Dt: 24/11/2020

D.S Assignment-2

Name: T. HIMABINDU

Branch: CSE-4

Regd No: 19131A05N7

UNIT-3

1 (a) Write a program for the implementation of Double link list ?

Ans:

→ Double linked list is a type of data structures that is made up of nodes that are created using self referential structures.

→ Each of these nodes contain three parts;

(A) The Data

(B) The reference to the next list node.

(C) The reference to the previous list node.

→ Only the reference to the first list node is required to access the whole linked list.

This is known as Head.

→ The last node in the listed "points to nothing" so it stores NULL in that part.

A C++ program to implement doubly linked list is given as follows:

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    struct Node *prev;
    struct Node *next;
};
struct Node*head = NULL;
void insert(int newdata) {
  struct Node* newnode=(struct Node*) malloc (sizeof(
                                        struct Node));
  newnode -> data = newdata;
  newnode -> prev = NULL;
  newnode -> next = head;
  if (head != NULL)
  head -> prev = newnode;
  head = newnode;
}
```

```
void display() {
    struct Node* ptr;
    ptr = head;
    while (ptr != NULL) {
        cout << ptr -> data << " ";
        ptr = ptr -> next;
    }
}

int main() {
    insert(3);
    insert(1);
    insert(7);
    insert(2);
    insert(9);
    cout << "The doubly linked list is:";
    display();
    return 0;
}
```

OUTPUT:

The doubly linked list is: 9 2 7 1 3

(b) Explain how to insert a node in the middle of singly linked list?

Ans:

→ Given a linked list containing "n" nodes. The problem is to insert a new node with "data x" at the middle of the list.

→ If n is even, then
insert the new node after the $(n/2)^{th}$ node.

else
insert the new node after the $(n+1)/2^{th}$ node.

Examples:

(1) List : 1 2 4 5

new data (x) = 3

Output : 1 2 3 4 5 (∵ n = even)

(2) List : 5 10 4 32 16

New data(x) = 41

Output : 5 10 4 41 32 16 (∵ n = odd)

→ We have a method using length of the linked list to solve inserting new node in middle of singly linked list.

Soln:

Find the number of nodes (or) length of linked list using one traversal.

let it be "len".

→ Calculate c:

c = (len/2). ---> if len is even.

c = (len+1)/2 ---> if len is odd.

→ Traverse again the first 'c' nodes and insert the new node after the c^{th} node.

Program (C++):

```
#include <bits/stdc++.h>
using namespace std;
// Structure of a node //
struct Node {
    int data;
    Node* next;
};
```

```
// function to create and return a node //
Node* getNode(int data)
{
    // Allocating space //
    ~~Node* getNode~~
    Node* newNode = (Node*) malloc (sizeof (Node));

    // Inserting the required data //
    newNode -> data = data;
    newNode -> next = NULL;
    return newNode;
}
// function to insert node at middle //
void insertAtMid (Node** head_ref, int x)
{
    // if list is empty //
    if (*head_ref == NULL)
     *head_ref = getNode (x);
    else {
        // get a new node //
        Node* newNode = getNode(x);
```

```
Node* ptr = *head_ref;
int len = 0;
// calculate length of linked list i.e., no.of nodes //
while (ptr != NULL) {
    len++;
    ptr = ptr->next;
}
// count the no.of nodes after which new node inserting //
int count = (((len % 2) == 0) ? (len/2) : (len+1)/2;
ptr = *head_ref;
// 'ptr' points to node after which new node to be insert //
newNode -> next = ptr -> next;
ptr -> next = newNode;
}
}
// function to display the linked list //
void display (Node* head)
{
while (head != NULL) {
    cout << head -> data << " ";
    head = head -> next;
```

```
    }
}
// Driver program to test above //
int main()
{
    // Creating the list 1->2->4->5 //
    Node* head = NULL;
    head = getNode(1);
    head->next = getNode(2);
    head->next->next = getNode(4);
    head->next->next->next = getNode(5);
    cout << " Linked list before insertion: ";
    display(head);
    int x=3;
    insertAtMid(&head, x);
    cout << "\n Linked list after insertion: ";
    return 0;
}
```

Time Complexity
O(n)

OUTPUT:

Linked list before insertion : 1 2 4 5
Linked list after insertion : 1 2 3 4 5

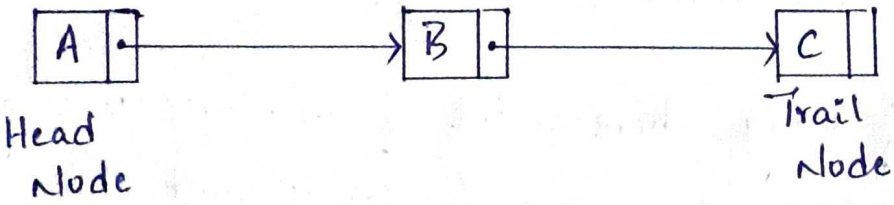
2

a) What are the advantages and disadvantages of linked list ?

Ans: → It is a data structure in which elements are linked using pointers.

→ A Node represents an element in linked list which have some data and a pointer pointing to next node.

→ The structure of a linked list looks as below:



ADVANTAGES OF LINKED LIST

Dynamic Data Structure :

→ Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and deallocating memory.

→ So there is no need to give initial size of linked list.

Insertion and deletion:

→ Insertion and deletion of nodes are really easier.

→ Unlike array, here we don't have to shift elements after insertion or deletion of an element.

→ In linked list, we have to update the address present in next pointer of a node.

No Memory Wastage:

→ As size of linked list can increase or decrease at run time so there is no memory wastage.

→ In case of array, there is lot of memory wastage like if we declare an array of size 10 and store only 6 elements in it then space of 4 elements are wasted.

→ There is no such problem in linked list as memory is allocated only when required.

Implementation:

Data structures such as stack and queues can be easily implemented using linked list.

DISADVANTAGES OF LINKED LIST

Memory Usage :

→ More memory is required to store elements in linked list as compared to array.

→ Because in linked list each node contains a pointer and it requires extra memory for itself.

Traversal :

→ Elements or nodes traversal is difficult in linked list.

→ We cannot randomly access any element as we do in array by index.

→ For example, we have to traverse all the nodes before it if we want to access a node at position n.

→ So, time required to access a node is large.

Reverse Traversing :

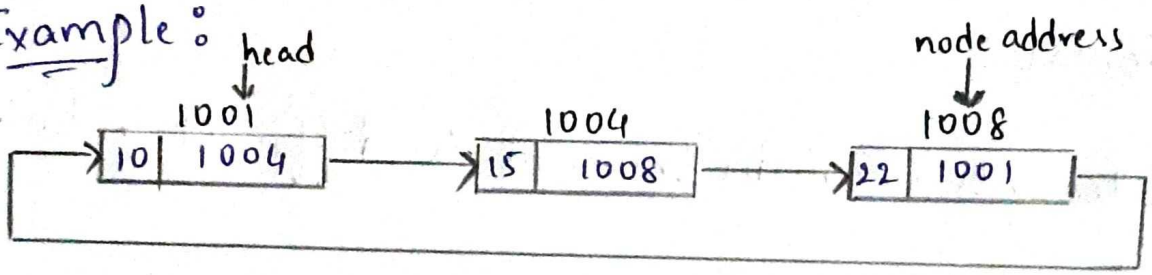
→ In linked list, reverse traversing is really difficult.

→ In doubly link list, extra memory is required for back pointer hence wastage of memory.

(b) Write a program for circular linked list ?

Ans: A Circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

Example:



C++ program to create circular linked list

```
#include <bits/stdc++.h>
using namespace std;
struct Node
{
   int data;
   struct Node *next;
};
struct Node *addToEmpty(struct Node * last, int data)
{
```

```
// This function is only for empty list
    if (last != NULL)
     return last;
// Creating a node dynamically
    struct Node * temp = (struct Node*) malloc (sizeof
                                (struct Node));
// Assigning the data
    temp -> data = data;
    Last = temp;
// Creating the link
    Last -> next = last;
    return last;
}
struct Node *addBegin(struct Node *last, int data)
{
    if (last == NULL)
      return addToEmpty (last, data);
    struct Node * temp = (struct Node *) malloc (sizeof(
                                struct Node));
    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;
```

```
    return last;
}
struct Node *addEnd (struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty (last, data);
    struct Node *temp = (struct Node*) malloc (sizeof(
                                    structNode));

