("stack empty \n");
return(-1);
}
else
{
x=top->data;
t=top;
top=top->rlink;
top->llink = NULL;
free(t);
return(x);
}}
```

Algorithm for Display

```
void display()
{
node*curr;
curr=top;
while(curr !=NULL)
{
printf("\n%d",curr->data);
curr=curr->rlink;
}
}
```

9. Construct a dequeue data structure in which the following operations to be implemented

Push(X,D) : Insert X on the front end of deque D
Pop(D) : Remove the front item from deque D and return it
Inject(X,D) : Insert item X on the rear end of deque D
Eject(D) : Remove the rear item from deque D and return it (JAN 2012)

```
Struct node
{
Int data;
    Struct node *link;
};
Structdequeue
{
Struct node *front;
Struct node *rear;
};
```

Push(X, D)

```
/* Insert X on the front end of deque D */
void push(int X, structdequeue *D)
{
    struct node *temp;
int *q;
temp = (struct node *) malloc(sizeof(struct node));
    temp->data = X;
    temp->link = NULL;
    if (D->front == NULL)
            D->front=D->rear =  temp;
    Else
    {
            Temp->link= D->front;
            D->front = temp;
    }
}
```

Pop(D) :

/* Remove the front item from deque D and return it */

```
int Pop(structdequeue *D)
{
    Struct node *temp = D->front;
    Int item;
    If (temp==NULL)
    {
            Printf("Queue is empty");
            Return 0
    }
    Else
    {
            Temp = D->front;
            Item = temp->data;
            p->front = temp->link;
            free(temp);
            if (temp == NULL)
                    D->rear = NULL;
            Return(item)
    }
}
```

Inject(X,D) :
/* Insert item X on the rear end of deque D */

```
Void insert(int X, struct node *D)
{
    Struct node *temp;
    Temp = (struct node *) malloc(sizeof( struct node));
    Temp->data = item;
    Temp->link = NULL;
    If (D->front == NULL)
            D->front = temp;
    Else
            D->rear->link = temp;
    D->rear = temp;
}
```

Eject(D) :
/*Remove the rear item from deque D and return it*/

```
IntEject(struct node *D)
{
    Struct node *temp, *rleft, *q;
    Int item;
    Temp = D->front;
    If (D->rear ==  NULL)
    {
            Printf("Queue is empty");
            Return 0;
    }
    Else
    {
            While (temp != D->rear)
            {
                    Rleft = temp;
                    Temp = temp->link;
            }
            q = D->rear;
            Item= q->data;
            Free(q);
            D->rear = rleft;
            D->rear->link = NULL;
            if ( D->rear ==NULL)
```

```
                D->front = NULL;
            Return(item);
        }
    }
```

10. Explain Circular queue operations with algorithm.

Algorithm for Insertion:

```
void insert(int x)
{
if (front == (rear+1)%MAZSIZE)
printf("\n Circular queue is full");
else
{
rear = (rear+1)%MAXSIZE;
q[rear]=x;
if(front==-1)
front=0;
}
}
```

Algorithm for Deletion:

```
intdelet()
{
if (front == -1)
{
printf("Circular Q is empty");
return(-1);
}
else
{
x=q[front];
if(front==rear)
front=rear=-1;
else
front=(front+1)%MAXSIZE;
return(x);
}
```

Algorithm for displaying the elements:

```
void display()
{
int i;
if(front==-1)
printf("\n Circular queue is empty");
else
{
if (front<=rear)
{
for(i=front;i<=rear;i++)
{
        printf("%d\n",q[i]);
}
}
else
{
for(i=front;i<=MAXSIZE-1;i++)
{
        printf("%d\n",q[i]);
}
for(i=0;i<=rear;i++)
{
        printf("%d\n",q[i]);
}
}
}
}
```

11. Give the algorithm for performing polynomial addition using linked list.

The structure of a node representing the polynomial term is:

```
structpolynode
{
intcoefft;
intexp;
structpolynode *link
};
```

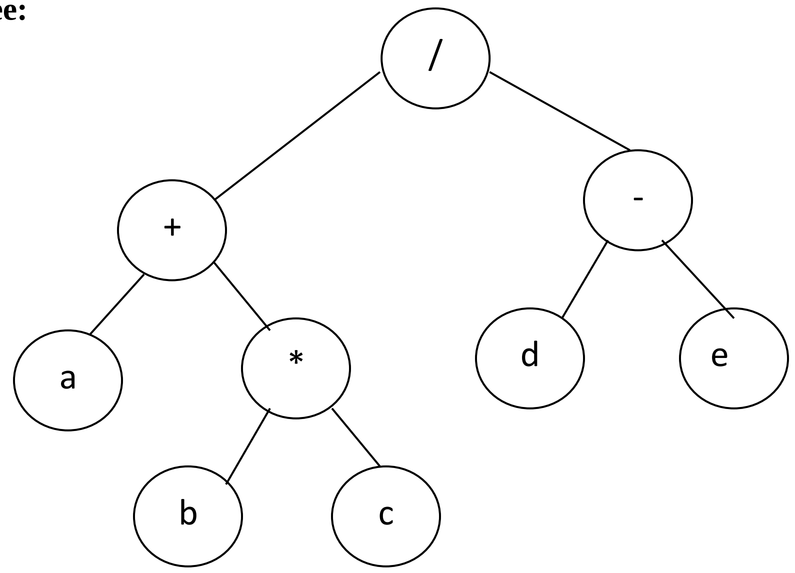
Algorithm for polynomial addition

```
Void PolynomialAddition(polynode *p1, polynode *p2)
{
Polynode *temp1, *temp2, *p3=NULL;
            temp1 = p1;
            temp2 = p2;
            while (temp1 != NULL && temp2 != NULL)
            {
                        temp3 = (polynode *)malloc(sizeof(polynode));
                        temp3->link = NULL;
                        if (temp1->exp == temp2->exp)
                        {
                            temp3->coefft = temp1->coefft + temp2->coefft;
                            temp3->exp = temp1->exp;
                        }
                        else if (temp1->exp> temp2->exp)
                        {
                             temp3->coefft = temp1->coefft;
                             temp3->exp=temp1->exp;
                        }
                        else
                        {
                             temp3->coefft = temp2->coefft;
                             temp3->exp=temp2->exp;
                        }
                        if (p3==NULL)
                            p3=temp3;
                        else
                        {
                             p3->link = temp3;
                             p3 = p3->link;
                        }
            }
            while (temp1 != NULL)
            {
                        temp3 = (polynode *)malloc(sizeof(polynode));
                        temp3->link = NULL;
                        temp3->coefft = temp1->coefft;
                       temp3->exp=temp1->exp;
                        if (p3==NULL)
                            p3=temp3;
                        else
                        {
                       p3->link = temp3;
                       p3 = p3->link;
                        }
            }
            while (temp2 != NULL)
            {
                        temp3 = (polynode *)malloc(sizeof(polynode));
                        temp3->link = NULL;
                        temp3->coefft = temp2->coefft;
                       temp3->exp=temp2->exp;
                        if (p3==NULL)
                            p3=temp3;
                        else
                        {
                       p3->link = temp3;
                       p3 = p3->link;
                        }
            }
    }
```

UNIT-II

1. Find out the inorder, preorder, postorder traversal for the binary tree representing the expression (a+b*c)/(d-e) with the help of procedures

Expression Tree:



Inorder traversal

The inorder traversal of a binary tree is performed as

- traverse the left subtree in inorder.
- Visit the root.
- Traverse the right subtree in inorder.

Recursive Routine for Inorder traversal

```
Void inorder(Tree T)
{
If(T!=NULL)
{
Inorder(T->left);
Printelement(t->element);
Inorder(t->right);
}
}
```

In order traversal for the given expression tree: a + b * c / d - e

Preorder traversal

The preorder traversal of a binary tree is performed as

- Visit the root.
- traverse the left subtree in inorder.
- Traverse the right subtree in inorder.

Recursive Routine for preorder traversal

```
Void preorder(Tree T)
{
If(T!=NULL)
{
Printelement(t->element);
preorder(T->left);
preorder(t->right);
}
}
```

Pre order traversal for the given expression tree:/ + a * b c - d e

Postorder traversal

The postorder traversal of a binary tree is performed as

- traverse the left subtree in inorder.
- Traverse the right subtree in inorder.
- Visit the root.

Recursive Routine for postorder traversal

```
Void postorder(Tree T)
{
If(T!=NULL)
{
postorder(T->left);
postorder(t->right);
Printelement(t->element);
}
}
```

Post order traversal for the given expression tree: a b c * + d e - /

2. A file contains only colons, spaces, newlines, commas and digits in the following frequency. colon-100, space – 605 newline – 100, comma – 705, 0-431, 1-242, 2-176, 2-59, 4-185, 5-250, 6-174,7-199, 8-205, 9-217. Construct the Huffman code. Explain Huffman algorithm

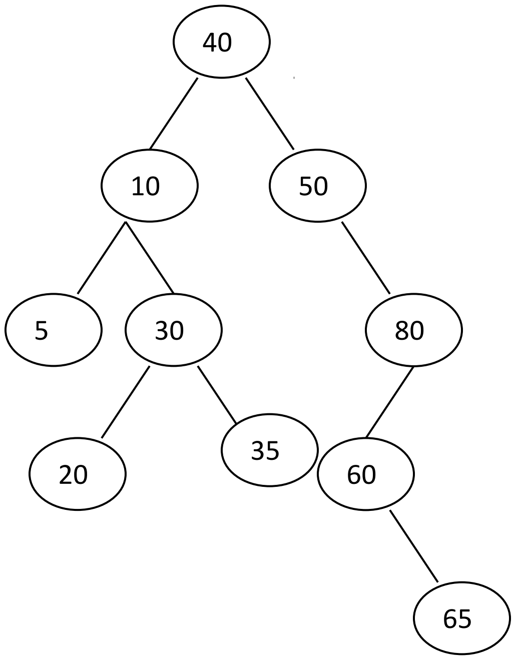
Symbol	**Code**
Colon	01011
Space	00
New line	0100
,	110
0	100
1	1010
2	0111
3	01010
4	11100
5	1011
6	0110

7	11101
8	11110
9	11111

4. What is Binary search tree? Write an algorithm to add a node into a binary search tree.

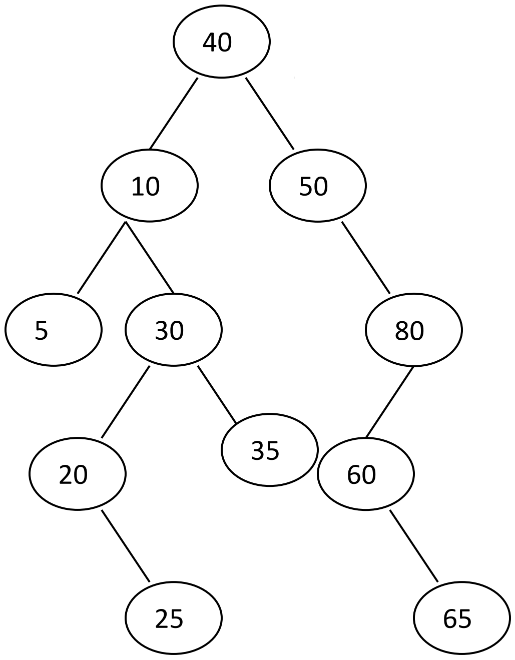
Algorithm for inserting an element into a BST:

```
            Void insert(int x)
            {
            node * prev,*curr;  /*prev is the parent of curr*/
            curr=root;
            prev=NULL;
              /* search for x */


while(curr!=NULL)
            {
            prev=curr;
            if(x==curr->data)
               {
            printf("duplicate value");
            return;
                }
            elseif(x<curr->data)
            curr=curr->lchild;
else
curr=curr->rchild;
}

