Name	Sujal Dingankar
UID no.	DSE24100720
Experiment No.	5

AIM:	Write a program to construct a binary search tree, insert an element in BST, delete an element from BST and traversed it.
THEORY:	**Binary Search Tree (BST) Data Structure :** A Binary Search Tree (BST) is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. In a BST, each node contains a key, and the tree is organized in such a way that for any given node, the key in the left child is less than the key of the node, and the key in the right child is greater than the key of the node. This property allows for efficient searching, insertion, and deletion operations. A Binary Search Tree is commonly used in scenarios where ordered data needs to be stored dynamically, and operations such as searching, insertion, and deletion must be performed efficiently. **Operations on Binary Search Tree:** 1) Insertion Operation: • At the Root: If the tree is empty, the new node becomes the root of the tree. • At a Leaf Node: If the tree is not empty, the algorithm traverses the tree by comparing the key with the node's key. It moves to the left subtree if the key is smaller and to the right subtree if the key is larger. Once it reaches a leaf node (where the next child pointer is NULL), it inserts the new node in the appropriate position. 2) Deletion Operation: • Leaf Node: If the node to be deleted has no children, it can simply be removed. • Node with One Child: If the node has only one child (either left or right), the node is removed and its child is linked directly to its parent. • Node with Two Children: If the node has two children, it is replaced by its in-order successor (smallest node in the right subtree) or in-order predecessor (largest node in the left subtree). The in-order successor or predecessor is then removed using the appropriate deletion logic (since it will be a leaf or have only one child). 3) Traversal Operation: • In-Order Traversal: Traverses the tree in ascending order by first visiting the left subtree, then the current node, and finally the right subtree.

- Pre-Order Traversal: Visits the current node before its children, traversing the left subtree and then the right subtree.
- Post-Order Traversal: Traverses the left subtree, then the right subtree, and finally visits the current node.

4) Search Operation:

- Searches for a specific key by traversing the tree from the root. If the key is smaller than the current node's key, the algorithm moves to the left subtree; if larger, it moves to the right subtree. If the key is found, the search is successful. If the traversal reaches a leaf node (NULL), the key does not exist in the tree.

5) IsEmpty Operation:

- Checks if the tree is empty by verifying if the root is NULL. Returns true if the tree is empty, otherwise returns false.
- Key Properties of a Binary Search Tree:
- Binary Structure: Every node has at most two children.
- Ordering Property: For each node, all keys in its left subtree are smaller, and all keys in its right subtree are larger.
- Efficient Operations: BST provides efficient search, insertion, and deletion operations, which can be performed in O(h) time, where h is the height of the tree. In a balanced BST, the height is log(n), where n is the number of nodes.

Applications of Binary Search Tree:

- Efficient Searching: BSTs are widely used in scenarios where dynamic data needs to be searched efficiently. The tree structure allows logarithmic time complexity for search operations in balanced trees.
- Database Indexing: In database systems, BSTs are used to implement indexes, enabling fast access to records.
- Memory Management: Binary search trees are used in memory management systems, such as garbage collection algorithms.
- Symbol Table in Compilers: BSTs are often used to implement symbol tables in compilers, where variables, functions, and their scope need to be stored and retrieved quickly.
- Priority Queue and Scheduling Algorithms: Variants of BSTs (such as AVL trees and red-black trees) are used to implement priority queues, which are essential for task scheduling algorithms.
- Routing Algorithms: BSTs can be used in network routing algorithms for efficient route lookup based on IP address ranges.

ALGORITHM:	**1) Start with an empty BST:** **Initialize the root as NULL and prepare for user interaction.** **2) Insert a new node into the BST:** • **Prompt the user to enter an integer key.** • **If the root is NULL, create a new node and set it as the root.** • **If the root exists, recursively traverse the left subtree if the key is smaller, or the right subtree if the key is larger.** • **Insert the node at the correct position based on the comparison.** **3) Preorder traversal:** • **Visit the current node (root).** • **Traverse the left subtree in a preorder manner.** • **Traverse the right subtree in a preorder manner.** **4) Postorder traversal:** • **Traverse the left subtree in a postorder manner.** • **Traverse the right subtree in a postorder manner.** • **Visit the current node (root)** **5) Inorder traversal:** • **Traverse the left subtree in an inorder manner.** • **Visit the current node (root).** • **Traverse the right subtree in an inorder manner.** **6) Search for a specific node:** • **Prompt the user for the value to search.** • **Starting from the root, recursively traverse the tree based on the value.** • **If the value is smaller than the current node, traverse the left subtree, otherwise, traverse the right subtree.**

- **Return the node if found, otherwise return NULL.**

7) Delete a node from the BST:

- **Prompt the user for the value to delete.**
- **Find the node to delete by traversing the tree recursively.**
- **If the node has no children, remove it directly.**
- **If the node has only one child, replace it with its child.**
- **If the node has two children, find its in-order successor or predecessor, replace the node's value with the successor/predecessor, and recursively delete the successor/predecessor node.**

8) In-order predecessor (optional helper function):

- **Find the largest node in the left subtree (predecessor).**
- **Traverse to the rightmost node of the left subtree to find it.**