    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;
    last = temp;
    return last;
}
struct Node *addAfter (struct Node *last, int data,
                                    int item)
{
    if (last == NULL)
        return NULL;
    struct Node *temp, *p;
    p = last -> next;
    do
{
```

```
if (p -> data == item)
{
temp =(struct Node *) malloc(sizeof (struct Node));
Temp -> data = data;
temp -> next = p -> next;
p -> next = temp;
if (p == last)
    last = temp;
return last;
}
p = p -> next;
}
while (p != last -> next);
cout << item << "not present in list." <<endl;
return last;
}
void Traverse (struct Node * last)
{
 struct Node * p;
// if list is empty, return.
if (last == NULL)
{
cout << "List is empty." << endl;
return;
}
```

```
// Pointing to first Node of list.
  p = last -> next ;

// Traversing the list.
  do
  {
    cout << p -> data << " ";
    p = p -> next;
  }
  while (p != last -> next);
}

// Driven Program
int main()
{
  struct Node * last = NULL;
  last = addToEmpty (last, 6);
  last = addBegin (last, 4);
  last = addBegin (last, 2);
  last = addEnd (last, 8);
  last = addEnd (last, 12);
  last = addAfter (last, 10, 8);
  traverse (last);
  return 0;
}
```

OUTPUT

2 4 6 8 10 12.

3 Give an algorithm to perform following operations in a double linked list.

(a) Insert a new node after a given node.

(b) Delete last node.

Ans: (a) Algorithm to insert a new node after a given node in double linked list:

→ In order to insert a new node after the specified node in the list, we need to skip the required no. of nodes to reach the mentioned node and then make the pointer adjustments as required.

USE THE FOLLOWING STEPS FOR THIS PURPOSE.

→ Allocate the memory for the new node. Use the following statements for this.

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

→ ~~Traverse the memory for new node. Use the following~~

→ Traverse the list by using the pointer "temp" to skip the required number of nodes in order to reach the specified node.

```
temp = head;
for (i = 0; i < loc; i++)
{
 temp = temp -> next;
 if (temp == NULL) // temp will be null if the list
 {                    doesn't last long upto mentioned
                      location //
  return;
 }
}
```

→ The Temp would point to the specified node at the end of the "for loop." The new node needs to be inserted after this node therefore we need to make a few pointer adjustments here. Make the next pointer of ptr point to the next node of temp.

```
ptr -> next = temp -> next;
```

→ Make the prev of the new node ptr point to temp.

```
ptr -> prev = temp;
```

→ Make the next pointer of temp point to the new node ptr.

```
temp -> next = ptr ;
```

→ Make the previous pointer of the next node of temp point to the new node.