/*perform insertion*/
curr=(node*)malloc(sizeof(node));
curr->data=x;
curr->lchilde=curr->rchild=NULL;
if(root==NULL)
root=curr;
else if(x<prev->data)
prev->lchild=curr;
else
prev->rchild=curr;
    }
```



X =25 is inserted into the binary tree
25<40
25>10
25<30
25>20

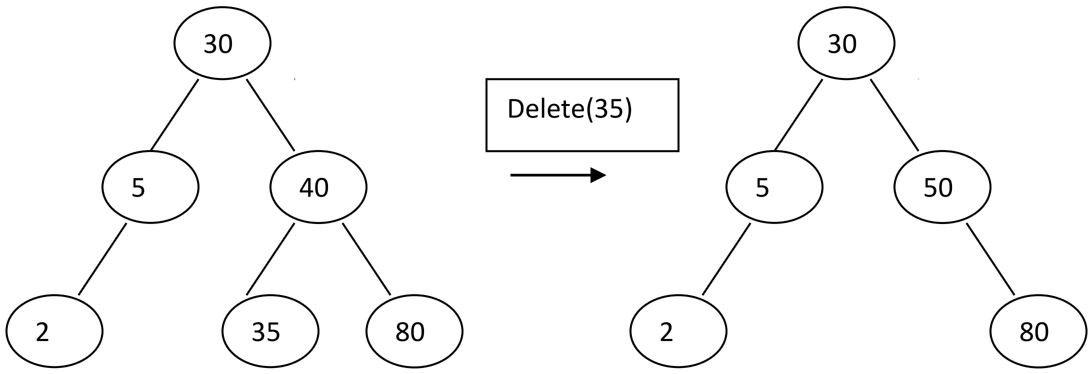
Search is finished and the element is not found. Hence, attach 25 as the right child of 20
Binary tree after insertion:



DELETING A NODE FROM BST:
3 CASES:

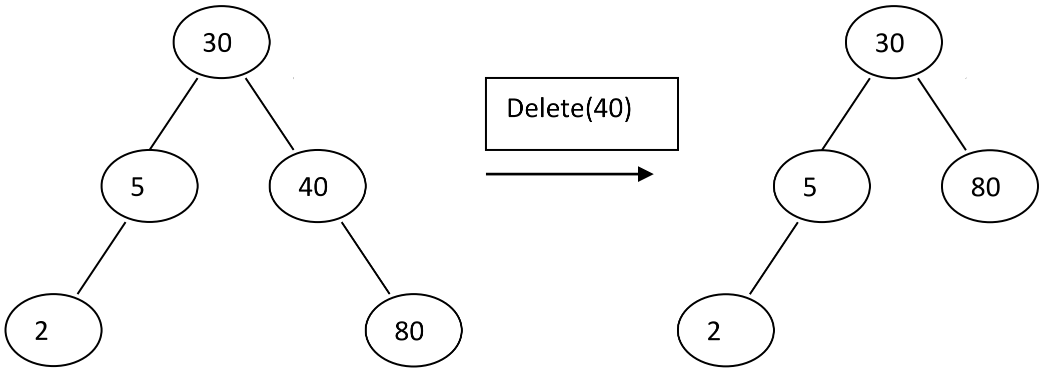
CASE 1: Deleting a leaf node.
CASE 2: Deleting a non leaf node of degree1.
CASE 3: Deleting a non leaf node of degree2.

CASE 1: Deleting a leaf node.



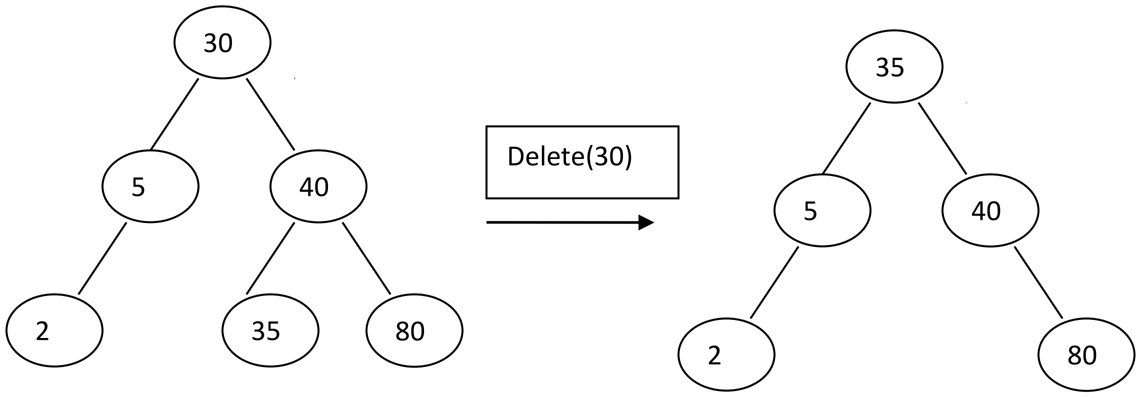
If the node to be deleted is a left child then
Parent->lchild=NULL
If the node to be deleted is a rchild the parent->rchild= NULL
Free (p)

CASE 2 : Non-leaf node with degree 1.

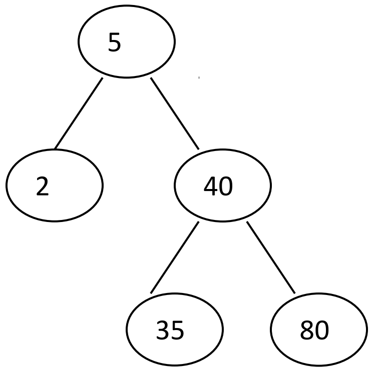


Non leaf node with degree1
The child of the deleted node have to take the position of its parent.

CASE 3: Non leaf node with degree2.



Take either the largest node (it is inorder predecessor) in the left subtree or the smallest node (then it is inorder successor) in the right subtree and replace the node to be deleted with this node and delete the in order predecessor or successor.



Both the largest element in the left subtree & the smallest element in the right subtree can have the degree atmost "one"

5. **Write an algorithm to find a node in a tree. Show the resulting binary search tree if the elements are added into it in the following order:**
   50, 20, 55, 80, 53, 30, 60, 25, 5, …

BINARY SEARCH TREE:-

A binary search tree is a binary tree, it may be empty, if it is not empty then it satisfies the following properties.

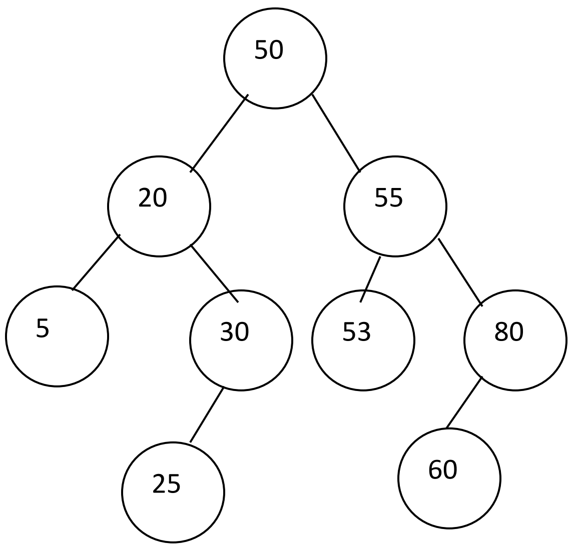
1. Every element has a key and the keys are distinct.
2. The keys in the left subtree are smaller than the key in the root.
3. Keys in the right subtree are larger than the key in the root.
4. Left and right subtrees are also binary search tree.

Algorithm to add a node into a BST:

```
Void insert(int x)
{
        node * prev,*curr;  /*prev is the parent of curr*/
```

