EXPERIMENT - 10

Data Structures

Aim

Implement insertion, deletion and display (inorder, preorder and postorder) on binary search tree.

Syeda Reeha Quasar
14114902719
3C7

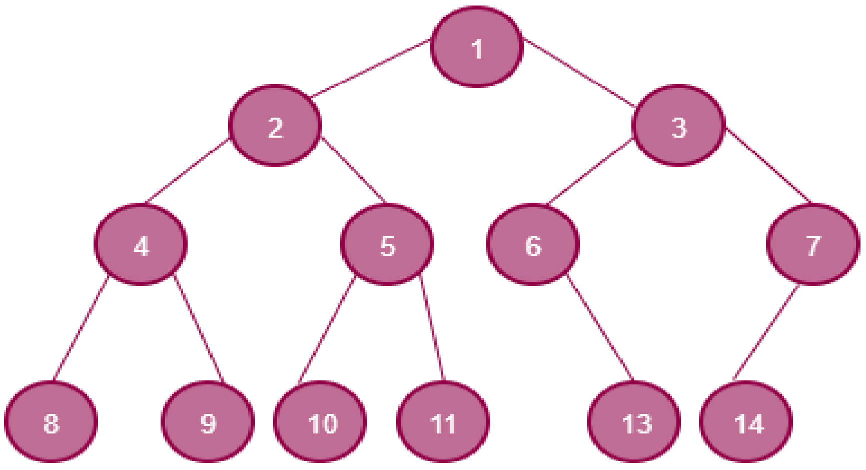
Name - Syeda Reeha Quasar **Roll no. – 14114802719** **Group – C7**

EXPERIMENT – 10

AIM: Implement insertion, deletion and display (inorder, preorder and postorder) on binary search tree.

THEORY

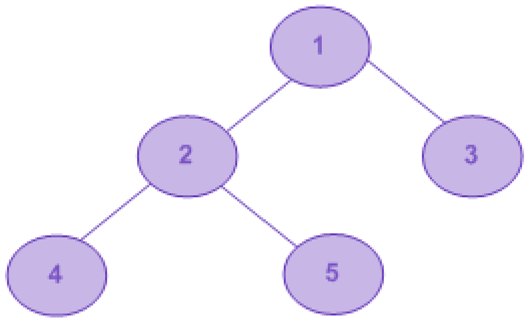
A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree
Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1
Breadth First or Level Order Traversal : 1 2 3 4 5
Please see this post for Breadth First Traversal.

Inorder Traversal:

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal:

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.
Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal:

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Uses of Postorder

Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation to for the usage of postfix expression.
Example: Postorder traversal for the above given figure is 4 5 2 3 1.

Insertion:

Inserting in a binary tree.

Algorithm:

Step 1: Create a function to insert the given node and pass two arguments to it, **the root node** and the **data to be inserted**.
Step 2: Define a temporary node to store the popped out nodes from the queue for search purpose.
Step 3: Define a queue data structure to store the nodes of the binary tree.
Step 4: Push the root node inside the queue data structure.
Step 5: Start a while loop and check for the condition that whether the queue is empty or not, if not empty then go to Step 6, else go to Step 9.
Step 6: **P**op out the first node from the queue and store it inside the temporary node.
Step 7: Check, for the current pooped out node, in the binary tree, inside the while loop, if its left child(in binary tree) is null then call the **memory allocation method** for the new node, with its left and right child set as null and then insert the given node to its new position else push its left child in the queue data structure.

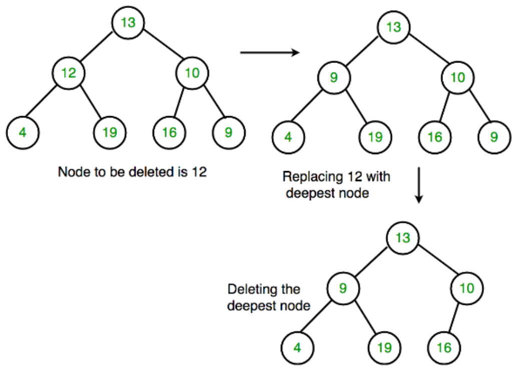
Step 8: Similarly repeat Step 7 for the right child of the current node in the binary tree.
Step 9: End of while loop.
Step 10: End of the function.