```
temp -> next -> prev = ptr ;
```

ALGORITHM :

STEP 1: If PTR = NULL
Write OVERFLOW
Go to step 15
(END OF IF)

STEP 2: SET NEW_NODE = PTR

STEP 3: SET PTR = PTR → NEXT

STEP 4: SET NEW_NODE → DATA = VAL

STEP 5: SET TEMP = START

STEP 6: SET I = 0

STEP 7: REPEAT 8 to 10 until I.

STEP 8: SET TEMP = TEMP → NEXT

STEP 9: IF TEMP = NULL

STEP 10: Write " LESS THAN DESIRED NUMBER OF ELEMENTS"

Goto STEP 15

(END OF IF)

(END OF LOOP)

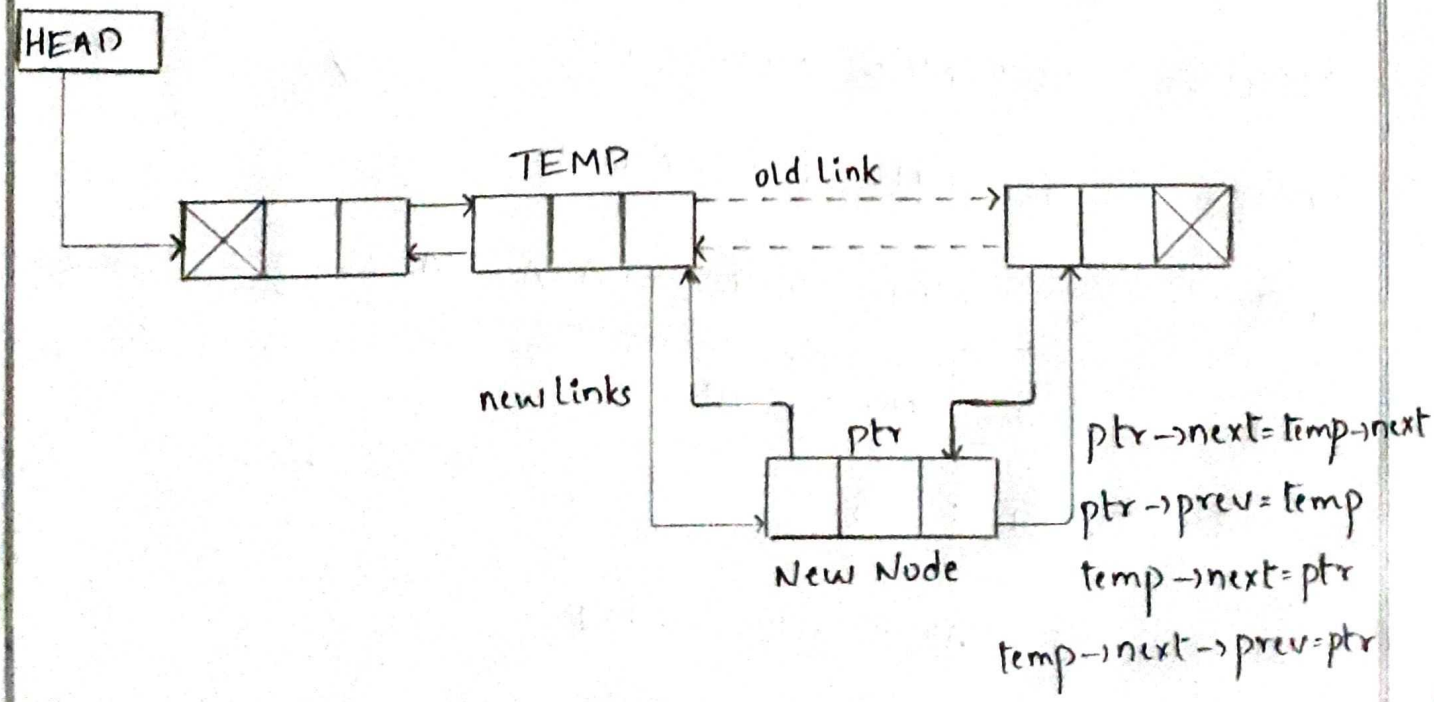
STEP 11 : SET NEW_NODE → NEXT = TEMP → NEXT

STEP 12 : SET NEW_NODE → PREV = TEMP.

STEP 13 : SET TEMP → NEXT = NEW_NODE

STEP 14: SET TEMP → NEXT → PREV = NEW_NODE.

STEP 15 : EXIT



Insertion into doubly linked list after specified node

(b) Deletion in doubly linked list at the End:

Ans: Deletion of last Node needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

ALGORITHM

Step 1: If HEAD = NULL

Write UNDERFLOW!

Go to step 7

(END OF IF)

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> NEXT != NULL.

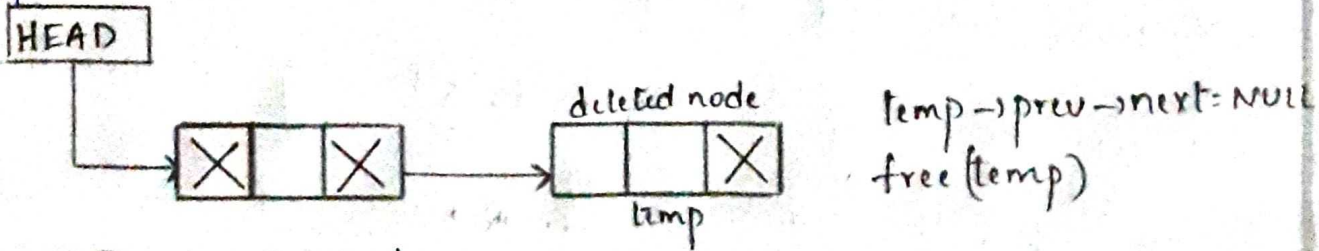
Step 4: SET TEMP = TEMP -> NEXT

(END OF LOOP)

Step 5: SET TEMP -> PREV -> NEXT = NULL

Step 6: FREE TEMP

Step 7: EXIT



Deletion in doubly linked list at end

UNIT-4

1 (a) What is binary tree ? How to represent binary tree ? Explain.

Ans:

BINARY TREE

A Binary Tree is a hierarchial data structure in which each node has at most two children generally referred as left and right child.

Each node contains three components :

1. Pointer to left subtree
2. Pointer to right subtree
3. Data element

The topmost node in the tree is called the root. An empty tree is represented by NULL pointer.

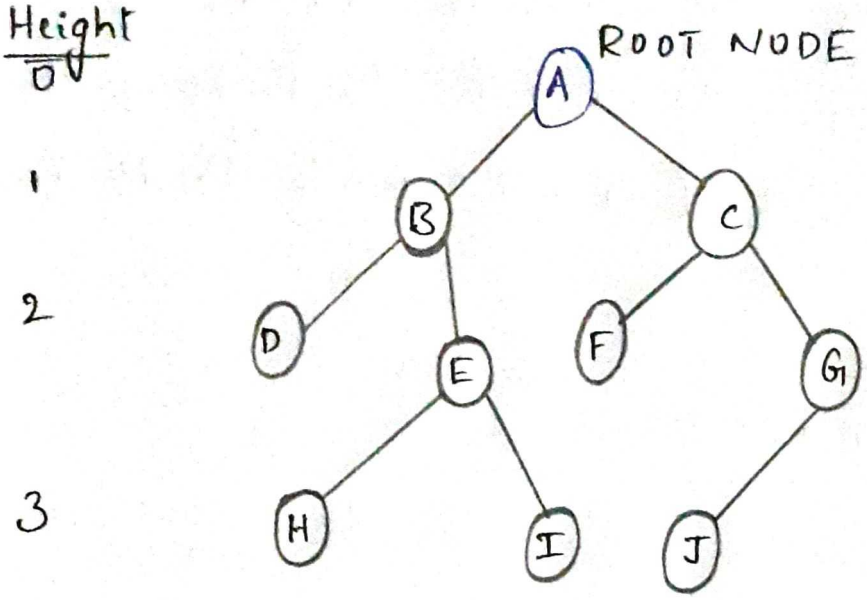
→ Binary Trees provides an efficient insertion and searching.

→ Trees reflect structural relationships in the data.

→ Trees are used to represent hierarchies.

→ Trees are very flexible data, allowing to move subtrees around with minimum effort.

BINARY TREE REPRESENTATION



ROOT: Topmost node in a Tree

PARENT: Every node (including a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent.

CHILD: A Node directly connected to another node when moving away from the root.

LEAF/EXTERNAL NODE: Node with no children.

INTERNAL NODE: Node with atleast one children.

DEPTH OF A NODE: No. of edges from the root to the node.

HEIGHT OF A NODE: No. of edges from the node to the deepest leaf. Height of tree is the height of the root.

In the above binary tree we see that the root node is A. The tree has 10 nodes with 5 internal nodes, i.e., A, B, C, E, G and 5 external nodes, i.e., D, F, H, I, J. The height of the tree is 3. B is the parent of D and E while D and E are children of B.

TYPES OF BINARY TREE

→ Rooted Binary Tree