```
            curr=root;
            prev=NULL;
              /* search for x */
            while(curr!=NULL)
            {
            prev=curr;
            if(x==curr->data)
               {
            printf("duplicate value");
            return;
                }
            elseif(x<curr->data)
            curr=curr->lchild;
    else
    curr=curr->rchild;
    }
    /*perform insertion*/
    curr=(node*)malloc(sizeof(node));
    curr->data=x;
    curr->lchilde=curr->rchild=NULL;
    if(root==NULL)
    root=curr;
    else if(x<prev->data)
    prev->lchild=curr;
    else
    prev->rchild=curr;
  }
```

Binary Search Tree for the given numbers:



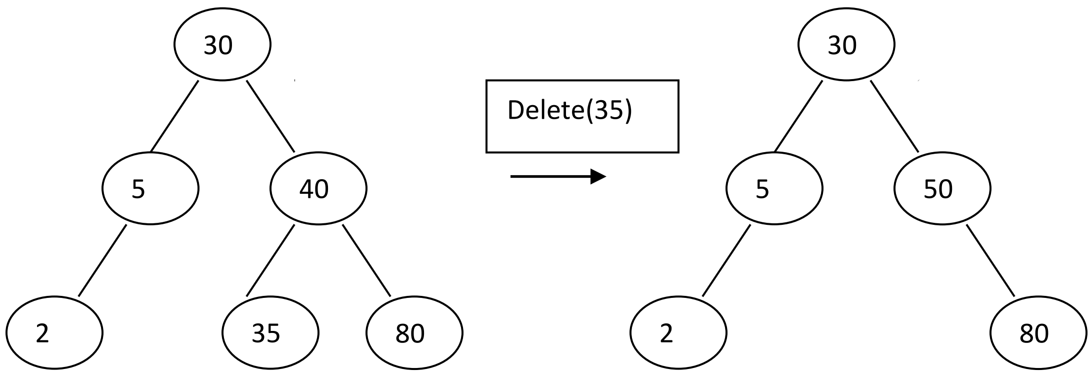
.

6. Write an algorithm to delete a node from a tree (it may contain 0, 1, or 2 children.

DELETING A NODE FROM BST:
3 CASES:

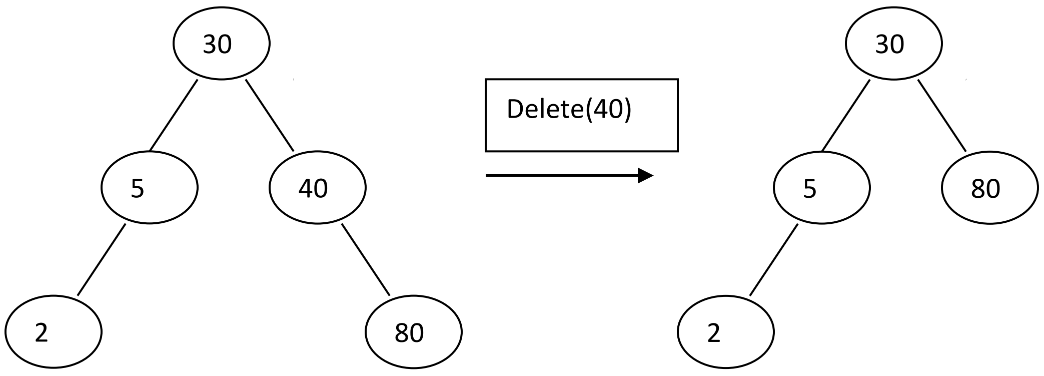
CASE 1: Deleting a leaf node.
CASE 2: Deleting a non leaf node of degree1.
CASE 3: Deleting a non leaf node of degree2.

CASE 1: Deleting a leaf node.



If the node to be deleted is a left child then
Parent->lchild=NULL
If the node to be deleted is a rchild the parent->rchild= NULL
Free (p)

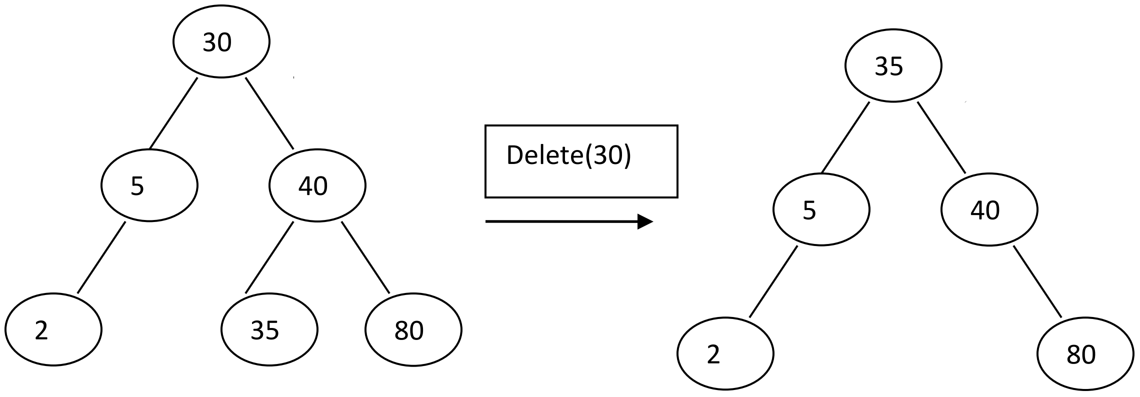
CASE 2 : Non-leaf node with degree 1.



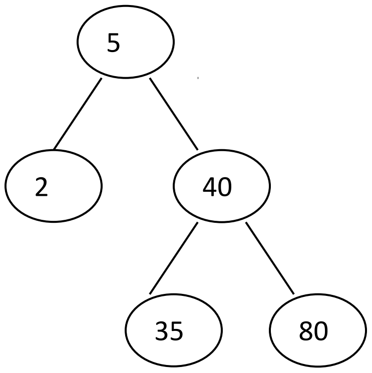
Non leaf node with degree1

The child of the deleted node have to take the position of its parent.

CASE 3: Non leaf node with degree2.

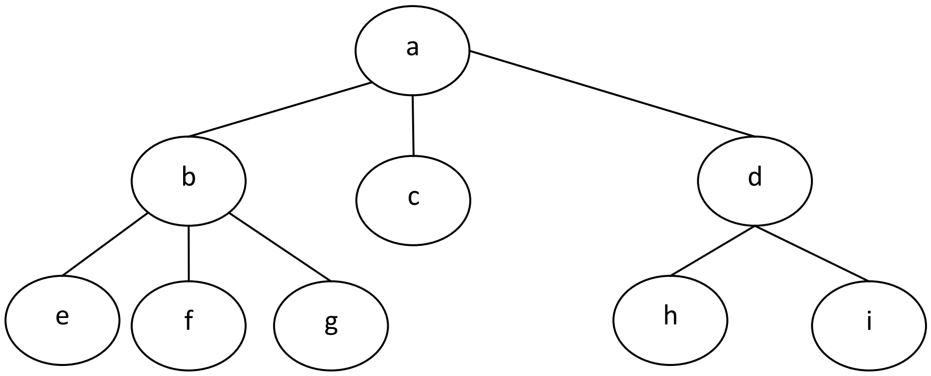


Take either the largest node (it is inorder predecessor) in the left subtree
The smallest node (then it is inorder successor) in the right subtree



Both the largest element in the left subtree & the smallest element in the right subtree can have the degree atmost "one".

7. Explain the steps involved in converting the general tree to a binary tree. Convert the following general tree to a binary tree.



Conversion of General Tree to Binary Tree:

The left most child becomes the left child

Other siblings become the right child of left most child.

Algorithm:

Step 1: Create a head node for the binary tree and push the address and level number onto the stack.

Step 2: Repeat through step 6 while there is data.

Step 3: Input the current node description in preorder (address& level)

Step 4: Create a treenode and initialize its contents.

Step 5: If the level number of the current node > the level number of the node at the top of the stack then

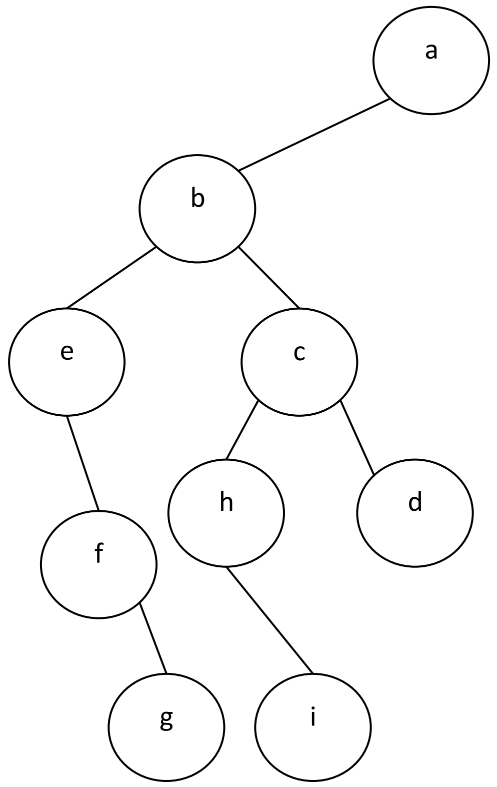
Connect the current node as the left child of the node at the top of the stack

Else

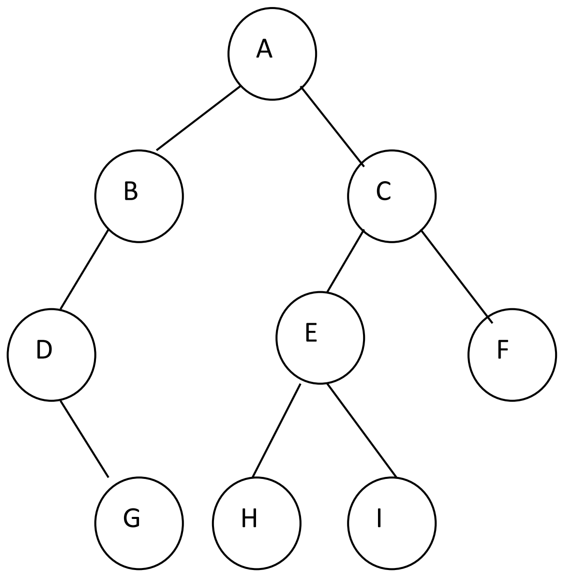
Remove all the nodes from the stack whose level number is greater the level number of the current node & connect the current node as a right child of the node at the top of the stack.

Step 6: Push the current node description onto the stack.

Binary tree for the given general tree:



8. Construct a binary tree given the preorder and in order sequences as below
preorder: A B D G C E H I F, Inorder : D G B A H E I C F



9. Prove "For any non-empty binary tree T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0 = n_2+1$"

Proof

Let n be the total no of nodes in the binary tree let n, be the no of nodes of degree 1

$n = n_0 + n_1 + n_2$-------- A

All the nodes except the root node has a branch coming into it. Let B be the no of branches in binary tree

$n = B+1$------ 1

(deg=0)

Nodes of degree-1 will have 1 branch

Nodes of degree – 2 will have 2 branch

$B = 0.n_0 + 1.n_1 + 2.n_2$

$B = n_1 + 2n_2$-- 2

Substitute eq (2) in (1)

$n = n_1+2n_2+1$---B

equate A and B

$n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$

$n_0 = 2n_2 - n_2+1$

$\boxed{n_0 = n_2 +1}$

Hence proved.

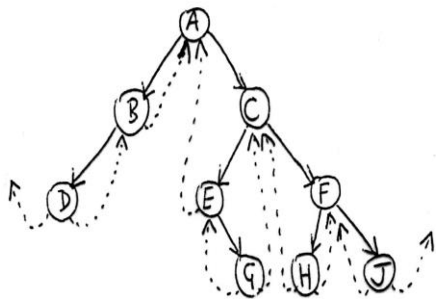
10.What do you mean by a threaded binary tree? Write the algorithm for in order traversal of a threaded binary tree. Trace the algorithm with an example.

In a binary tree, all the leaf nodes are having the left child and right child fields to be NULL. Here more memory space is wasted to store the NULL values. These NULL pointers can be utilized to store useful information. The NULL left child is used to point the in order predecessor and the NULL right child is used to store the in order successor. This is called as in order threaded binary tree.

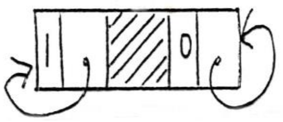
Structure of a node:

LTHREAD	**LLINK**	**DATA**	**RLINK**	**RTHREAD**

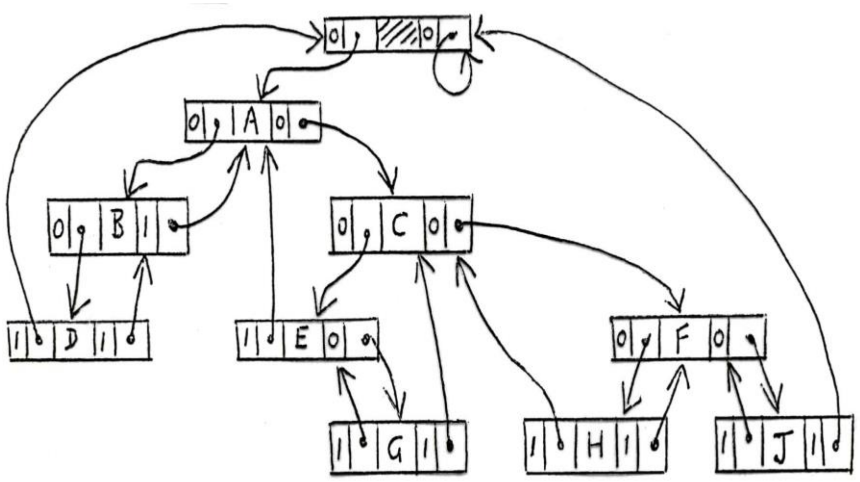
if LTHREAD= 0, LLINK points to the left child;
if LTHREAD = 1, LLINK points to the in-order predecessor;
if RTHREAD = 0, RLINK points to the right child;
if RTHREAD= 1, RLINK points to the in-order successor.



HEAD



Conventionally, HEAD.RLINK = HEAD and HEAD.RTAG = 0 for any threaded binary tree. The tree shown earlier would therefore be represented as:



Algorithm

Step-1: For the current node check whether it has a left child which is not there in the visited list. If it has then go to step-2 or else step-3.

Step-2: Put that left child in the list of visited nodes and make it your current node in consideration. Go to step-6.

Step-3: For the current node check whether it has a right child. If it has then go to step-4 else go to step-5

Step-4: Make that right child as your current node in consideration. Go to step-6.

Step-5: Check for the threaded node and if its there make it your current node.

Step-6: Go to step-1 if all the nodes are not over otherwise quit

In order Traversal for the above threaded binary tree: D B A E G C H F J

11. What is the representation of binary tree in memory? Explain in detail. / Explain the B-tree with insertion and deletion operations.

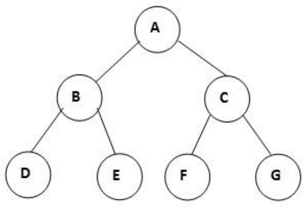
Representation of Binary tree in memory:

1. Array Representation
2. Linked List Representation

Array Representation:

- The root node is stored at location 0.
- Left child of the node at location i is stored at location 2i+1
- Right child of the node at location i is stored at location 2i+2

If the child is in ith location, its parent will be in (i-1)/2 thlocation.



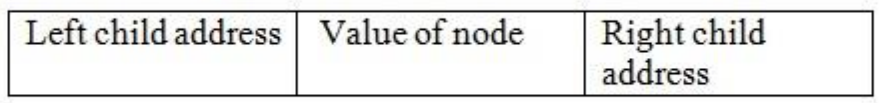
A	B	C	D	E	F	G
0	1	2	3	4	5	6

Linked List Representation:

Structure of a node:

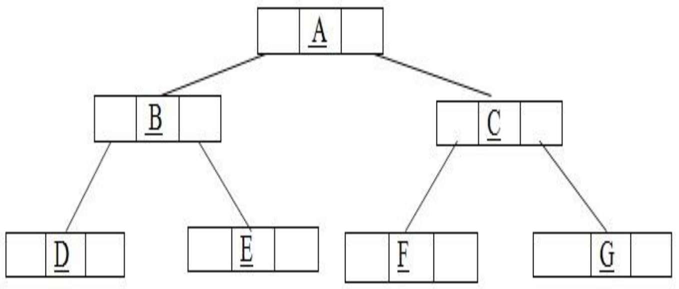
```
Structtreenode
{
        Structtreenode *leftchild;
        Int data;
        Structtreenode *rightchild;
};
```

Example:-



Left child address	Value of node	Right child address

Node



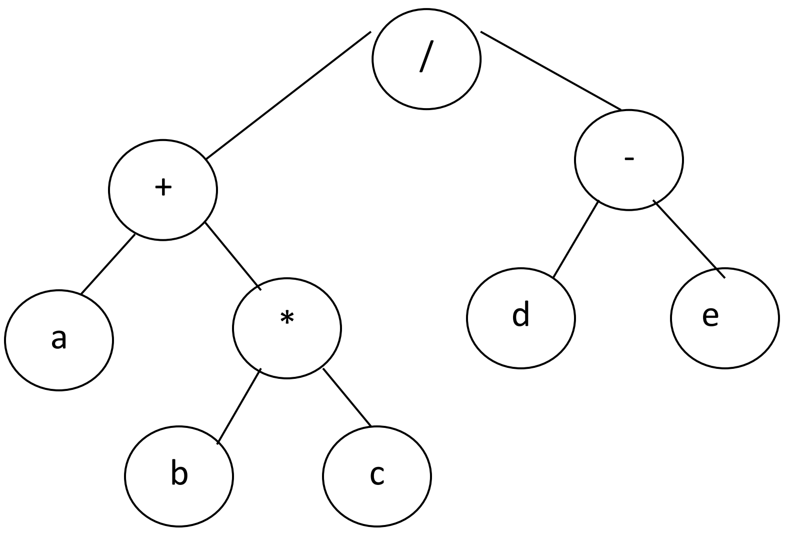
12.Define expression tree. How to construct an expression tree for the post fix expression? / Write steps involved in constructing expression tree.

Expression tree:

An expression tree is built up from the simple operands and operators of an(arithmetic or logical) expression by placing the simple operands as the leaves of a binary tree and the operators as the interior nodes.

Example:
(a+b*c)/(d-e)

Expression Tree:



Inorder traversal
The inorder traversal of a binary tree is performed as

- traverse the left subtree in inorder.
- Visit the root.
- Traverse the right subtree in inorder.

Recursive Routine for Inorder traversal

```
Void inorder(Tree T)
{
If(T!=NULL)
{
Inorder(T->left);
Printelement(t->element);
Inorder(t->right);
}
}
```

In order traversal for the given expression tree: a + b * c / d - e

Preorder traversal

The preorder traversal of a binary tree is performed as

- Visit the root.
- traverse the left subtree in inorder.
- Traverse the right subtree in inorder.

Recursive Routine for preorder traversal

```
Void preorder(Tree T)
{
If(T!=NULL)
{
Printelement(t->element);
preorder(T->left);
preorder(t->right);
}
}
```

Pre order traversal for the given expression tree:/ + a * b c - d e

Postorder traversal

The postorder traversal of a binary tree is performed as

- traverse the left subtree in inorder.
- Traverse the right subtree in inorder.
- Visit the root.

Recursive Routine for postorder traversal