Deletion:

Deletion of a node from binary tree. This can be done after searching for the element to be deleted.

Algorithm

1. Starting at root, find the deepest and rightmost node in binary tree and node which we want to delete.
2. Replace the deepest rightmost node's data with node to be deleted.
3. Then delete the deepest rightmost node.



Source code:

```
#include <stdio.h>

#include <stdlib.h>

#include <queue>


using namespace std;


// A binary tree node containing data, left pointer and right pointer

struct bNode {

        int data; // data

        struct bNode* left; // poiter to the left

        struct bNode* right; // pointer to the right

};
```

```
// newNode creator for tree nodes

struct bNode* bNewNode(int data) {

        struct bNode* bNode = (struct bNode*) malloc(sizeof(struct bNode)); // allocating memory

        bNode -> data = data; // data

        bNode -> left = NULL; // poiter to the left

        bNode -> right = NULL; // pointer to the right


        return (bNode);

}


void insert(bNode ** tree, int val){

  bNode *root = NULL; // temp for creation if no root present


  if (!(*tree)) { // checking if tree is empty or not

    root = (bNode *) malloc (sizeof (bNode) ); // allocating memory

    root -> left = root -> right = NULL;

    root -> data = val;

    *tree = root;

    return;

  }


  if (val < (*tree) -> data) {

    insert(&(*tree) -> left, val); // insertion at left

  }

  else if (val > ((*tree) -> data)) {

    insert(&(*tree) -> right, val); // insertion at right

  }

}


// function to delete the entire tree
```

```
void deltree(bNode * tree){
   if (tree) { // checking if tree exists
      deltree(tree -> left);
      deltree(tree -> right);
      delete(tree);
   }
}

void deletDeepest(bNode* root, bNode* d_node){
   queue<struct bNode*> q;
   q.push(root);

   // Do level order traversal until last node
   struct bNode* temp;
   while (!q.empty()) {
      temp = q.front();
      q.pop();
      if (temp == d_node) {
         temp = NULL;
         delete (d_node);
         return;
      }
      if (temp -> right) {
         if (temp -> right == d_node) {
            temp -> right = NULL;
            delete (d_node);
            return;
         }
         else
            q.push(temp -> right);
```