→ Full binary tree
→ Perfect binary tree
→ Complete binary tree
→ Balanced binary tree
→ Degenerate tree.

(b) Explain the delete operation of Binary search tree with an example?

Ans: → Delete function is used to delete the specified node from a binary search tree.

→ However, we must delete a node from a binary search tree in such a way that the property of binary search tree

doesn't violate.

There are three situations of deleting a node from binary search tree.

(1) The Node to be deleted is a leaf Node

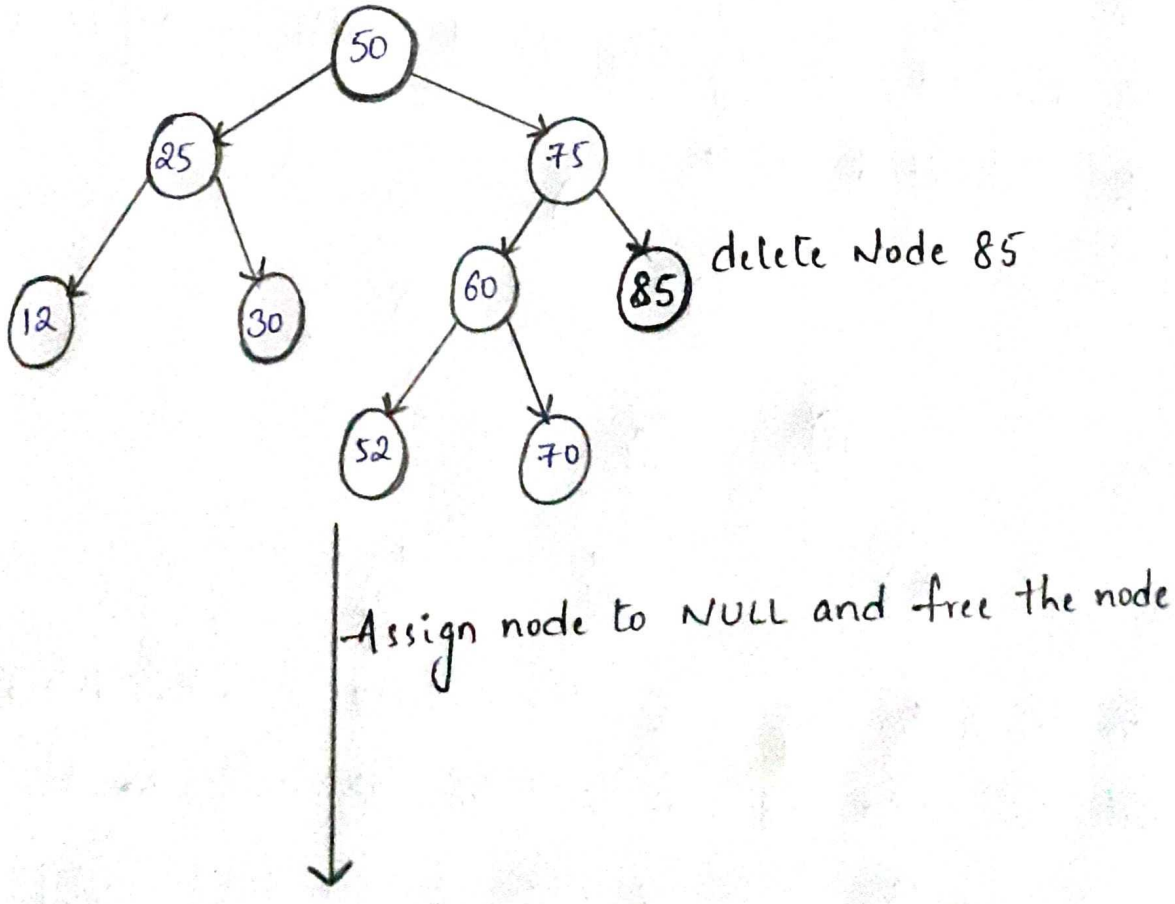
(2) The Node to be deleted has only one child.

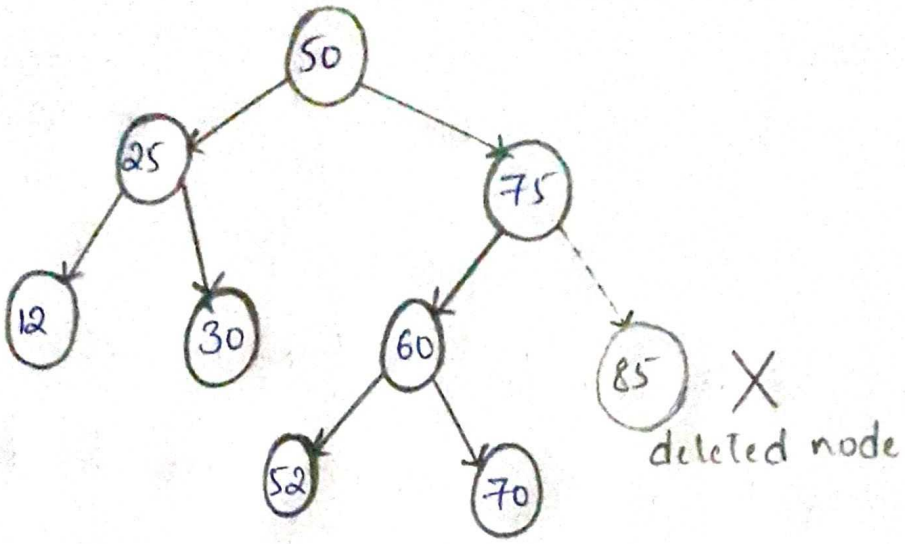
(3) The Node to be deleted has 2 children (can be root also)

(1) THE NODE TO BE DELETED IS A LEAF NODE

→ It is the simplest case.

→ In this case, replace the leaf node with the NULL and simple free the allocated space.

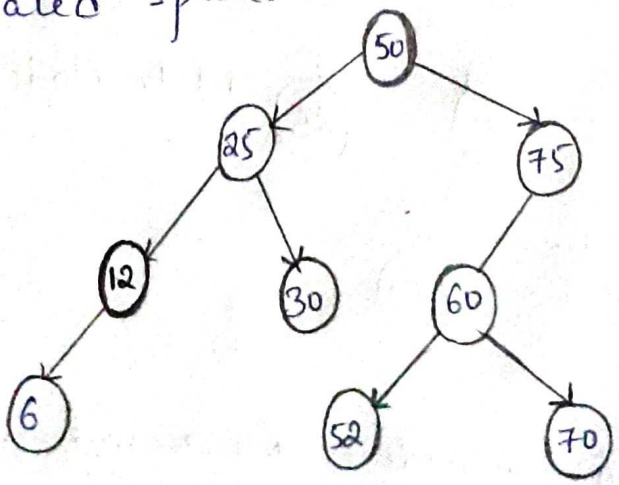




→ Here we delete the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.

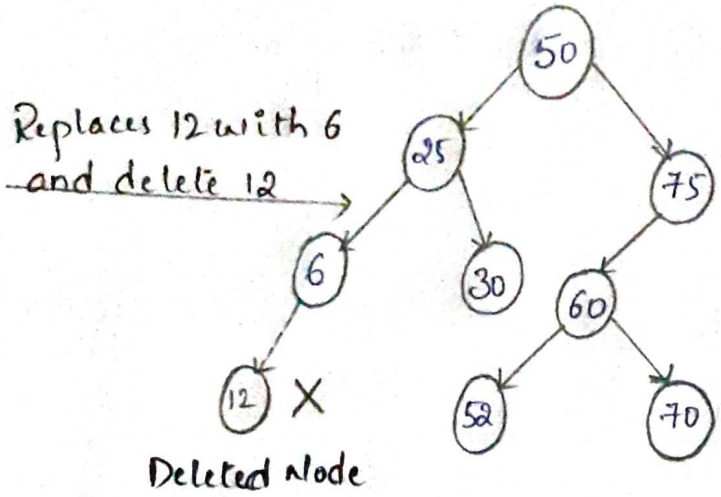
(2) THE NODE TO BE DELETED HAS ONLY ONE CHILD.

→ In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.



In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node)

will simply be deleted.



(3) THE NODE TO BE DELETED HAS TWO CHILDREN

A Node with two children may be deleted from the BST in the following two ways:

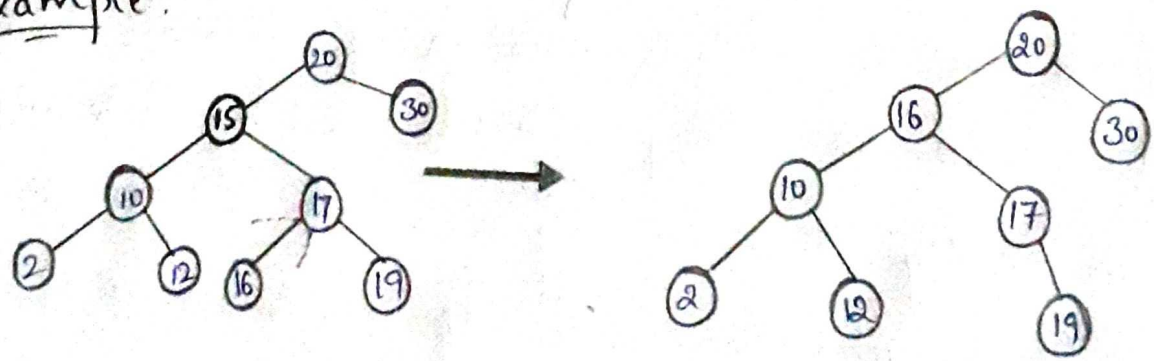
METHOD-01:

→ Visit to the right subtree of the deleting node.

→ Pluck the least value element called as "inorder successor".

→ Replace the deleting element with its inorder successor.

Example:



Here the Node with value = 15 is deleted from the BST.

METHOD-2

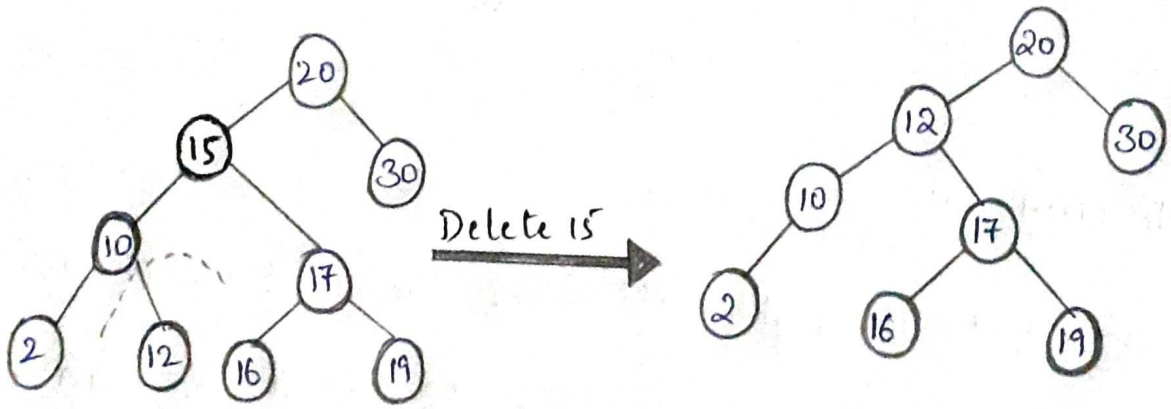
→ Visit to the left subtree of the deleting node.

→ Pluck the greatest value element called as inorder predecessor.

→ Replace the deleting element with its inorder predecessor.

Example:

Consider the following example where node with value = 15 is deleted from the BST-



2

a) Explain about the binary tree traversing techniques?

Ans: Tree Traversal is a finite collection of elements and The tree traversal is a process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner. If search result in a visit to all the vertices, it is called a traversal.

There are basically three traversal techniques for a binary tree that are,

1. Preorder Traversal
2. Inorder Traversal
3. Postorder Traversal.

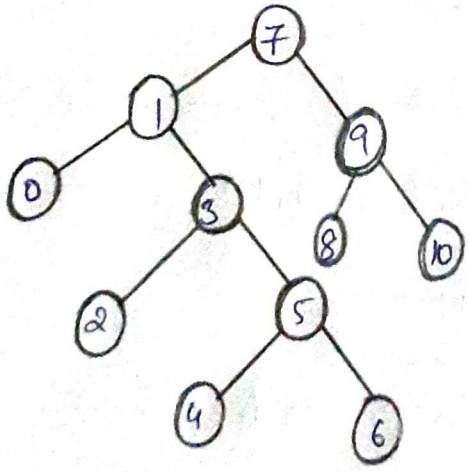
1) PREORDER TRAVERSAL:

To traverse a binary tree in preorder, following operations are carried out:

1. Visit the Root
2. Traverse the left sub tree of root.
3. Traverse the right sub tree of root.

* Preorder Traversal is also known as NLR-traversal.

Example:



∴ The PreOrder Traversal of the above tree will be:

7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10.

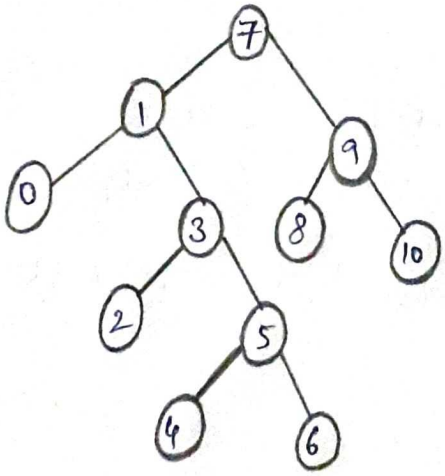
2) INORDER TRAVERSAL:

To traverse a binary tree inorder traversal, following operations are carried out:

1. Traverse the left most sub tree.
2. Visit the root.
3. Traverse the right most sub tree.

* InOrder Traversal is also known as LNR traversal.

Example:



Therefore, the inorder traversal of above will be:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

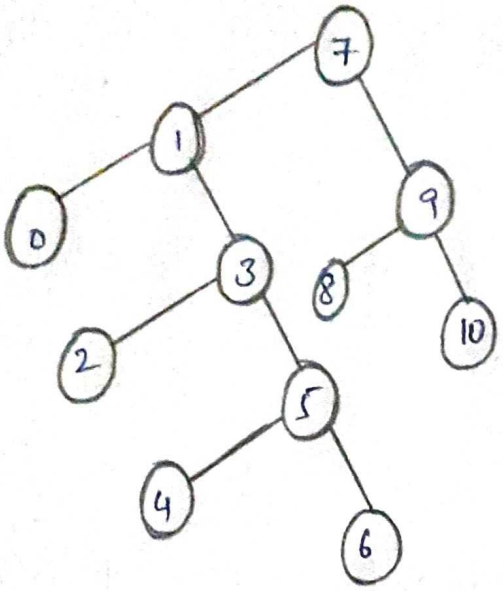
3) POSTORDER TRAVERSAL:

To traverse a binary tree in postorder traversal, following operationals are carried out:

1. Traverse the left sub tree of root.
2. Traverse the right sub tree of root.
3. Visit the root.

* Postorder Traversal is also known as LRN Traversal.

Example:



Therefore, the postorder traversal of the above tree will be: 0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7.

2. (b) Explain the rotations in AVL Tree when the element has to be inserted?

Ans: → Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. The new node is added to AVL tree as the leaf node.

→ However it may lead to the violation in AVL tree property and therefore the tree may need balancing.

→ The tree can be balanced by applying rotations.

→ Rotation is required only if, the balance-factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.

Depending upon the type of insertion, the rotations are categorized into four categories :-

1. LL ROTATION :

The new node is inserted into the left sub-tree of left-subtree of critical node.

2. RR ROTATION :

The new node is inserted to the right sub-tree of the right sub-tree of the critical node.

3. LR ROTATION :

The new node is inserted to the right sub-tree of the left-sub-tree of the critical node.

4. RL ROTATION :

The new node is inserted to the left-sub tree of the right sub-tree of the critical node.

Example: Construct an AVL tree by inserting the following elements in the given order.

63, 9, 19, 27, 18, 108, 99, 81.

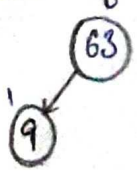
→ At each step, we must calculate the balance factor for every node, if it is found to be more than 2 or less than -2, then we need a rotation to rebalance the tree.

→ The type of rotation will be estimated by the location of the inserted element with respect to the critical node.

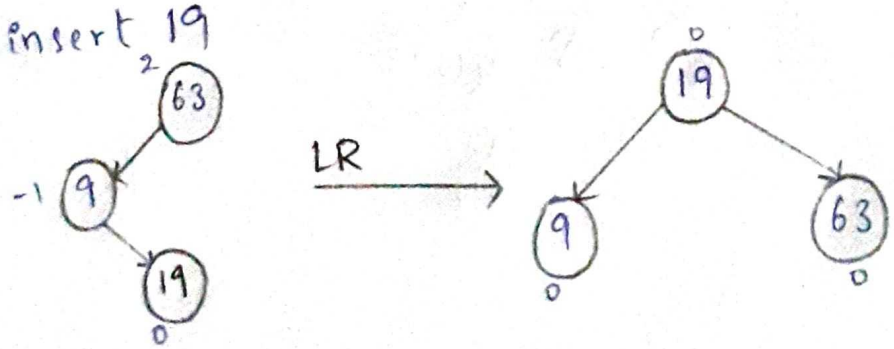
→ All the elements are inserted in order to maintain the order of binary search tree.

Step 1: insert 63

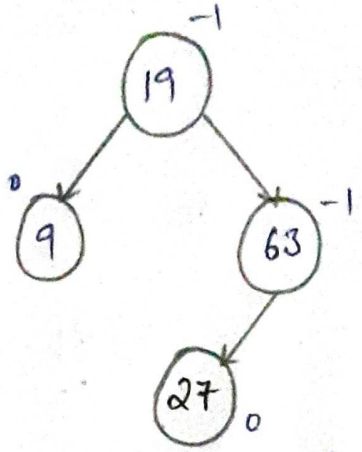
Step 2: insert 9



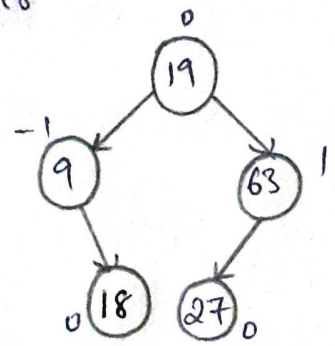
Step 3: insert 19



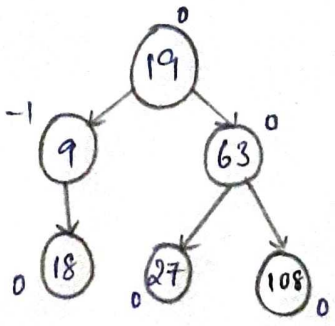
Step 4: Insert 27



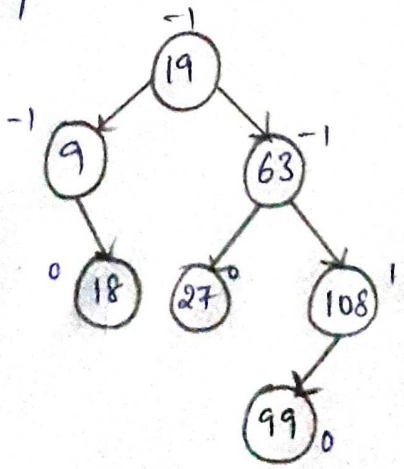