```
Void postorder(Tree T)
{
If(T!=NULL)
{
postorder(T->left);
postorder(t->right);
Printelement(t->element);
}
}
```

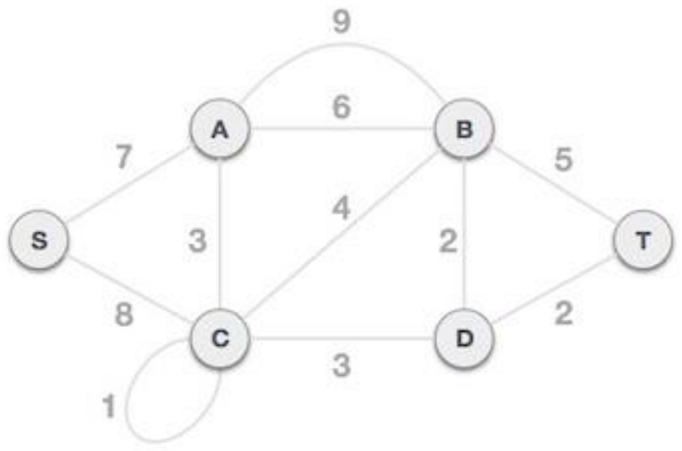
Post order traversal for the given expression tree: a b c * + d e - /

UNIT-III

1. **Construct a minimum spanning tree using Kruskal's algorithm with your own example**

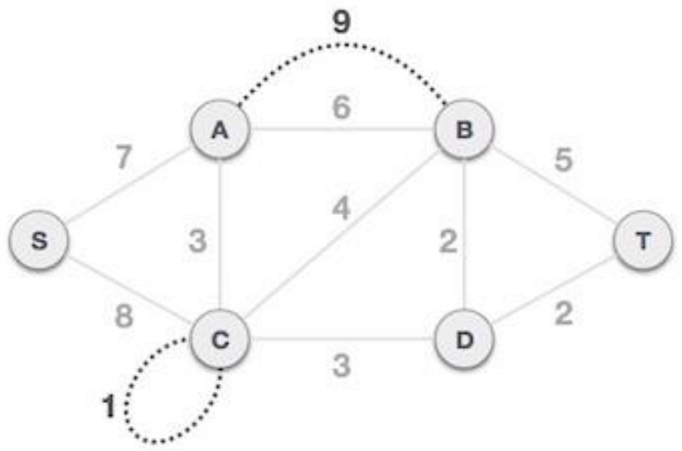
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

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

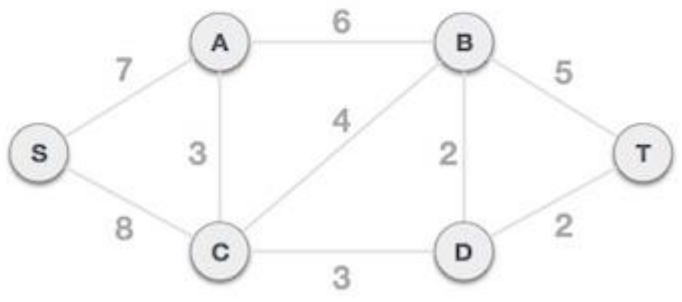


Step 1 - Remove all loops and Parallel Edges

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.



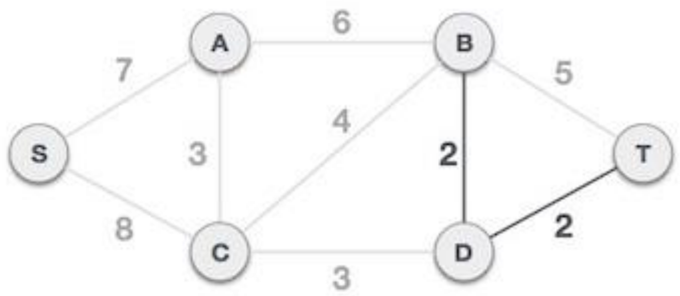
Step 2 - Arrange all edges in their increasing order of weight

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

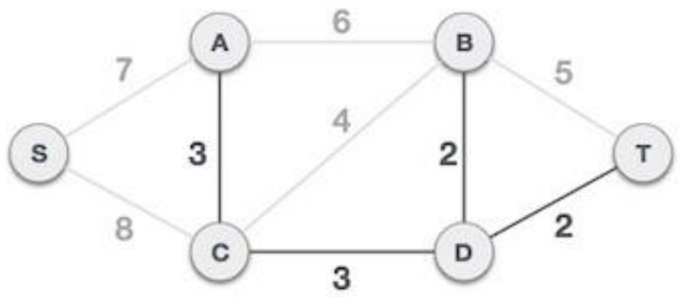
Step 3 - Add the edge which has the least weightage

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.

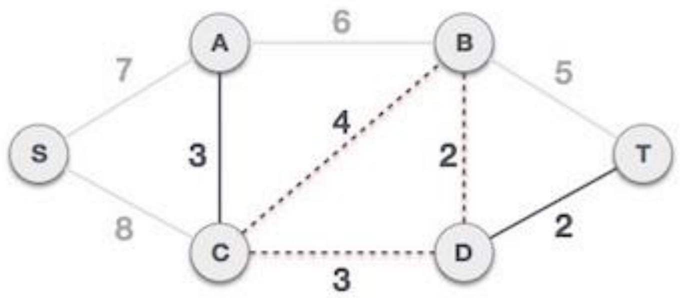


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

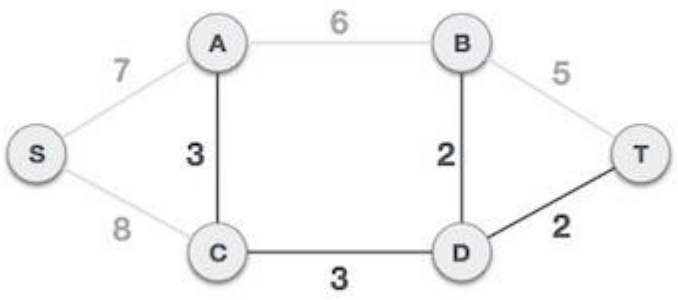
Next cost is 3, and associated edges are A,C and C,D. We add them again −



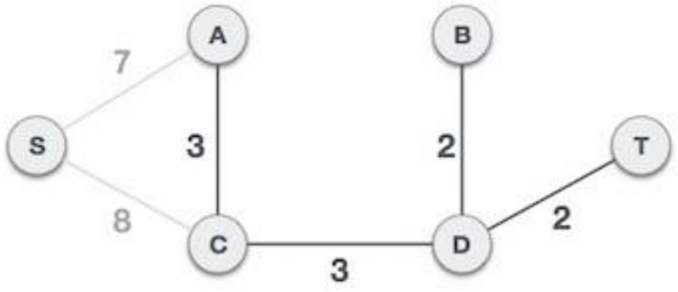
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



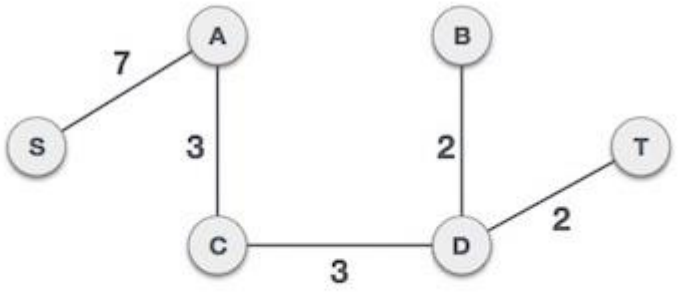
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



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

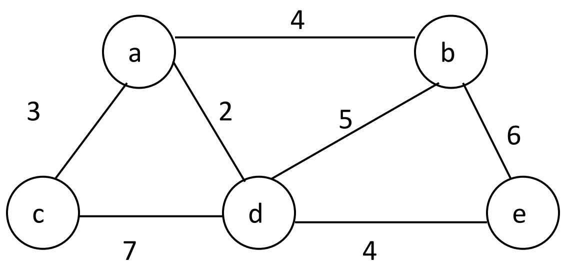


Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



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

2. How will find the shortest path between two given vertices using Dijikstra's algorithm? Explain the pseudo code with an example



Dijkstra's algorithm finds the shortest path from a source vertex(v) to all the remaining vertices.

Steps:

1. **1.** Initialize s[i] =false &dist[i] = length[v][i] for all i=0 to n-1.
2. **2.** Assign s[v] = true &dist[v] = 0;
3. **3.** Choose a vertex u with minimum dist& s[u] = false
4. **4.** Put s[u] = true.
5. **5.** Modify dist[w] for all vertices with s[w]= false

   Dist[w] = min { dist[w], dist[u] + length[u][w]}
6. **6.** repeat the steps 3 to 5 until the shortest path is found for all the remaining vertices.

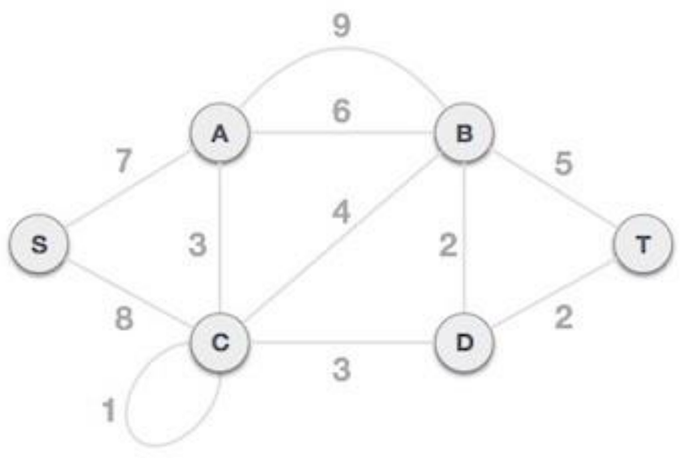
Ans:

a-b = 4
a-c = 3
a-d = 2
a-e = 6

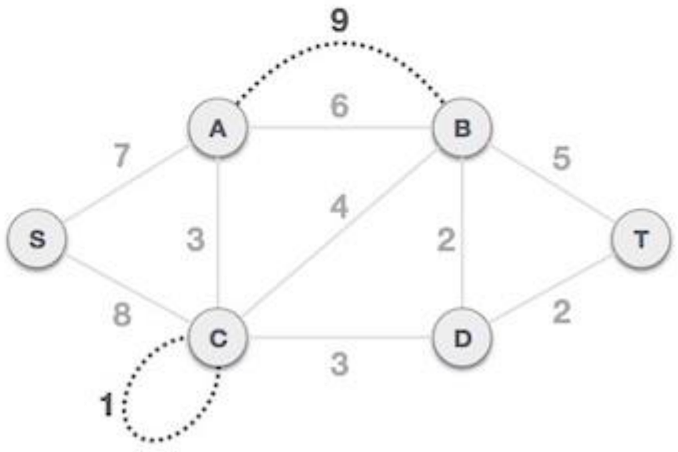
3. Discuss about the algorithm and pseudocode to find minimum spanning tree using Prim's algorithm.

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.

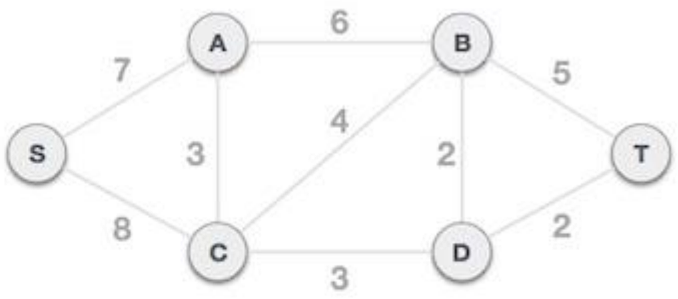
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



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.

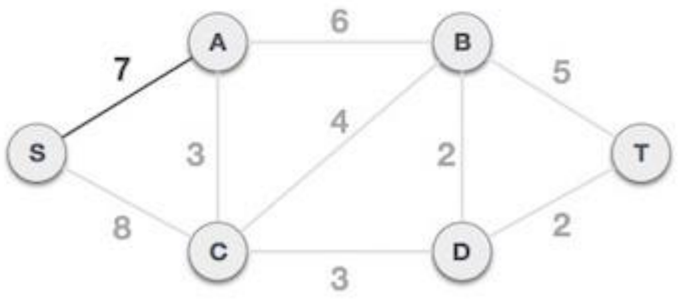


Step 2 - Choose any arbitrary node as root node

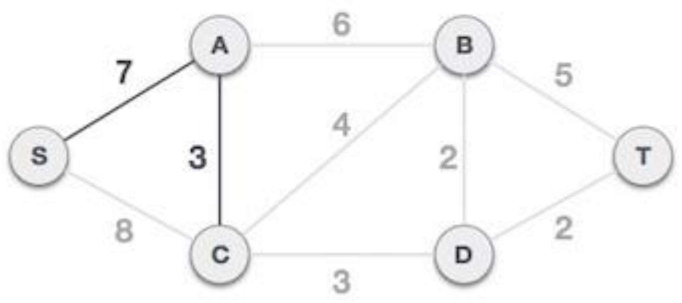
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then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

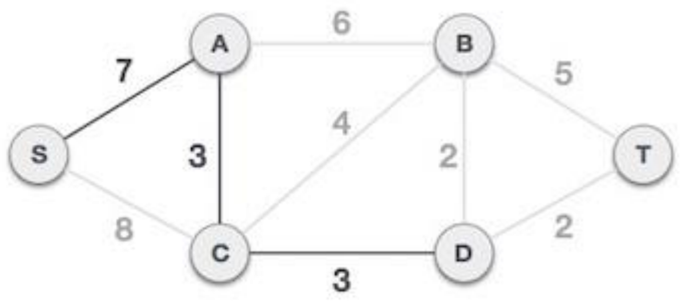
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



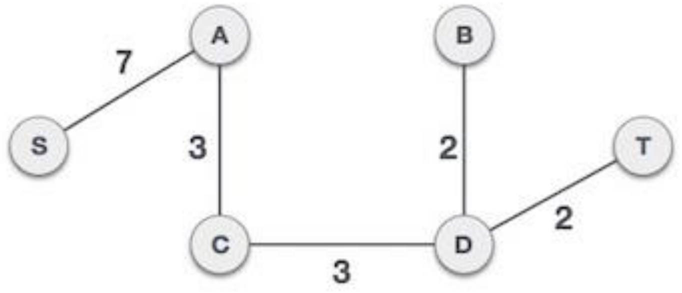
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



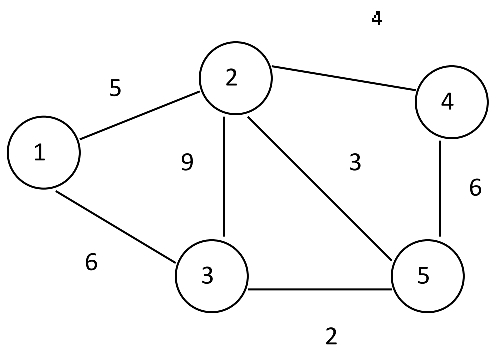
After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



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.



4. **Write Floyd's algorithm for the all-pairs shortest path problem and explain with an example**