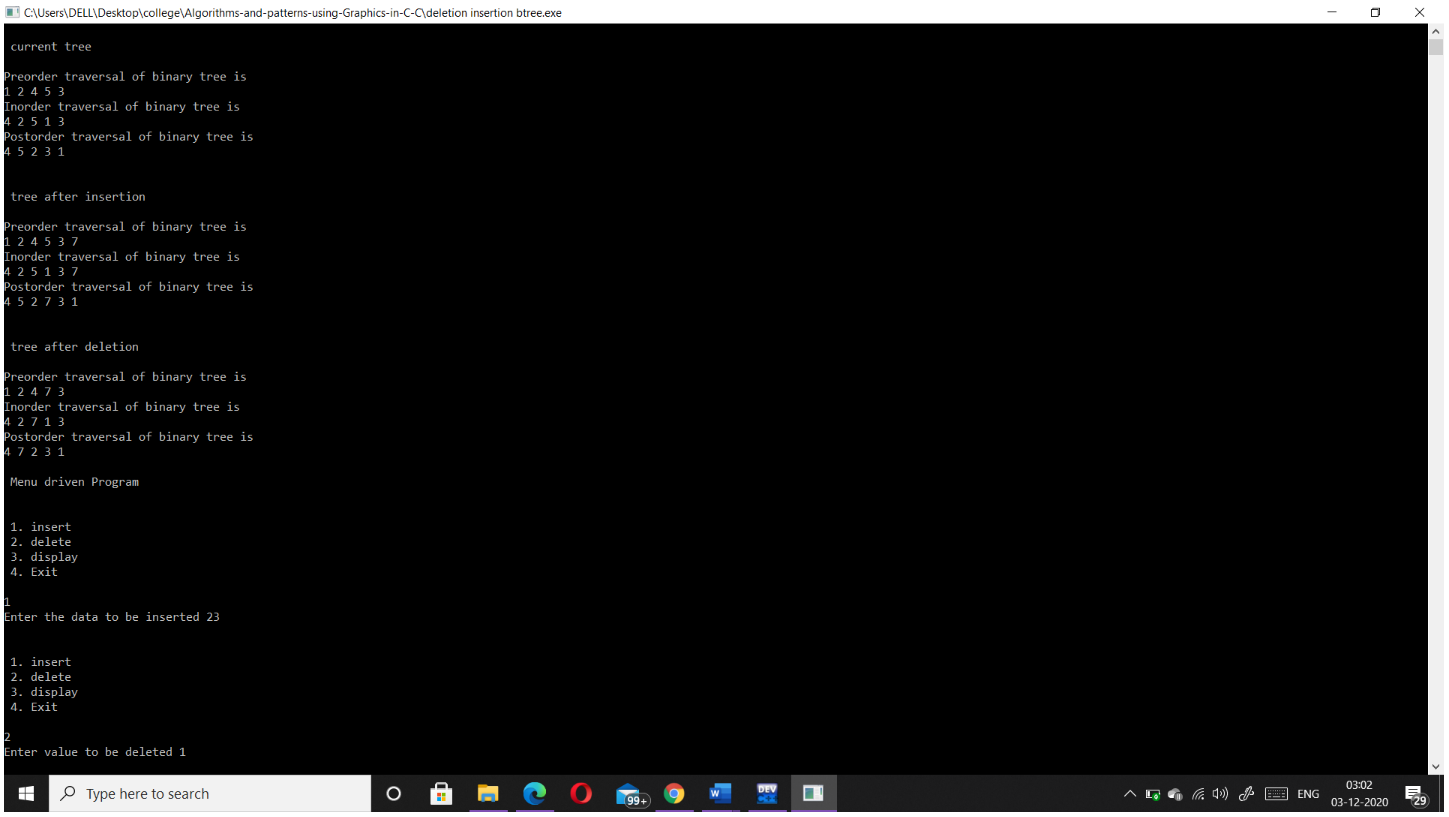
```
        }

        if (temp -> left) {
            if (temp -> left == d_node) {
                temp -> left = NULL;
                delete (d_node);
                return;
            }
            else
                q.push(temp -> left);
        }
    }
}
//function for node deletion
bNode* deletion(bNode* root, int key)
{
    if (root == NULL) // checking if tree is empty or not
        return NULL;

    if (root -> left == NULL && root->right == NULL) { // if only root exists and root is not equal to key
        if (root -> data == key)
            return NULL;
        else
            return root;
    }

    queue <struct bNode*> q;
    q.push(root);

    struct bNode* temp;
    struct bNode* key_node = NULL;

    // Do level order traversal to find deepest
    // node(temp) and node to be deleted (key_node)
    while (!q.empty()) {
        temp = q.front();
        q.pop();

        if (temp -> data == key)
            key_node = temp;

        if (temp -> left)
            q.push(temp -> left);

        if (temp -> right)
            q.push(temp -> right);
    }

    if (key_node != NULL) {
        int x = temp -> data;
        deletDeepest(root, temp);
        key_node -> data = x;
    }
    return root;
}

// function for post order traversal
void printPostorder(bNode* bNode) {
            if (bNode == NULL) //empty tree
                        return;
        printPostorder(bNode -> left); // first recur on left subtree

        printPostorder(bNode -> right); // then recur on right subtree

        printf("%d ", bNode -> data); //printing each node data
}

// function for inorder traversal
void printInorder(bNode* bNode) {
        if (bNode == NULL)  // tree empty
                return;

        printInorder(bNode -> left); // first recur on left child

        printf("%d ", bNode -> data); // then print the data of nod

        printInorder(bNode -> right); // now recur on right child
}

// preorder traversal function
void printPreorder(bNode* bNode) {
        if (bNode == NULL) // empty tree case
                return;

        printf("%d ", bNode -> data); // first print data of node

        printPreorder(bNode -> left); // then recur on left sutree

        printPreorder(bNode -> right); // now recur on right subtree
```

```
}

void display(bNode* root){
        if (root == NULL) {
                return;
        }
   printf("\nPreorder traversal of binary tree is \n");
        printPreorder(root);

        printf("\nInorder traversal of binary tree is \n");
        printInorder(root);

        printf("\nPostorder traversal of binary tree is \n");
        printPostorder(root);
        return;
}



int main() {
        struct bNode *root = bNewNode(1);
        root -> left = bNewNode(2);
        root -> right   = bNewNode(3);
        root -> left -> left = bNewNode(4);
        root -> left -> right = bNewNode(5);

        printf("\n current tree\n");
        display(root);

   insert(&root, 7);
```