Step 5: Insert 18



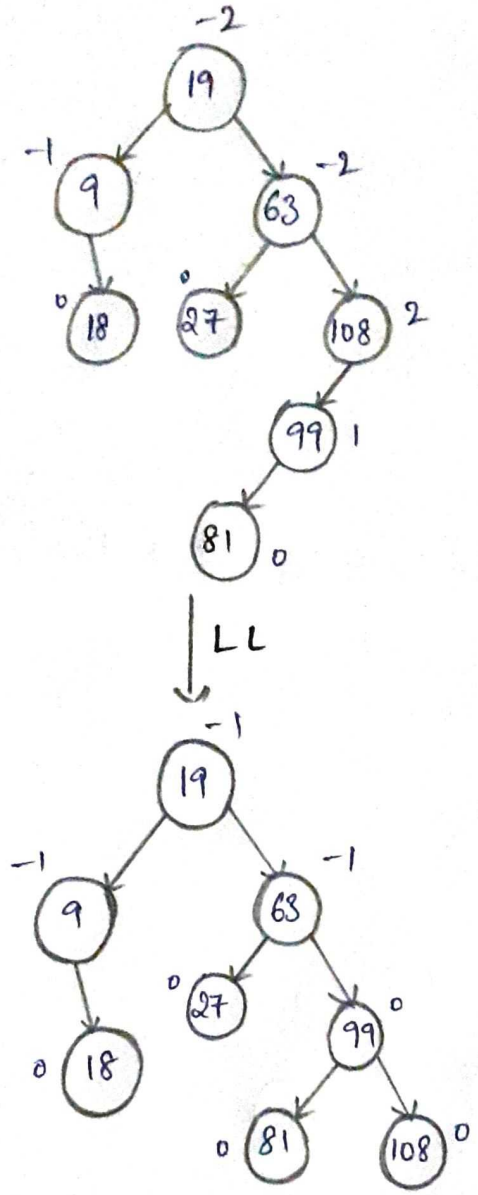
Step 6: Insert 108



Step 7: Insert 99



Step 8: Insert 81.



3

(a) Explain the following with neat Example?

(i) Binary Tree

(ii) Full Binary Tree.

(i) Explain Binary Tree:

Ans: A Binary Tree is a hierarchial data structure in which

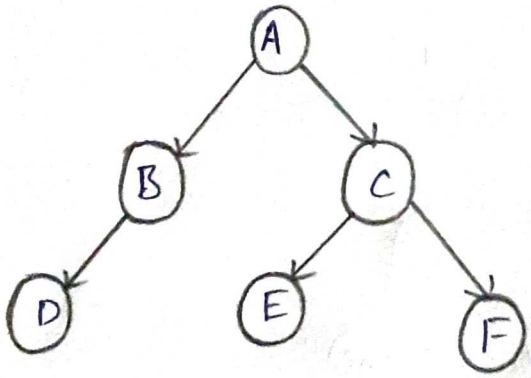
each node has at most two children generally referred as left child and right child.

Each node contains three components:

1. Pointer to left subtree
2. Pointer to right subtree
3. Data element.

The Topmost node in the tree is called the root.

Example of Binary Tree:

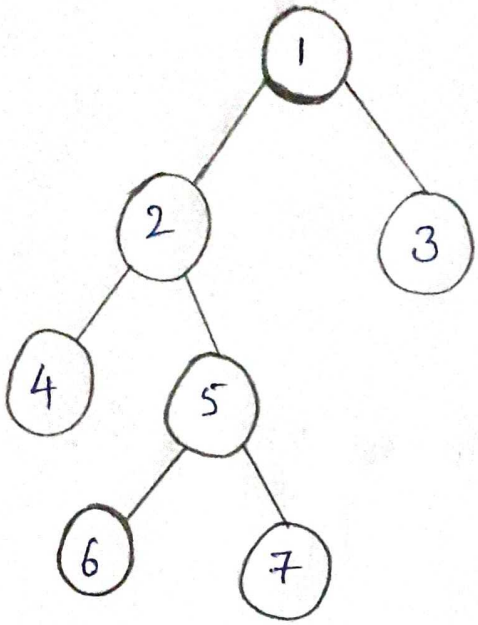


(ii) Full Binary Tree:

A Binary Tree is a full binary tree if every node has 0 or 2 children.

We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

Example for Full Binary Tree:



— Full Binary Tree —

3 (b) Construct a binary search tree whose elements are inserted in the following order?

50, 72, 96, 94, 107, 26, 12, 11, 9, 2, 10, 25, 51, 16, 17, 95.

Ans: In a Binary search tree (BST), each node contains-

→ Only smaller values in its left sub tree

→ Only larger values in its right sub tree.

→ We always consider the first element as the root node.

→ Consider the given elements and insert them in the BST one by one.

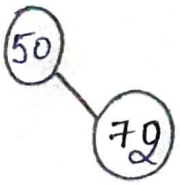
The following order is:

50, 72, 96, 94, 107, 26, 12, 11, 9, 2, 10, 25, 51, 16, 17, 95.

Insert 50-

Insert 72

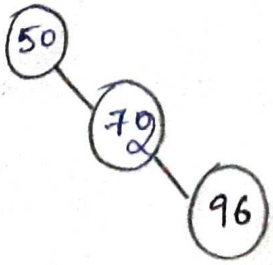
→ 70 > 50, so insert 70 to the right of 50.



Insert 96:

→ As 96 > 50, so insert 96 to the right of 50.

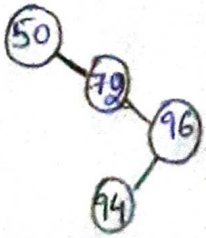
→ As 96 > 70, so insert 96 to the right of 70.



Insert 94:

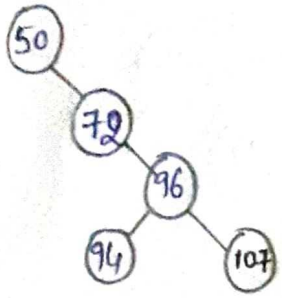
→ As 94 > (50, 70), so insert 94 to right of 70.

→ As 94 < 96, so insert 94 to the left of 96.



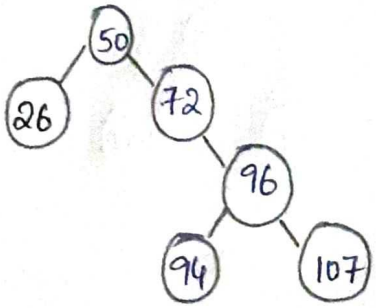
Insert 107:

→ As 107 > (50, 70, 96), so insert 107 to the right of 96:

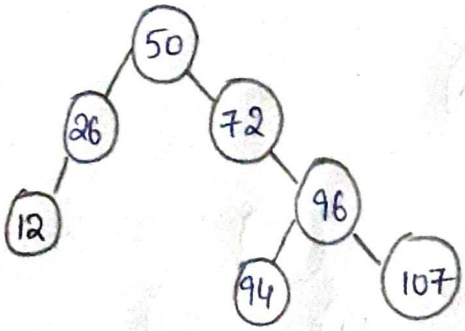


Insert 26:

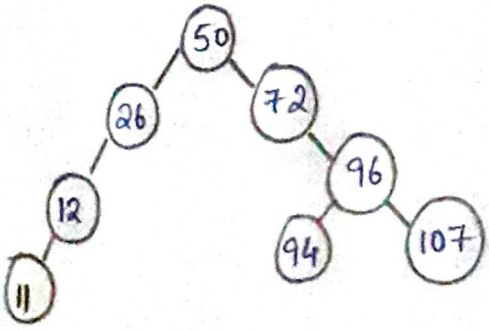
→ As 26 < 50, insert 26 to the left of 50.



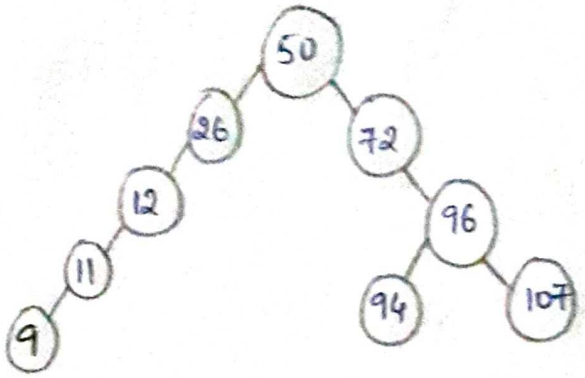
Insert 12:



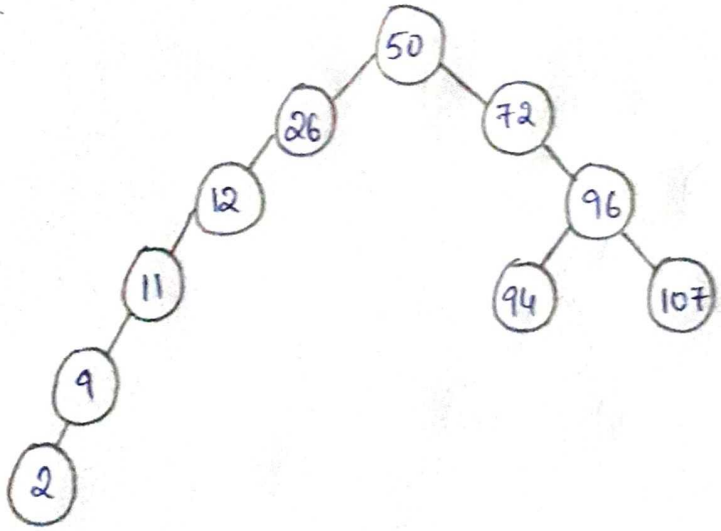
Insert 11:



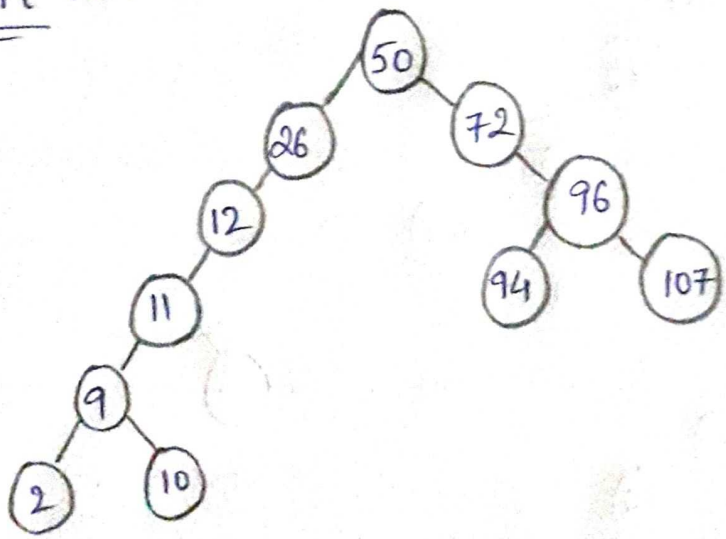
Insert 9:



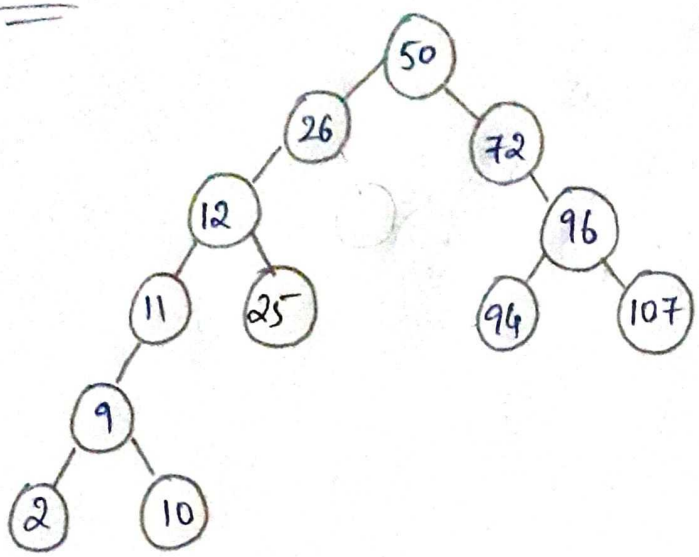
Insert 2:



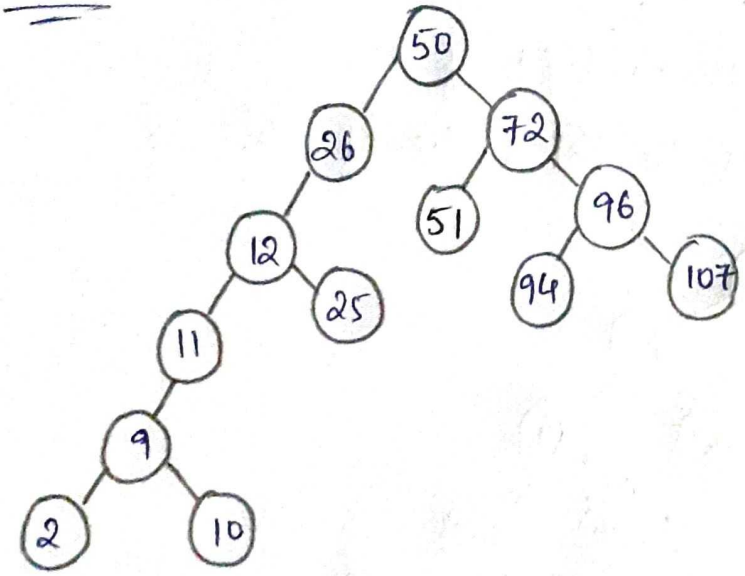
Insert 10:



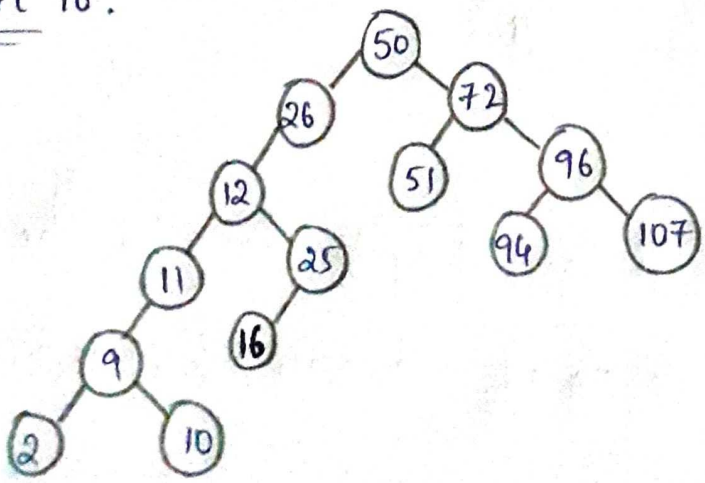
Insert 25:



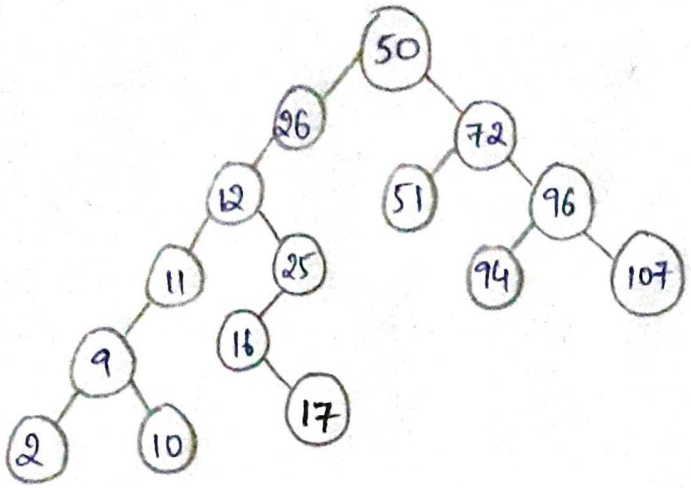
Insert 51:



Insert 16:



Insert 17:

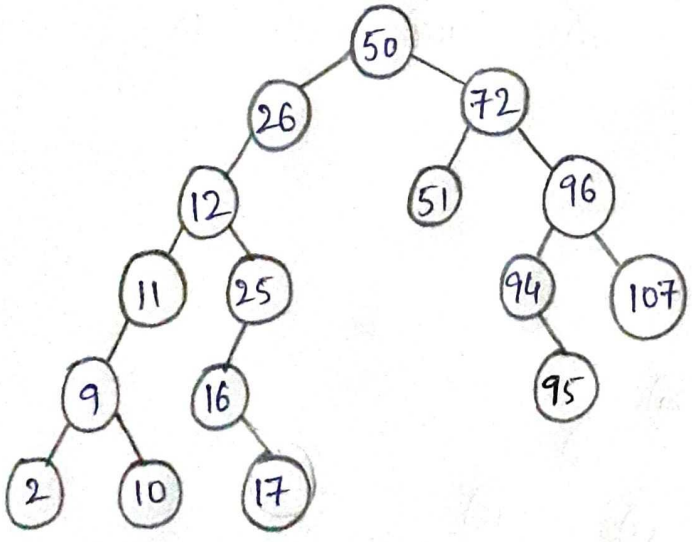


Insert 95:

→ As 95 > (50, 72), it is inserted to right of 72.

→ As 95 < 96, insert 95 to left of 96.

→ As 95 > 94, insert 95 to the right of 94.



This is the Required Binary Search Tree for the Order