```
Algorithm floyd(w[1..n, 1..n])
{
D(0) = A

For k = 1 to n do
For i = 1 to n do
For j = 1 to n do
Dk[I,j] = min{Dk-1[I,j] or Dk-1[I,j] and Dk-1[k,j]
Return D(n)
}
```

Ans:

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 5 & 6 & 9 & 8 \\ 2 & 5 & 0 & 5 & 4 & 3 \\ 3 & 6 & 5 & 0 & 8 & 2 \\ 4 & 9 & 4 & 8 & 0 & 6 \\ 5 & 8 & 3 & 2 & 6 & 0 \end{array}$$

5. Explain in detail about Bellman-Ford algorithm with suitable example

MST solves the problem of finding a minimum total weight subset of edges that spans all the vertices. Another common graph problem is to find the shortest paths to all reachable vertices from a given source. We have already seen how to solve this problem in the case where all the edges have the *same* weight (in which case the shortest path is simply the minimum *number* of edges) using BFS. Now we will examine two algorithms for finding *single source shortest paths* for directed graphs when the edges have *different* weights - Bellman-Ford and Dijkstra's algorithms. Several related problems are:

- Single destination shortest path - find the transpose graph (i.e. reverse the edge directions) and use single source shortest path
- Single pair shortest path (i.e. a specific destination) - asymptotically this problem can be solved no faster than simply using single source shortest path algorithms to all the vertices
- All pair shortest paths - one technique is to use single source shortest path for each vertex, but later we will see a more efficient algorithm

Single Source Shortest Path

Problem

Given a directed graph *G(V,E)* with *weighted edgesw(u,v)*, define the *path weight* of a path *p* as

$$w(p)=\sum_{i=1}^{k} w(v_{i-1},v_i)$$

For a given source vertex *s*, find the *minimum weight paths* to every vertex reachable from *s* denoted

$$\delta(s,v)=\begin{cases}\min\{w(p) \quad s\rightarrow v\} \\ \infty \text{ otherwise}\end{cases}$$

The final solution will satisfy certain caveats:

- The graph cannot contain any *negative weight cycles* (otherwise there would be no minimum path since we could simply continue to follow the negative weight cycle producing a path weight of -∞).
- The solution cannot have any *positive weight cycles* (since the cycle could simply be removed giving a lower weight path).
- The solution can be assumed to have no zero weight cycles (since they would not affect the minimum value).

Therefore given these caveats, we know the shortest paths must be *acyclic* (with ≤ |*V*| distinct vertices) ⇒ ≤ |*V*| - 1 edges in each path.

Generic Algorithm

The single source shortest path algorithms use the same notation as BFS (see lecture 17) with predecessor *π* and distance *d* fields for each vertex. The optimal solution will have *v.d* = δ(*s*,*v*) for all *v*∈*V*.

The solutions utilize the concept of *edge relaxation* which is a test to determine whether going through edge (*u*,*v*) reduces the distance to *v* and if so update *v.π* and *v.d*. This is accomplished using the condition

$$\text{if } v.d > u.d + w(u,v)$$

$$v.d = u.d + w(u,v)$$

$$v.\pi = u$$

Bellman-Ford Algorithm

The *Bellman-Ford algorithm* uses relaxation to find single source shortest paths on directed graphs that may contain *negative weight edges*. The algorithm will also detect if there are any *negative weight cycles* (such that there is no solution).

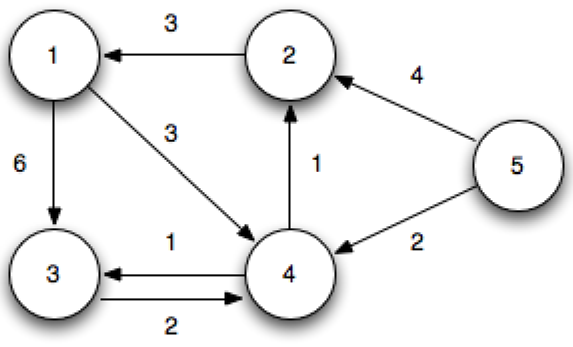
```
BELLMAN-FORD(G,w,s)
 INITIALIZE-SINGLE-SOURCE(G,s)
for i = 1 to |G.V|-1
for each edge (u,v) ∈ G.E
RELAX(u,v,w)
for each edge (u,v) ∈ G.E     if v.d>u.d + w(u,
return FALSE
return TRUE

INITIALIZE-SINGLE-SOURCE(G,s)
for each vertex v ∈ G.V
v.d = ∞
v.pi = NIL
s.d = 0
```

RELAX(u,v,w) if v.d>u.d + w(u,v)
v.d = u.d + w(u,v)
v.pi = u
Basically the algorithm works as follows:

1. Initialize *d*'s, π's, and set *s.d* = 0 ⇒ O(*V*)
2. Loop |*V*|-1 times through all edges checking the relaxation condition to compute minimum distances ⇒ (|*V*|-1) O(*E*) = O(*VE*)
3. Loop through all edges checking for negative weight cycles which occurs if any of the relaxation conditions fail ⇒ O(*E*)

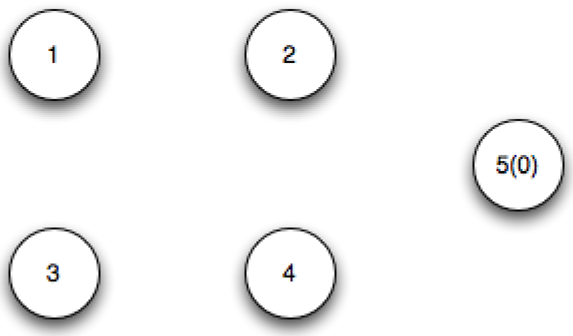
The run time of the Bellman-Ford algorithm is O(*V* + *VE* + *E*) = O(*VE*).
Note that if the graph is a DAG (and thus is known to not have any cycles), we can make Bellman-Ford more efficient by first *topologically sortingG* (O(*V*+*E*)), performing the same initialization (O(*V*)), and then simply looping through each vertex *uin topological order* relaxing only the edges in Adj[*u*] (O(*E*)). This method only takes O(*V* + *E*) time. This procedure (with a few slight modifications) is useful for finding *critical paths* for PERT charts.

6. Given the following directed graph



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

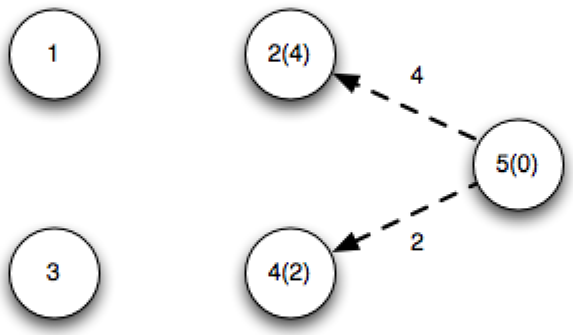
Using vertex 5 as the source (setting its distance to 0), we initialize all the other distances to ∞.



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	∞	∞	∞	0
π	/	/	/	/	/

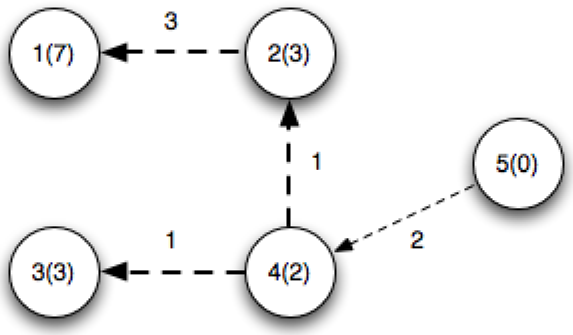
Iteration 1: Edges (u_5,u_2) and (u_5,u_4) relax updating the distances to 2 and 4



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	**4**	∞	**2**	0
π	/	**5**	/	**5**	/

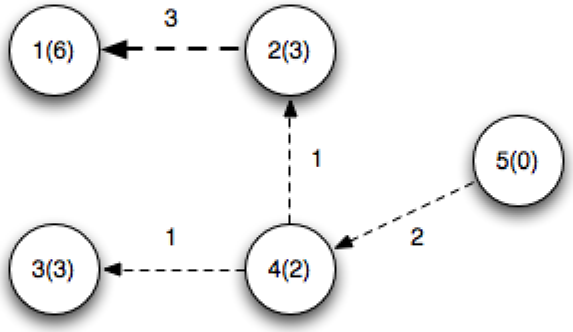
Iteration 2: Edges (u_2,u_1), (u_4,u_2) and (u_4,u_3) relax updating the distances to 1, 2, and 4 respectively. Note edge (u_4,u_2) finds a shorter path to vertex 2 by going through vertex 4



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**7**	**3**	**3**	2	0
π	**2**	**4**	**4**	5	/

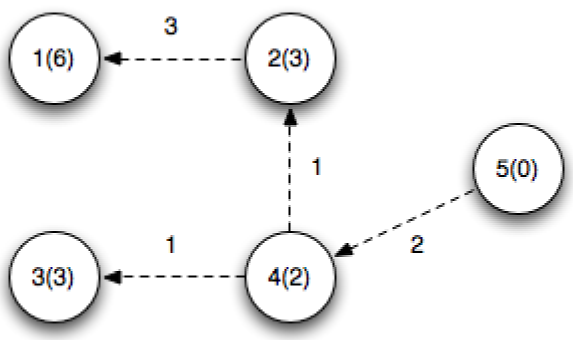
Iteration 3: Edge (u_2,u_1) relaxes (since a shorter path to vertex 2 was found in the previous iteration) updating the distance to 1



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**6**	3	3	2	0
π	**2**	4	4	5	/

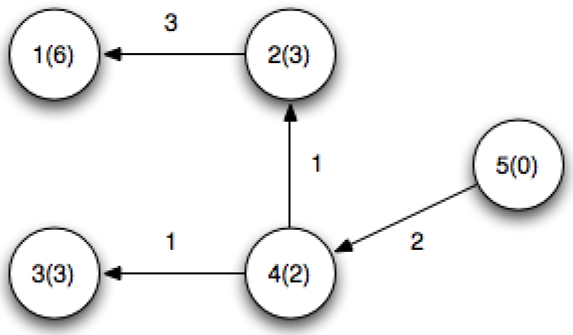
Iteration 4: No edges relax



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

The final shortest paths from vertex 5 with corresponding distances is



	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Negative cycle checks: We now check the relaxation condition one additional time for each edge. If any of the checks pass then there exists a negative weight cycle in the graph.

$v_3.d>u_1.d + w(1,3) \Rightarrow 4 \ngtr 6 + 6 = 12$ ✓

$v_4.d>u_1.d + w(1,4) \Rightarrow 2 \ngtr 6 + 3 = 9$ ✓

$v_1.d>u_2.d + w(2,1) \Rightarrow 6 \ngtr 3 + 3 = 6$ ✓

$v_4.d>u_3.d + w(3,4) \Rightarrow 2 \ngtr 3 + 2 = 5$ ✓

$v_2.d>u_4.d + w(4,2) \Rightarrow 3 \ngtr 2 + 1 = 3$ ✓

$v_3.d>u_4.d + w(4,3) \Rightarrow 3 \ngtr 2 + 1 = 3$ ✓

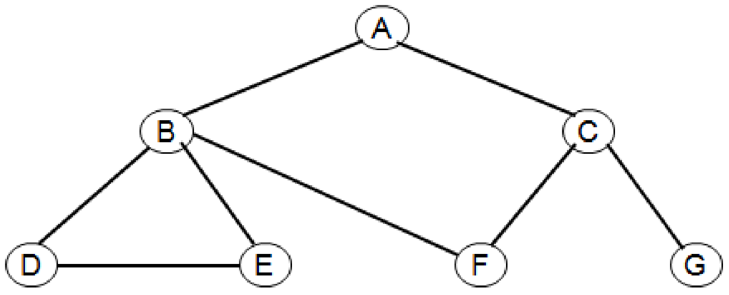
$v_2.d>u_5.d + w(5,2) \Rightarrow 3 \ngtr 0 + 4 = 4$ ✓

$v_4.d>u_5.d + w(5,4) \Rightarrow 2 \ngtr 0 + 2 = 2$ ✓

Note that for the edges *on the shortest paths* the relaxation criteria gives equalities. Additionally, the path to any reachable vertex can be found by starting at the vertex and following the π's back to the source. For example, starting at vertex 1, $u_1.\pi = 2$, $u_2.\pi = 4$, $u_4.\pi = 5$ ⇒ the shortest path to vertex 1 is {5,4,2,1}

7. Describe in detail about depth first and breadth first traversals with appropriate example

Breadth First Search



This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used here. If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. **"A B C D E F G".** The BFS visits the nodes level by level, so it will start with level A which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are D,E,F and F.

Breadth First Traversal:

1. Visit vertex v.
2. Visit all the unvisited vertices that are adjacent to v.
3. Unvisited vertices that are adjacent to the newly visited vertices are visited.

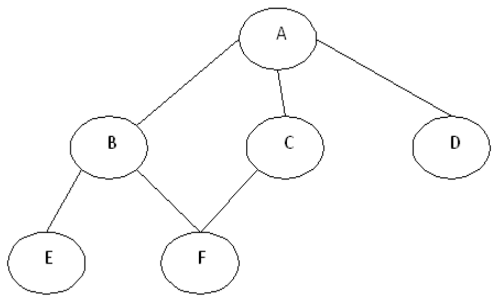
Algorithmic Steps

Step 1: Push the root node in the Queue.
Step 2: Loop until the queue is empty.
Step 3: Remove the node from the Queue.
Step 4: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

Algorithm:

```
bfs ( )
{
mark v visited;
enqueue (v);
while ( not is_empty (Q) )
{
x = front (Q);
dequeue (Q);
for each y adjacent to x if y unvisited {
mark y visited;
enqueue (y);
insert ( (x, y) in T );
}
}
}
```

Depth First Traversal



The aim of DFS traversal is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search. If we do the depth first traversal of the above graph and print the visited node, it will be **"A B E F C D".** DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

Depth First Traversal.

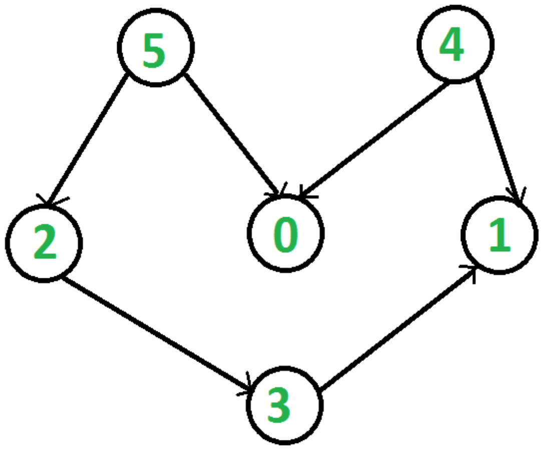
1. Visit the vertex v.
2. Visit an unvisited vertex w that is adjacent to v.
3. Initiate depth first search from w.

8. Explain in detail about topological sorting with an example

Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



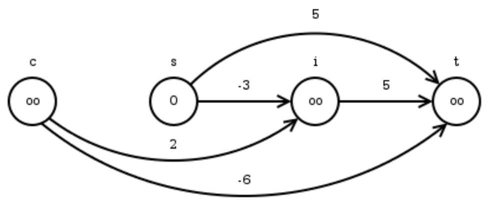
Topological Sorting vs

In DFS, we prir for its adjacent vertices. In topological sorting, we vertices. For example, in the given graph, the vertex t unlike DFS, the vertex '4' should also be printed is different from DFS. For example, a DFS of the sl pological sorting

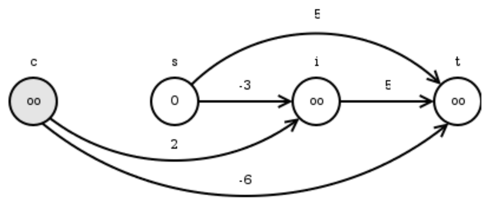
9. Explain in detail about the Single-Source Shortest Paths in DAGs

By relaxing the edges in a DAG according to their topological sort of its vertices. We can achieve Θ(n+m) time complexity.

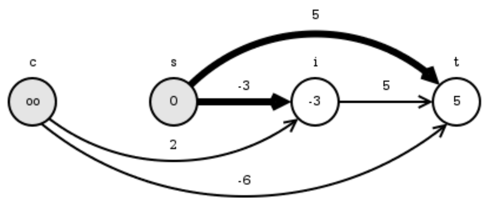
```
DAG-SHORTEST (G,w,s)
    Topologically sort the vertices of G
      INITIALIZE (G,s)
for each vertex u taken in topologically sorted (increasing) order
do for  v∈Adj [u]
do  RELAX (u,v,w)
```



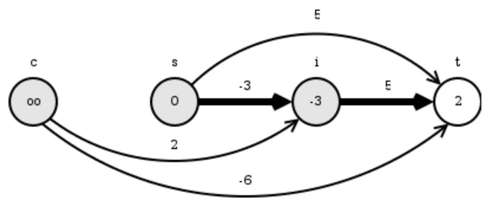
(a) Initialise



(b) Consider c; relax i, and t

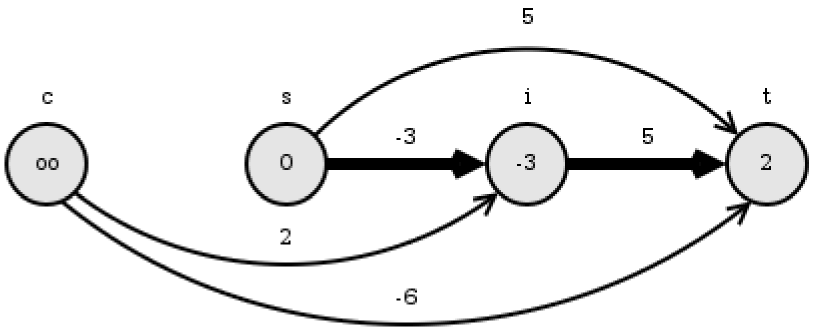


(c) Consider s; relax i, and t



(d) Consider i; relax t

5



(e) Consider t

UNIT-IV

1. **Discuss briefly the sequence of steps in designing and analyzing an algorithm.**

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the **algorithm is independent from any programming languages**.

Algorithm Design

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Problem Development Steps

The following steps are involved in solving computational problems.

- Problem definition
- Development of a model
- Specification of an Algorithm
- Designing an Algorithm
- Checking the correctness of an Algorithm
- Analysis of an Algorithm
- Implementation of an Algorithm
- Program testing
- Documentation

Characteristics of Algorithms

The main characteristics of algorithms are as follows −

- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term **"analysis of algorithms"** was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

The Need for Analysis

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.
- **Best-case** − The minimum number of steps taken on any instance of size **a**.
- **Average case** − An average number of steps taken on any instance of size **a**.
- **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa. In this context, if we compare **bubble sort** and **merge sort**. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited

2. Explain in detail about asymptotic notations used in algorithm analysis

In designing of Algorithm, complexity analysis of an algorithm is an essential aspect. Mainly, algorithmic complexity is concerned about its performance, how fast or slow it works.The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.

Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

Time Complexity

It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "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

It's a function describing the amount of memory an algorithm takes in terms of the size 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.

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

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by **T(n)**, where **n** is the input size.
Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta
- **o** − Little Oh
- **ω** − Little omega

O: Asymptotic Upper Bound
'O' (Big Oh) is the most commonly used notation. A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that $-f(n) \leqslant c.g(n)$ for $n>n0$ in all case
Hence, function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.

Example
Let us consider a given function, $f(n)=4.n3+10.n2+5.n+1$, Considering $g(n)=n^3$
$f(n) \leqslant 5.g(n)$ for all the values of $n>2$
Hence, the complexity of ***f(n)*** can be represented as $O(g(n))$, i.e. $O(n^3)$
Ω: Asymptotic Lower Bound

We say that $f(n)=\Omega(g(n))$ when there exists constant **c** that $f(n) \geqslant c.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f***; after a certain value of ***n, f*** will never go below ***g***.

Example

Let us consider a given function, $f(n)=4.n3+10.n2+5.n+1$
.Considering $g(n)=n^3$, $f(n) \geqslant 4.g(n)$ for all the values of $n>0$
Hence, the complexity of ***f(n)*** can be represented as $\Omega(g(n))$, i.e. $\Omega(n^3)$

θ: Asymptotic Tight Bound **We say that $f(n)=\theta(g(n))$**
When there exist constants $\mathbf{c_1}$ and $\mathbf{c_2}$ that $c1.g(n) \leqslant f(n) \leqslant c2.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer. This means function **g** is a tight bound for function **f**.

Example
Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$
Considering $g(n)=n^3$, $4.g(n) \leqslant f(n) \leqslant 5.g(n)$for all the large values of **n**.
Hence, the complexity of ***f(n)*** can be represented as $\theta(g(n))$, i.e. $\theta(n3)$

.O - Notation
The asymptotic upper bound provided by **O-notation** may or may not be asymptotically tight. The bound $2.n2=O(n2)$is asymptotically tight, but the bound $2.n=O(n2)$is not.
We use **o-notation** to denote an upper bound that is not asymptotically tight.We formally define ***o(g(n))*** (little-oh of g of n) as the set ***f(n) = o(g(n))*** for any positive constant $c>0$and there exists a value $n0>0$, such that $0 \leqslant f(n) \leqslant c.g(n)$
Intuitively, in the **o-notation**, the function ***f(n)*** becomes insignificant relative to ***g(n)*** as **n** approaches infinity; that is,
$\lim n \to \infty (f(n)g(n))=0$

Example
Let us consider the same function, $f(n)=4.n3+10.n2+5.n+1$
Considering $g(n)=n4$ $\lim n \to \infty (4.n3+10.n2+5.n+1n4)=0$
Hence, the complexity of ***f(n)*** can be represented as $o(g(n))$, i.e. $o(n4)$

ω – Notation

We use **ω-notation** to denote a lower bound that is not asymptotically tight. Formally, however, we define ***ω(g(n))*** (little-omega of g of n) as the set ***f(n) = ω(g(n))*** for any positive constant ***C > 0*** and there exists a value $n0>0$
, such that $0\leqslant c.g(n)<f(n)$,For example, $n22=\omega(n)$
, but $n22\neq\omega(n2)$. The relation $f(n)=\omega(g(n))$implies that the following limit exists
lim$n\to\infty(f(n)g(n))=\infty$
That is, ***f(n)*** becomes arbitrarily large relative to ***g(n)*** as **n** approaches infinity.

Example

Let us consider same function, $f(n)=4.n3+10.n2+5.n+1$
Considering $g(n)=n2$
lim$n\to\infty(4.n3+10.n2+5.n+1n2)=\infty$
Hence, the complexity of ***f(n)*** can be represented as $o(g(n))$, i.e. $\omega(n2)$
Apriori and Apostiari Analysis
Apriori analysis means, analysis is performed prior to running it on a specific system.This analysis is a stage where a function is defined using some theoretical model. Hence, we determine the time and space complexity of an algorithm by just looking at the algorithm rather than running it on a particular system with a different memory, processor, and compiler.
Apostiari analysis of an algorithm means we perform analysis of an algorithm only after running it on a system. It directly depends on the system and changes from system to system.
In an industry, we cannot perform Apostiari analysis as the software is generally made for an anonymous user, which runs it on a system different from those present in the industry.
In Apriori, it is the reason that we use asymptotic notations to determine time and space complexity as they change from computer to computer; however, asymptotically they are the same

3. Explain in detail about divide and conquer approach

Many algorithms are recursive in nature to solve a given problem recursively dealing with sub-problems.
In **divide and conquer approach**, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.
Generally, divide-and-conquer algorithms have three parts −

- **Divide the problem** into a number of sub-problems that are smaller instances of the same problem.
- **Conquer the sub-problems** by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine the solutions** to the sub-problems into the solution for the original problem.

Pros and cons of Divide and Conquer Approach
Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.
In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

Application of Divide and Conquer Approach

Following are some problems, which are solved using divide and conquer approach.

- Finding the maximum and minimum of a sequence of numbers
- Strassen's matrix multiplication
- Merge sort
- Binary search

4. Describe in detail about merge sort with an example

Problem Statement

The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sub-lists.

Solution

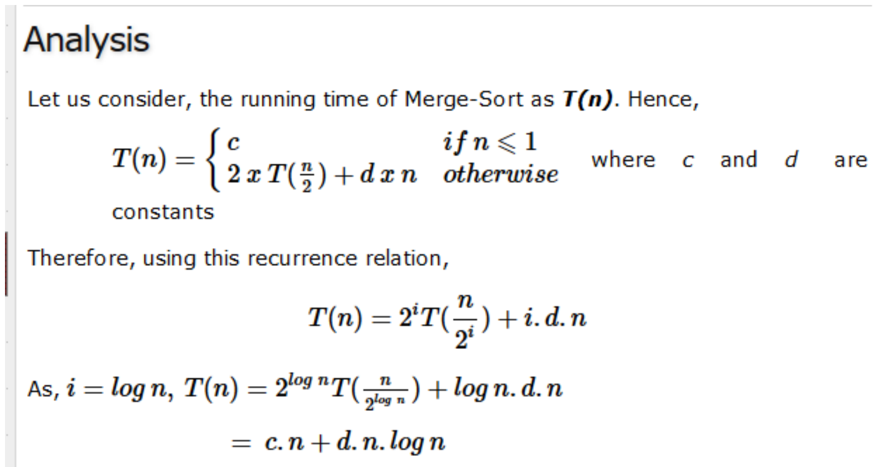
In this algorithm, the numbers are stored in an array ***numbers[]***. Here, ***p*** and ***q*** represents the start and end index of a sub-array.

Algorithm: Merge-Sort (numbers[], p, r)

```
if p < r then
q = ⌊(p + q) / 2⌋
Merge-Sort (numbers[], p, q)
   Merge-Sort (numbers[], q + 1, r)
   Merge (numbers[], p, q, r)
```

Function: Merge (numbers[], p, q, r)

```
$n_1 = q – p + 1$
$n_2 = r – q$
declareleftnums[1…$n_1$ + 1] and rightnums[1…$n_2$ + 1] temporary arrays
for i = 1 to $n_1$
leftnums[i] = numbers[p + i - 1]
for j = 1 to $n_2$
rightnums[j] = numbers[q+ j]
leftnums[$n_1$ + 1] = ∞
rightnums[$n_2$ + 1] = ∞
i = 1
j = 1
for k = p to r
ifleftnums[i] ≤ rightnums[j]
numbers[k] = leftnums[i]
    i = i + 1
else
numbers[k] = rightnums[j]
    j = j + 1
```



Analysis

Let us consider, the running time of Merge-Sort as ***T(n)***. Hence,

$$T(n) = \begin{cases} c & if\, n \leqslant 1 \\ 2\,x\,T(\frac{n}{2}) + d\,x\,n & otherwise \end{cases}$$ where c and d are constants

Therefore, using this recurrence relation,

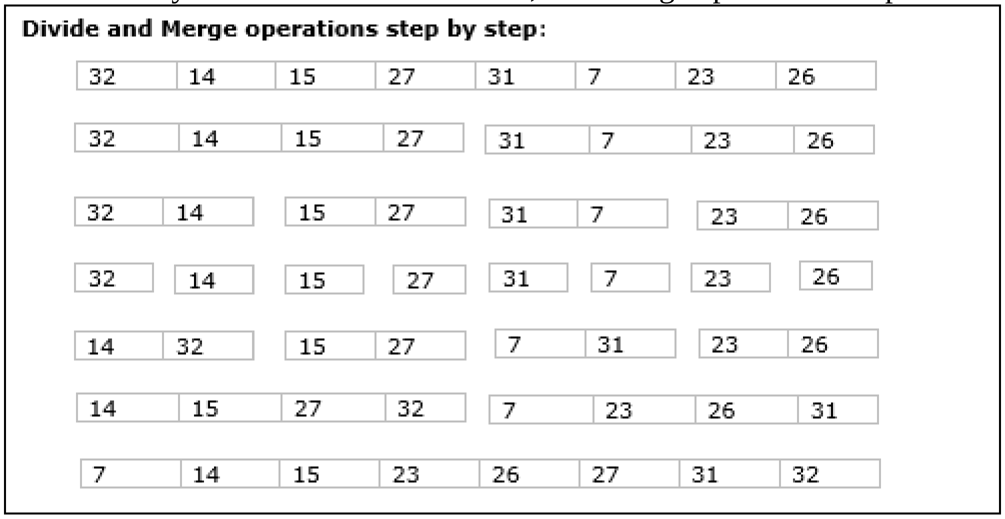
$$T(n) = 2^i T(\frac{n}{2^i}) + i.d.n$$

As, $i = \log n,\ T(n) = 2^{\log n} T(\frac{n}{2^{\log n}}) + \log n.d.n$

$$= c.n + d.n.\log n$$

Example

In the following example, we have shown Merge-Sort algorithm step by step. First, every iteration array is divided into two sub-arrays, until the sub-array contains only one element. When these sub-arrays cannot be divided further, then merge operations are performed.



5. What do you meant by Quick Sort? Explain

It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations as it is not difficult to implement. It is a good general purpose sort and it consumes relatively fewer resources during execution.

Advantages

- It is in-place since it uses only a small auxiliary stack.
- It requires only ***n (log n)*** time to sort **n** items.
- It has an extremely short inner loop.
- This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

Disadvantages

- It is recursive. Especially, if recursion is not available, the implementation is extremely complicated.
- It requires quadratic (i.e., n2) time in the worst-case.
- It is fragile, i.e. a simple mistake in the implementation can go unnoticed and cause it to perform badly.

Quick sort works by partitioning a given array ***A[p ... r]*** into two non-empty sub array ***A[p ... q]*** and ***A[q+1 ... r]*** such that every key in ***A[p ... q]*** is less than or equal to every key in ***A[q+1 ... r]***. Then, the two sub-arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index ***q*** is computed as a part of the partitioning procedure.

Algorithm: Quick-Sort (A, p, r)