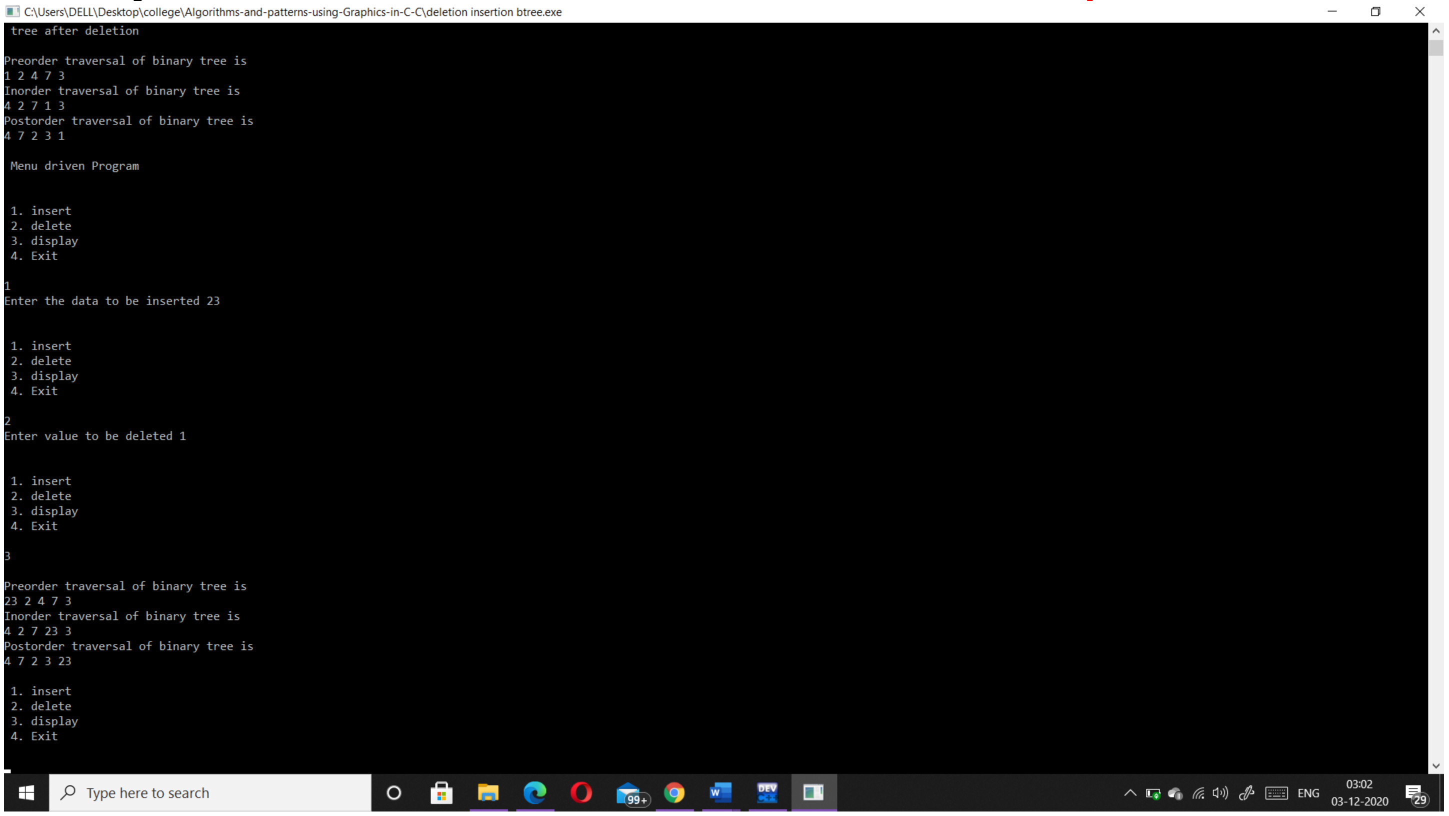
```
printf("\n\n\n tree after insertion\n");
display(root);


deletion(root, 5);


printf("\n\n\n tree after deletion\n");
display(root);


//user interaction menu driven
printf("\n\n Menu driven Program \n");
int s = 3, val;
while (s != 4) {
    printf("\n\n 1. insert \n 2. delete \n 3. display \n 4. Exit \n\n");
    scanf("%d", &s);
    switch (s) {
        case 1:
            printf("Enter the data to be inserted");
            scanf ("%d", &val);
            insert(&root, val);
                break;
            case 2:
                printf("Enter value to be deleted");
                scanf ("%d", &val);
                deletion(root, val);
                break;
            case 3:
                display(root);
                break;
            case 4:
                break;
```

```
                        default:

                                printf("Entered a wrong key");

                                break;

                }

        }


        return 0;

}
```