50, 72, 96, 94, 107, 26, 12, 11, 9, 2, 10, 25, 51, 16, 17, 95.

UNIT-5

1

a) Differentiate between DFS and BFS ?

Ans:

SN	B F S	D F S
1.	BFS stands for Breadth First Search.	DFS stands for Depth first Search.
2.	BFS uses Queue data structure for finding the shortest path.	DFS uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex a source.
4.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from the source.
5.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems.
6.	Time complexity is O(V+E). V = vertices E = EDGES	Time Complexity of DFS also O(V+E) when adjacent list is used.

(b) **What is minimum spanning tree? Discuss with example?**

Ans =

→ Minimum Spanning Tree (MST) is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight.

→ To derive an MST, prim's algorithm or Kruskal's algo-rithm can be used.

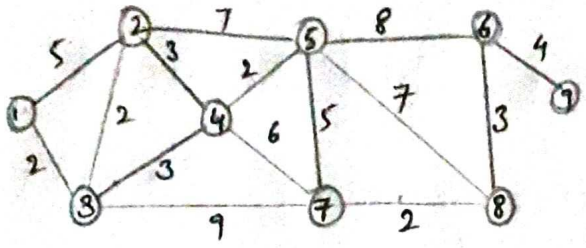
→ One graph may have more than one spanning tree.

→ If there are 'n' number of vertices, the spanning tree should have 'n-1' number of edges.

→ In this context, if each edge of the graph is associ-ated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.

→ Moreover, if there exist any duplicate weighted edge, the graph may have multiple min. spanning tree.

Example:



In the Above graph, we have shown a spanning tree though it's not the minimum spanning tree.

The cost of this spanning tree is $(5+7+3+3+5+8+3+4) = 38$.

(c) Explain Prim's algorithm and trace with an example?

Ans: → Prim's Algorithm is used for finding the Minimum Spanning Tree (MST) of a given graph.

→ To Apply Prim's Algorithm, the given graph must be weighted, connected and undirected.

The implementation of Prim's algorithm is explained in the following steps :-

Step 01:

→ Randomly choose any vertex.

→ The vertex connecting to the edge having least

weight is usually selected.

Step 2:

→ Find All the edges that connect the tree to new vertices.

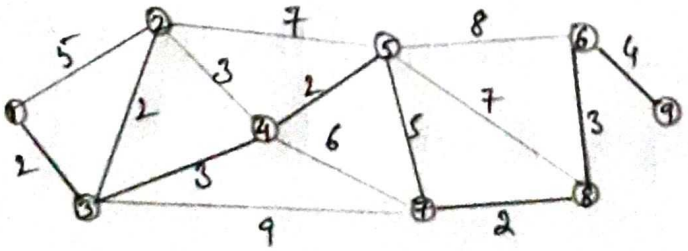
→ Find the least weight edge among those edges and include it in the existing tree.

→ If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step 3:

→ Keep repeating step-2 until all the vertices are included and Minimum spanning tree is obtained.

Example:



→ Let us start from vertex 1.

→ Vertex 3 is connected to vertex 1 with minimum edge cost hence edge (1,2) is added to the spinning tree.

→ Next, edge (2,3) is considered as this is the minimum among other edges which are remaining.

→ In next step, we selected edge (3,4).

→ In similar way, edges (4,5), (5,7), (7,8), (8,6), (6,9) are selected.

→ As All the vertices are visited, now the algorithm stops.

Therefore, The cost of the spinning tree is

$(2+2+3+2+5+2+3+4) = 23.$