```
if p < r then
  q Partition (A, p, r)
  Quick-Sort (A, p, q)
  Quick-Sort (A, q + r, r)
```

Note that to sort the entire array, the initial call should be ***Quick-Sort (A, 1, length[A])***
As a first step, Quick Sort chooses one of the items in the array to be sorted as pivot. Then, the array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move towards the left, while the elements that are greater than or equal to pivot will move towards the right.
Partitioning the Array
Partitioning procedure rearranges the sub-arrays in-place.

Function: Partition (A, p, r)

```
x ← A[p]
i ← p-1
j ← r+1
while TRUE do
  Repeat j ← j - 1
until A[j] ≤ x
  Repeat i← i+1
until A[i] ≥ x
if i < j then
exchange A[i] ↔ A[j]
else
return j
```

Analysis
The worst case complexity of Quick-Sort algorithm is ***O(n²)***. However using this technique, in average cases generally we get the output in ***O(n log n)*** time

6. Describe in detail about binary search with an example

Binary search can be performed on a sorted array. In this approach, the index of an element **x** is determined if the element belongs to the list of elements. If the array is unsorted, linear search is used to determine the position.
Solution
In this algorithm, we want to find whether element **x** belongs to a set of numbers stored in an array ***numbers[]***. Where ***l*** and ***r*** represent the left and right index of a sub-array in which searching operation should be performed.

Algorithm: Binary-Search(numbers[], x, l, r)

```
if l = r then
return l
else
m := ⌊(l + r) / 2⌋
if x ≤ numbers[m]  then
return Binary-Search(numbers[], x, l, m)
else
return Binary-Search(numbers[], x, m+1, r)
```

Analysis

Linear search runs in ***O(n)*** time. Whereas binary search produces the result in ***O(log n)*** time
Let **T(n)** be the number of comparisons in worst-case in an array of **n** elements.
Hence,
$T(n)=\{0T(n2)+1 if n=1 otherwise$

Using this recurrence relation $T(n)=logn$

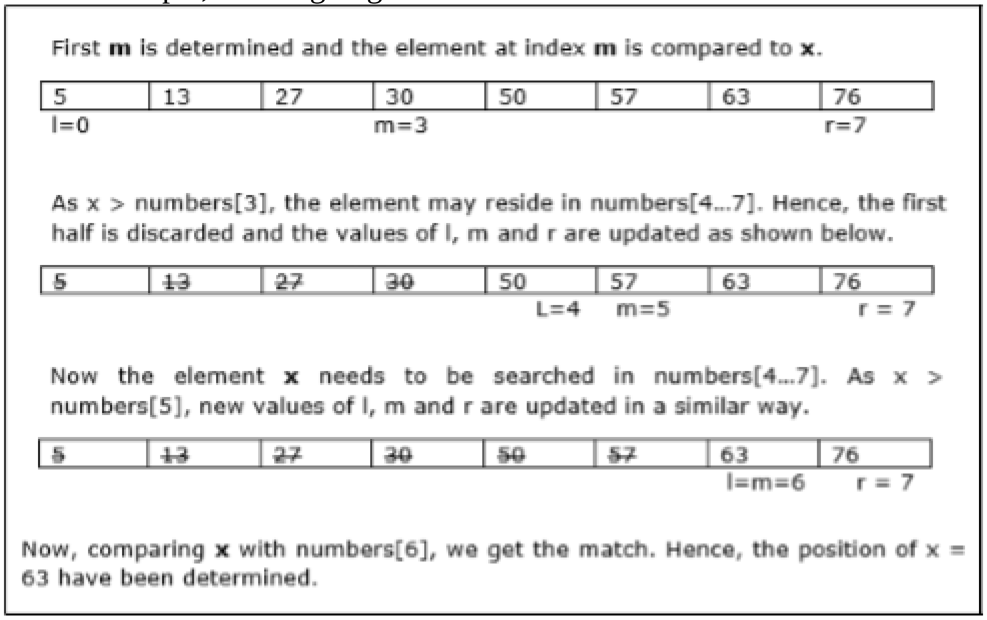
.

Therefore, binary search uses $O(logn)$

time.

Example

In this example, we are going to search element 63.



First **m** is determined and the element at index **m** is compared to **x**.

5	13	27	30	50	57	63	76
l=0			m=3				r=7

As x > numbers[3], the element may reside in numbers[4…7]. Hence, the first half is discarded and the values of l, m and r are updated as shown below.

~~5~~	~~13~~	~~27~~	~~30~~	50	57	63	76
				L=4	m=5		r = 7

Now the element **x** needs to be searched in numbers[4…7]. As x > numbers[5], new values of l, m and r are updated in a similar way.

~~5~~	~~13~~	~~27~~	~~30~~	~~50~~	~~57~~	63	76
						l=m=6	r = 7

Now, comparing **x** with numbers[6], we get the match. Hence, the position of x = 63 have been determined.

7.Explain in detail about Greedy Algorithms.

Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit.

This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

Components of Greedy Algorithm

Greedy algorithms have the following five components −

- **A candidate set** − A solution is created from this set.
- **A selection function** − Used to choose the best candidate to be added to the solution.
- **A feasibility function** − Used to determine whether a candidate can be used to contribute to the solution.
- **An objective function** − Used to assign a value to a solution or a partial solution.
- **A solution function** − Used to indicate whether a complete solution has been reached.

Areas of Application
Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using Dijkstra's algorithm.
- Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

Where Greedy Approach Fails
In many problems, Greedy algorithm fails to find an optimal solution, moreover it may produce a worst solution. Problems like Travelling Salesman and Knapsack cannot be solved using this approach.

8.Discuss in detail about the Knapsack Problem

The Greedy algorithm could be understood very well with a well-known problem referred to as Knapsack problem. Although the same problem could be solved by employing other algorithmic approaches, Greedy approach solves Fractional Knapsack problem reasonably in a good time. Let us discuss the Knapsack problem in detail.
Knapsack Problem
Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?
In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.
Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack
In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.
According to the problem statement,

- There are **n** items in the store
- Weight of **i**th item wi>0
- Profit for **i**th item pi>0and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction $\boldsymbol{x_i}$ of **i**th item.

$0 \leqslant xi \leqslant 1$

The **i**th item contributes the weight $xi.wi$to the total weight in the knapsack and profit $xi.pi$ to the total profit.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
do x[i] = 0
weight = 0
for i = 1 to n
if weight + w[i] ≤ W then
x[i] = 1
weight = weight + w[i]
else
x[i] = (W - weight) / w[i]
weight = W
break
return x
```

Hence, the objective of this algorithm is to

$$maximize \sum_{n=1}^{n}(x_i.pi)$$

subject to constraint,

$$\sum_{n=1}^{n}(x_i.wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n}(x_i.wi) = W$$

In this context, first we need to sort those items according to the value of $\frac{p_i}{w_i}$, so that $\frac{p_i+1}{w_i+1} \leq \frac{p_i}{w_i}$. Here, $\boldsymbol{x}$ is an array to store the fraction of items.

Analysis

If the provided items are already sorted into a decreasing order of $\frac{p_i}{w_i}$, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack ***W*** = **60** and the list of provided items are shown in the following table –

Item	**A**	**B**	**C**	**D**
Profit	280	100	120	120
Weight	40	10	20	24
Ratio $\left(\frac{p_i}{w_i}\right)$	7	10	6	5

As the provided items are not sorted based on $\frac{p_i}{w_i}$. After sorting, the items are as shown in the following table.

Item	**B**	**A**	**C**	**D**
Profit	100	280	120	120
Weight	10	40	20	24
Ratio $\left(\frac{p_i}{w_i}\right)$	10	7	6	5

Solution

After sorting all the items according to *pi/wi*
.First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.
Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.
Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.
The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**
And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**
This is the optimal solution. We cannot gain more profit selecting any different combination of items

9. Explain in detail about Dynamic Programming

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.
Overlapping Sub-Problems
Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.
For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.
Optimal Sub-Structure
A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.
For example, the Shortest Path problem has the following optimal substructure property −
If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.
The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.
Steps of Dynamic Programming Approach
Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

10. Discuss in detail about in Optimal Binary Search Tree

A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null nodes. The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.
When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a BST, which has minimal expected cost of locating each node
Search time of an element in a BST is ***O(n)***, whereas in a Balanced-BST search time is ***O(log n)***. Again the search time can be improved in Optimal Cost Binary Search Tree, placing the most

frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.

Here, the Optimal Binary Search Tree Algorithm is presented. First, we build a BST from a set of provided **n** number of distinct keys < ***k_1, k_2, k_3, ...k_n***>. Here we assume, the probability of accessing a key ***K_i*** is ***p_i***. Some dummy keys (***d_0, d_1, d_2, ...d_n***) are added as some searches may be performed for the values which are not present in the Key set ***K***. We assume, for each dummy key ***d_i*** probability of access is ***q_i***.

Optimal-Binary-Search-Tree(p, q, n)

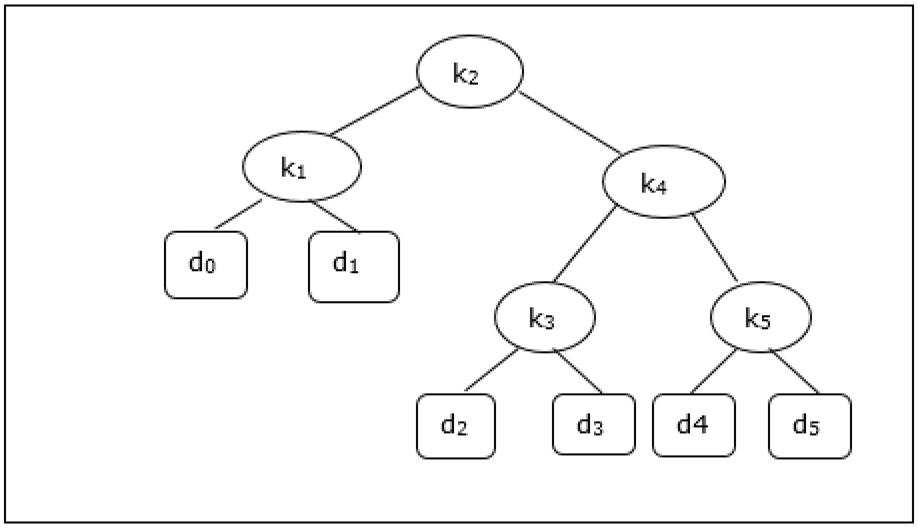
```
e[1…n + 1, 0…n],
w[1…n + 1, 0…n],
root[1…n + 1, 0…n]
for i = 1 to n + 1 do
e[i, i - 1] := q_i - 1
w[i, i - 1] := q_i - 1
for l = 1 to n do
for i = 1 to n – l + 1 do
    j = i + l – 1 e[i, j] := ∞
w[i, i] := w[i, i -1] + p_j + q_j
for r = i to j do
t := e[i, r - 1] + e[r + 1, j] + w[i, j]
if t < e[i, j]
e[i, j] := t
root[i, j] := r
return e and root
```

Analysis

The algorithm requires **O (n^3)** time, since three nested **for** loops are used. Each of these loops takes on at most **n** values.

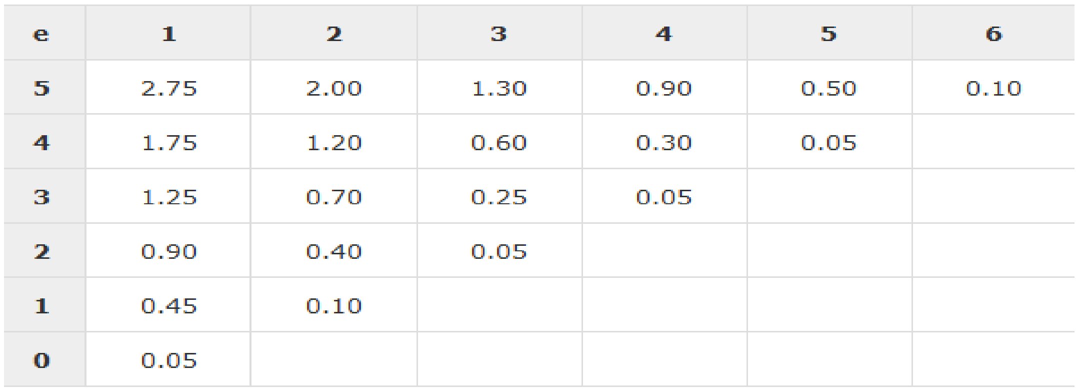
Example

Considering the following tree, the cost is 2.80, though this is not an optimal result.