OUTPUT



```
C:\Users\DELL\Desktop\college\Algorithms-and-patterns-using-Graphics-in-C-C\deletion insertion btree.exe

 current tree

Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1


 tree after insertion

Preorder traversal of binary tree is
1 2 4 5 3 7
Inorder traversal of binary tree is
4 2 5 1 3 7
Postorder traversal of binary tree is
4 5 2 7 3 1


 tree after deletion

Preorder traversal of binary tree is
1 2 4 7 3
Inorder traversal of binary tree is
4 2 7 1 3
Postorder traversal of binary tree is
4 7 2 3 1

 Menu driven Program


 1. insert
 2. delete
 3. display
 4. Exit

1
Enter the data to be inserted 23


 1. insert
 2. delete
 3. display
 4. Exit

2
Enter value to be deleted 1
```


C:\Users\DELL\Desktop\college\Algorithms-and-patterns-using-Graphics-in-C-C\deletion insertion btree.exe

```
 tree after deletion

Preorder traversal of binary tree is
1 2 4 7 3
Inorder traversal of binary tree is
4 2 7 1 3
Postorder traversal of binary tree is
4 7 2 3 1

 Menu driven Program


 1. insert
 2. delete
 3. display
 4. Exit


1
Enter the data to be inserted 23


 1. insert
 2. delete
 3. display
 4. Exit


2
Enter value to be deleted 1


 1. insert
 2. delete
 3. display
 4. Exit


3

Preorder traversal of binary tree is
23 2 4 7 3
Inorder traversal of binary tree is
4 2 7 23 3
Postorder traversal of binary tree is
4 7 2 3 23

 1. insert
 2. delete
 3. display
 4. Exit
```

VIVA VOICE

Q1. Give any application of BST.