There is No more spanning tree in this graph with cost less than 23.

---

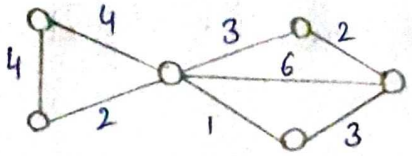
2 (a) Explain in detail about Dijkstra's algorithm for finding shortest path with an Example?

Ans: → Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

→ It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

Example of Dijkstra's Algorithm:

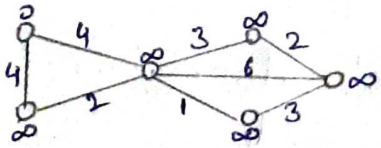
Start with a weighted graph:



Step:1

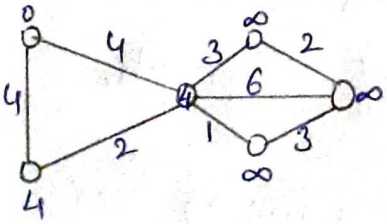
Step 2:

Choose a starting vertex and assign infinity path values to all other devices.



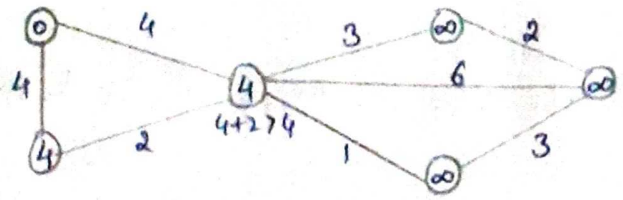
Step 3:

Go to each vertex and update its path length.



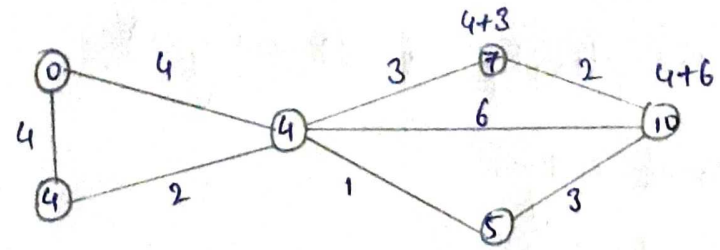
Step 4:

If path length of the adjacent vertex is lesser than new path length, dont update it.



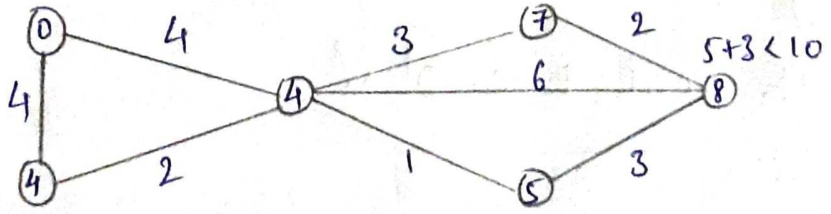
Step 5:

Avoid updating path lengths of already visited vertices.



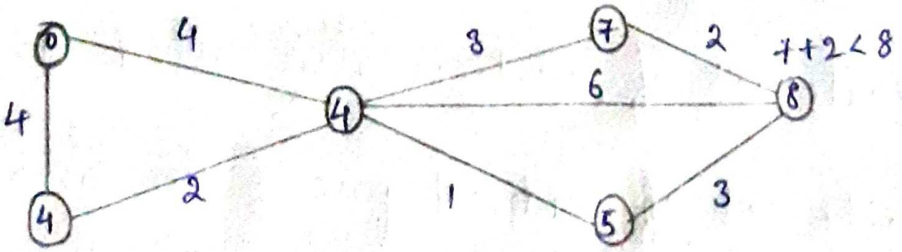
Step 6:

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7.



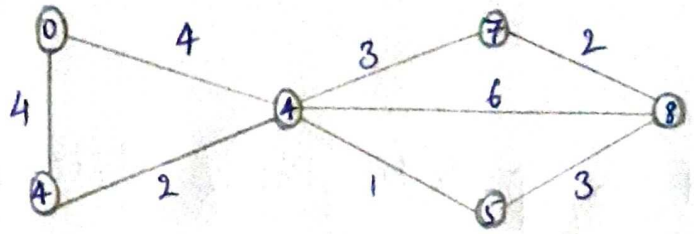
Step 7:

Notice how the rightmost vertex has it's path length updated twice.



Step 8:

Repeat until all the vertices have been visited.



→ The time complexity is $O(E \log V)$.

E → No. of Edges

V → No. of Vertices.

→ Space Complexity is $O(V)$.

(b) Define graph and explain the representation of Graphs with neat Examples?

Ans:

GRAPH:

A graph is a data structure that consists of the following two components:

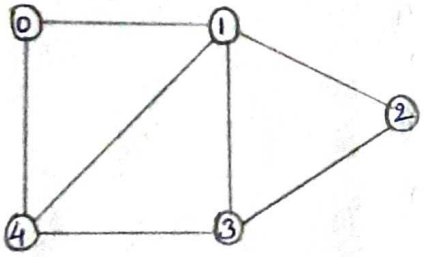
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u,v) called as Edge.

→ Graphs are used to represent many real-life applications: Graphs are used to represent networkes.

→ Graphs are also used in social networks like linkedin, Facebook.

Example:

Following is an example of an undirected graph with 5 vertices.



There are mainly two most Commonly used representations of a graph:

1. Adjacency Matrix.
2. Adjacency list.

ADJACENCY MATRIX:

→ Adjacency Matrix is a 2D array of size V x V, where V is the number of vertices in a graph.

Let the 2D Array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.

→ Adjacency matrix for undirected graph is always symmetric.

→ If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The Adjacency Matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

ADJACENCY LIST:

→ An array of lists is used. The size of the array is equal to the number of vertices.

→ Let an array be an array[].

→ An entry array[i] represents the list of vertices adjacent to the ith vertex.

→ This representation can also be used to represent a weighted

graph.

→ The weights of edges can be represented as list of pairs.

Following is the Adjacency list representation of the above graph.