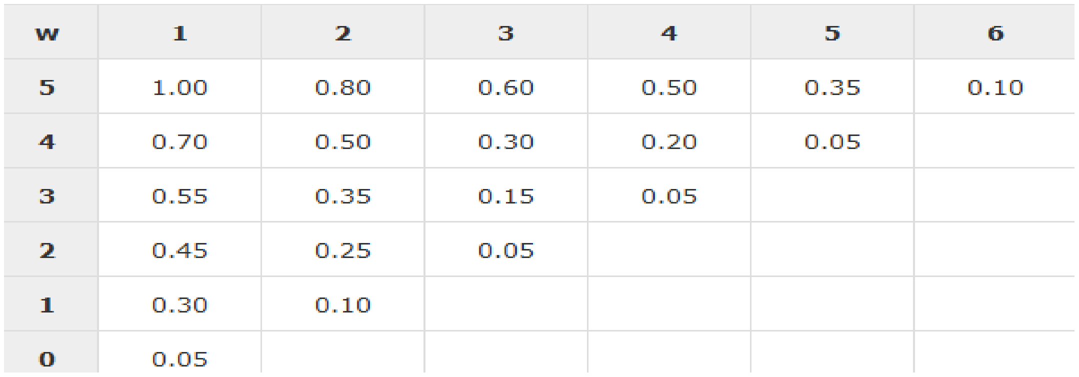


Node	Depth	Probability	Contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

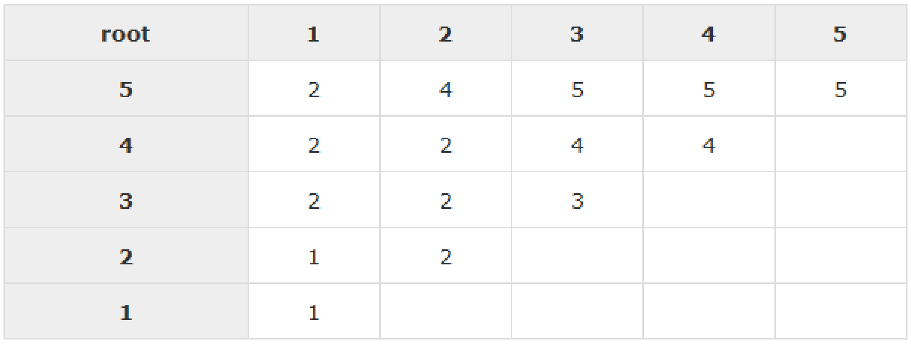
To get an optimal solution, using the algorithm discussed in this chapter, the following tables are generated.
In the following tables, column index is **i** and row index is **j**.



e	1	2	3	4	5	6
5	2.75	2.00	1.30	0.90	0.50	0.10
4	1.75	1.20	0.60	0.30	0.05	
3	1.25	0.70	0.25	0.05		
2	0.90	0.40	0.05			
1	0.45	0.10				
0	0.05					



w	1	2	3	4	5	6
5	1.00	0.80	0.60	0.50	0.35	0.10
4	0.70	0.50	0.30	0.20	0.05	
3	0.55	0.35	0.15	0.05		
2	0.45	0.25	0.05			
1	0.30	0.10				
0	0.05					

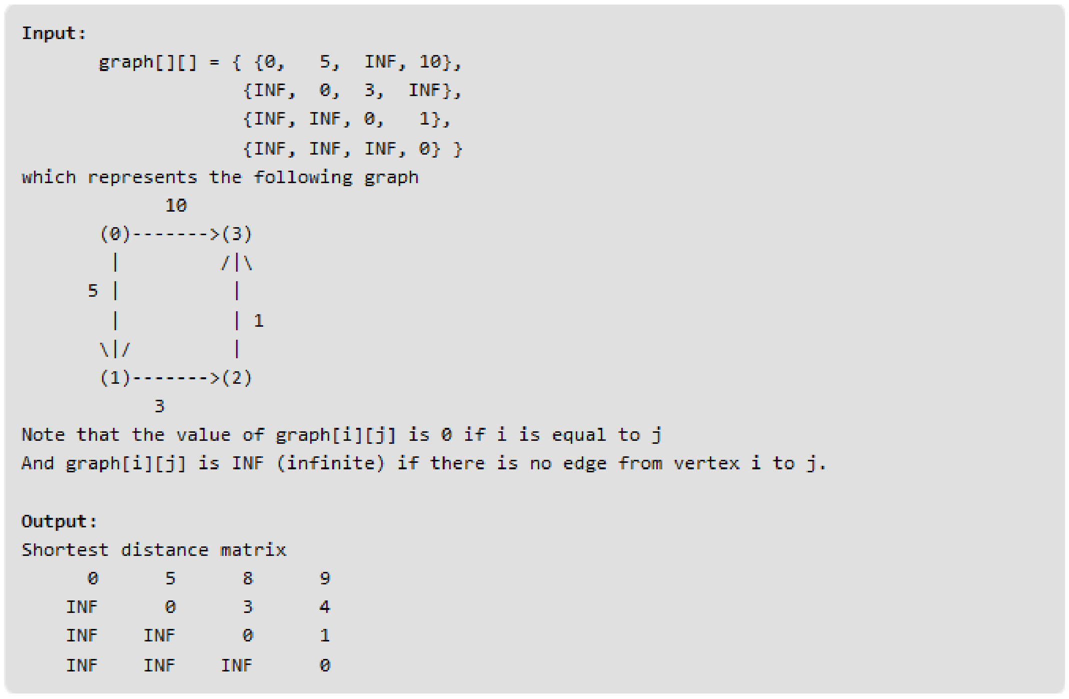


root	1	2	3	4	5
5	2	4	5	5	5
4	2	2	4	4	
3	2	2	3		
2	1	2			
1	1				

From these tables, the optimal tree can be formed.

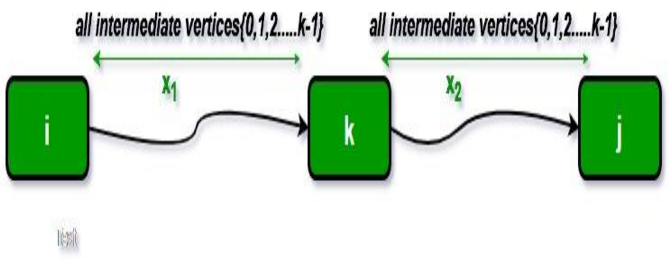
11.Explain in detail about the Warshall‟s Algorithm for Finding Transitive Closure.

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph. Example:



```
Input:
       graph[][] = { {0,   5,  INF, 10},
                    {INF,  0,   3,  INF},
                    {INF, INF,  0,   1},
                    {INF, INF, INF, 0} }
which represents the following graph
             10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3
Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

Output:
Shortest distance matrix
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

Floyd Warshall Algorithm
We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
1)k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
2)k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].
The following figure shows the above optimal substructure property in the all-pairs shortest path problem.


all intermediate vertices{0,1,2.....k-1}
all intermediate vertices{0,1,2.....k-1}
x1
x2
i
k
j

UNIT-V

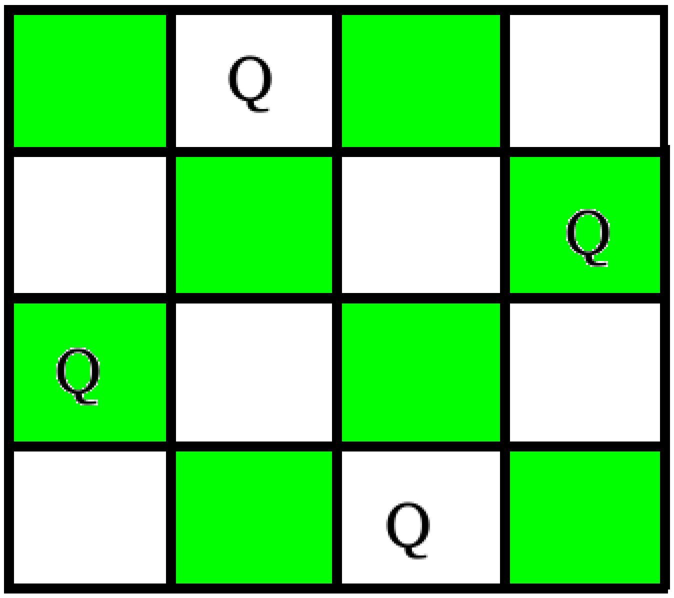
1. Discuss in detail about Backtracking with N-Queens Problem

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other. Given an integer n, print all distinct solutions to the n-queens puzzle. Each solution contains distinct board configurations of the n-queens' placement, where the

ADVANCED ALGORITHM DESIGN AND ANALYSIS

– N-Queen's Problem - Branch and Bound – Assignment Problem - P & NP problems – NP-complete problems – Approximation algorithms for NP-hard problems – Traveling salesman problem-Amortized Analysis.

solutions are a permutation of [1,2,3..n] in increasing order, here the number in the *ith* place denotes that the *ith*-column queen is placed in the row with that number. For eg below figure represents a chessboard [3 1 4 2].



Algorithm

1) Start in the leftmost column
2) If all queens are placed
return true
3) Try all rows in the current column. Do following for every tried row.
a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
b) If placing the queen in [row, column] leads to a solution then return true.

c) If placing queen doesn't lead to a solution then umark this [row, column] (Backtrack) and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked, return false to trigger backtracking.

Input:
The first line of input contains an integer T denoting the no of test cases. Then T test cases follow. Each test case contains an integer n denoting the size of the chessboard.

Output:
For each test case, output your solutions on one line where each solution is enclosed in square brackets '[', ']' separated by a space . The solutions are permutations of {1, 2, 3 …, *n*} in increasing order where the number in the ith place denotes the ith-column queen is placed in the row with that number, if no solution exists print -1.

Constraints:
1<=T<=10
1<=n<=10

Example:
Input
2
1
4
Output:
[1]
[2 4 1 3] [3 1 4 2]

2. Describe in detail about P and NP problems

many problems are solved where the objective is to maximize or minimize some values, whereas in other problems we try to find whether there is a solution or not. Hence, the problems can be categorized as follows −

Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

- Finding the minimum number of colors needed to color a given graph.
- Finding the shortest path between two vertices in a graph.

Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

- Whether a given graph can be colored by only 4-colors.
- Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

What is Language?

Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer 'yes' for a specific input. A language is the totality of inputs for which the answer is Yes. Most of the algorithms discussed in the previous chapters are **polynomial time algorithms**.

For input size ***n***, if worst-case time complexity of an algorithm is $O(n^k)$, where***k*** is a constant, the algorithm is a polynomial time algorithm.

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time. However there are many problems, such as traveling salesperson, optimal graph coloring, Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the **NP-Complete** problems, whose status is unknown.

In this context, we can categorize the problems as follows −

P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where **k** is constant.

These problems are called **tractable**, while others are called **intractable or superpolynomial**.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial$p(n)$ such that the algorithm can solve any instance of size **n** in a time$O(p(n))$.

Problem requiring $\Omega(n^{50})$ time to solve are essentially intractable for large ***n***. Most known polynomial time algorithm run in time $O(n^k)$ for fairly low value of***k***.

The advantages in considering the class of polynomial-time algorithms is that all reasonable **deterministic single processor model of computation** can be simulated on each other with at most a polynomial slow-d

NP-Class

The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren’t asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in *NP - P* are intractable.

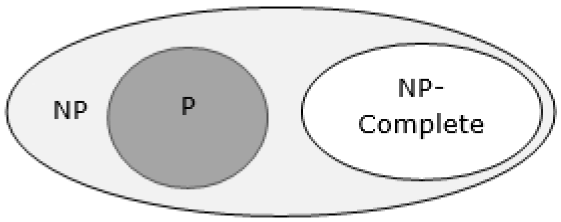
It is not known whether $P = NP$. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If $P \neq NP$, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class **P** if it’s easy to find a solution for the problem. The problem belongs to **NP**, if it’s easy to check a solution that may have been very tedious to find.

4. Explain in detail about NP hard problems

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

5. Describe in detail about NP complete problems

Definition of NP-Completeness

A language **B** is ***NP-complete*** if it satisfies two conditions

- **B** is in NP
- Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove ***TSP is NP-Complete***, first we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle*** $\leq_p$ ***TSP*** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.
Hence, an instance of TSP is constructed. We create the complete graph ***G′ = (V, E′)***, where
E′={(*i*,*j*):*i*,*j*∈*Vandi*≠*j*EijijVandij
Thus, the cost function is defined as follows −
t(*i*,*j*)={01*if*(*i*,*j*)∈*Eotherwise*tij0ifijE1otherwise
Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in ***G′*** as each edge belongs to ***E***. Therefore, ***h*** has a cost of**0** in ***G′***. Thus, if graph ***G*** has a Hamiltonian cycle, then graph ***G′*** has a tour of **0**cost.
Conversely, we assume that ***G′*** has a tour ***h′*** of cost at most **0**. The cost of edges in ***E′*** are **0** and **1** by definition. Hence, each edge must have a cost of **0**as the cost of ***h′*** is **0**. We therefore conclude that ***h′*** contains only edges in ***E***.
We have thus proven that ***G*** has a Hamiltonian cycle, if and only if ***G′*** has a tour of cost at most **0**. TSP is NP-complete.

6. Discuss in detail about amortized analysis

Amortized Analysis
Amortized analysis is generally used for certain algorithms where a sequence of similar operations are performed.

- Amortized analysis provides a bound on the actual cost of the entire sequence, instead of bounding the cost of sequence of operations separately.
- Amortized analysis differs from average-case analysis; probability is not involved in amortized analysis. Amortized analysis guarantees the average performance of each operation in the worst case.

It is not just a tool for analysis, it's a way of thinking about the design, since designing and analysis are closely related.
Aggregate Method
The aggregate method gives a global view of a problem. In this method, if **n**operations takes worst-case time ***T(n)*** in total. Then the amortized cost of each operation is ***T(n)/n***. Though different operations may take different time, in this method varying cost is neglected.
Accounting Method
In this method, different charges are assigned to different operations according to their actual cost. If the amortized cost of an operation exceeds its actual cost, the difference is assigned to the object as credit. This credit helps to pay for later operations for which the amortized cost less than actual cost.
If the actual cost and the amortized cost of **ith** operation are *ci*ci and *cl*^cl, then
∑*i*=1*ncl*^⩾∑*i*=1*nci*i1ncli1nci
Potential Method
This method represents the prepaid work as potential energy, instead of considering prepaid work as credit. This energy can be released to pay for future operations.
If we perform ***n*** operations starting with an initial data structure ***D_0***. Let us consider, ***c_i*** as the actual cost and ***D_i*** as data structure of **ith** operation. The potential function Φ maps to a real number Φ(***D_i***), the associated potential of***D_i***. The amortized cost *cl*^cl can be defined by
cl^=*ci*+Φ(*Di*)−Φ(*Di*−1)clciΦDiΦDi1
Hence, the total amortized cost is

$\sum i=1ncl\hat{}=\sum i=1n(ci+\Phi(Di)-\Phi(Di-1))=\sum i=1nci+\Phi(Dn)-\Phi(D0)$i1ncli1nciΦDiΦDi1i1nciΦDnΦD0

Dynamic Table

If the allocated space for the table is not enough, we must copy the table into larger size table. Similarly, if large number of members are erased from the table, it is a good idea to reallocate the table with a smaller size.

Using amortized analysis, we can show that the amortized cost of insertion and deletion is constant and unused space in a dynamic table never exceeds a constant fraction of the total space.