Ans.

1. It is used to efficiently store data in sorted form in order to access and search stored elements quickly
2. They can be used to represent arithmetic expressions
3. BST used in Unix kernels for managing a set of virtual memory areas (VMAs).

Q2. Whether a binary search tree is balanced or not?

Ans.

A non-empty binary tree is height-balanced if:

1. Its left subtree is height-balanced.
2. Its right subtree is height-balanced.
3. The difference between heights of left & right subtree is not greater than 1.

Q3. Which traversal will generate a ascending order list of nodes in BST?

Ans.

Inorder traversal of BST prints it in ascending order. The only trick is to modify recursion termination condition in standard Inorder Tree Traversal.

Q4. what is a binary tree?

Ans.

A normal tree has no restrictions on the number of children each node can have. Binary trees, on the other hand, can have at most two children for each parent. Every node contains a 'left' reference, a 'right' reference, and a data element. The topmost node in the tree is called the root node. Nodes with children are parent nodes, and the child nodes contain references to their parents. A node with no children is called a leaf node. Thus, each node in a binary tree can have either 0, 1 or 2 children

Q5. What are different types of binary trees?

Ans.

There are three different types of binary trees that will be discussed in this lesson:

- Full binary tree: Every node other than leaf nodes has 2 child nodes.
- Complete binary tree: All levels are filled except possibly the last one, and all nodes are filled in as far left as possible.
- Perfect binary tree: All nodes have two children and all leaves are at the same level.

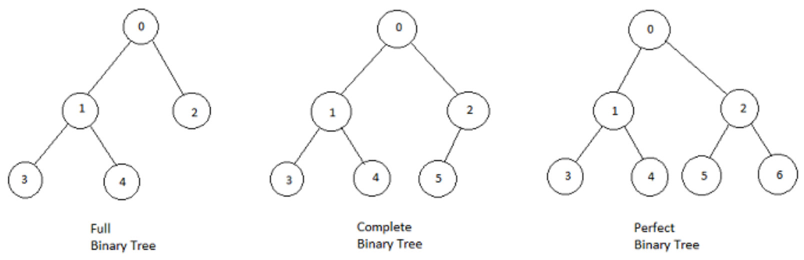


Figure 2: Types of Binary Trees

Q6. Which of the following is false about a binary search tree?

a) The left child is always lesser than its parent

b) The right child is always greater than its parent

c) The left and right sub-trees should also be binary search trees
d) In order sequence gives decreasing order of elements

Ans. Answer: d
Explanation: In order sequence of binary search trees will always give ascending order of elements. Remaining all are true regarding binary search trees.

Q7. What is the speciality about the inorder traversal of a binary search tree?
a) It traverses in a non increasing order
b) It traverses in an increasing order
c) It traverses in a random fashion
d) It traverses based on priority of the node

Ans. Answer: b
Explanation: As a binary search tree consists of elements lesser than the node to the left and the ones greater than the node to the right, an inorder traversal will give the elements in an increasing order.

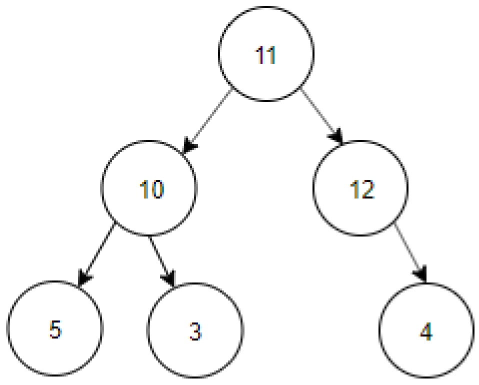
Q8.What does the following piece of code do?

```
public void func(Tree root)
{
        func(root.left());
        func(root.right());
        System.out.println(root.data());
}
```

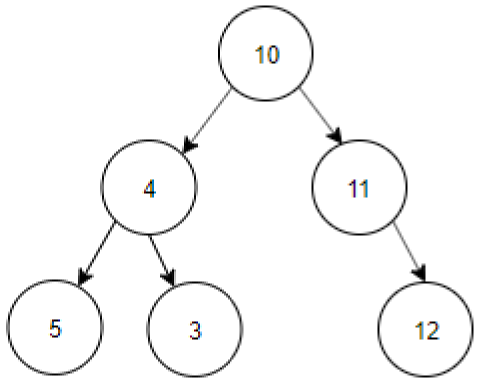
a) Preorder traversal
b) Inorder traversal
c) Postorder traversal
d) Level order traversal

Ans. Answer: c
Explanation: In a postorder traversal, first the left child is visited, then the right child and finally the parent.

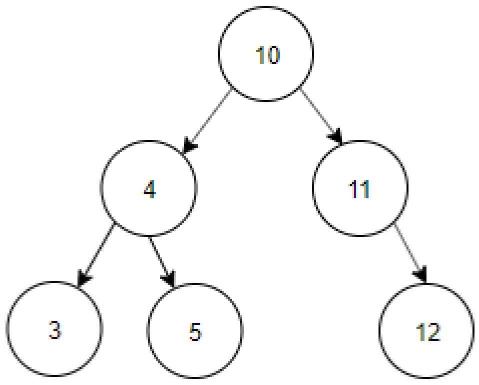
Q9. Construct a binary search tree with the below information.
The preorder traversal of a binary search tree 10, 4, 3, 5, 11, 12.



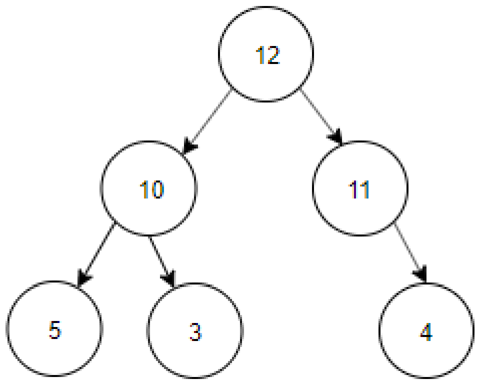
a)



b)



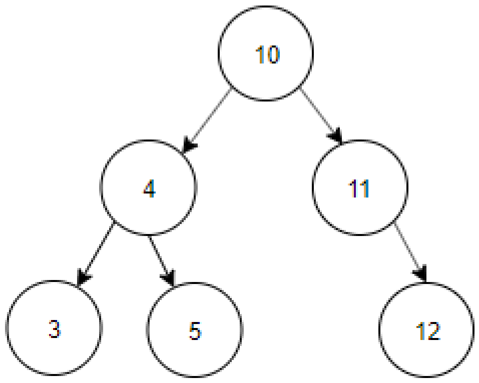
c)



d)

Ans. Answer: c

Explanation: Preorder Traversal is 10, 4, 3, 5, 11, 12. Inorder Traversal of Binary search tree is equal to ascending order of the nodes of the Tree. Inorder Traversal is 3, 4, 5, 10, 11, 12. The tree constructed using Preorder and Inorder traversal is



Q10. **What are the applications of binary trees?**

Ans.

- Binary Search Tree - Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
- Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.
- Binary Tries - Used in almost every high-bandwidth router for storing router-tables.
- Hash Trees - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
- Heaps - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers, and A* (path-finding algorithm used in AI applications, including robotics and video games). Also used in heap-sort.
- Huffman Coding Tree (Chip Uni) - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
- GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.
- Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expressions.
- Treap - Randomized data structure used in wireless networking and memory allocation.

- T-tree - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so.