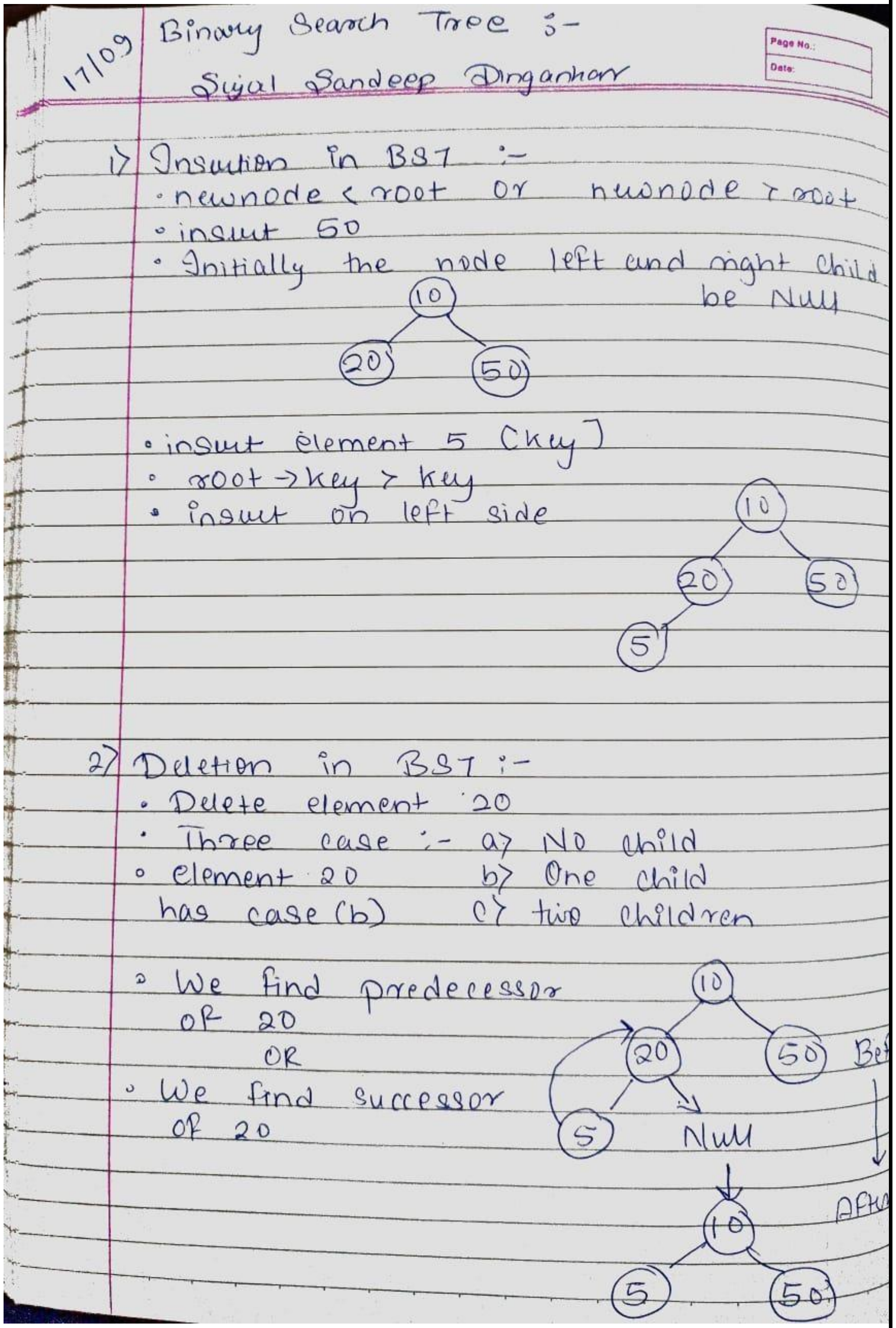
9) In-order successor (optional helper function):

- **Find the smallest node in the right subtree (successor).**
- **Traverse to the leftmost node of the right subtree to find it.**

10) Exit:

- **Continue accepting user inputs for the above operations until the user chooses to exit the program.**

PROBLEM SOLVING:



17/09 Binary Search Tree :-

Sijal Sandeep Dingankar

Page No.:

Date:

1) Insertion in BST :-

- newnode < root or newnode > root
- insert 50
- Initially the node left and right child be Null

10
20
50

- insert element 5 (key)
- root → key > key
- insert on left side

10
20
50
5

2) Deletion in BST :-

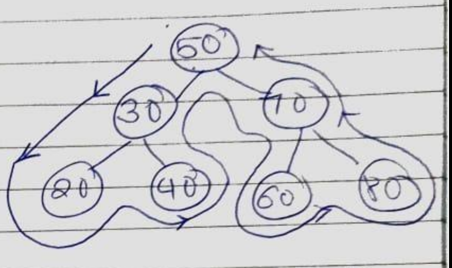
- Delete element 20
- Three case :- a) No child
- element 20 b) One child
has case (b) c) two children

- We find predecessor of 20
OR
- We find successor of 20

3) Traversing of BST :-

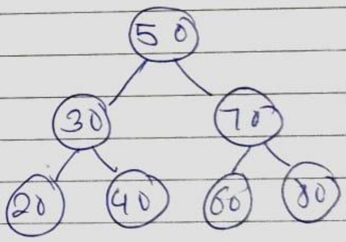
a) Preorder :- ~~Et~~ [Root - Left - Right]

→ 50, 30, 20, 40, 70, 60, 80



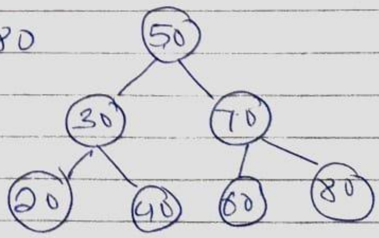
b) Postorder :- [~~Root~~ left - Right - Root]

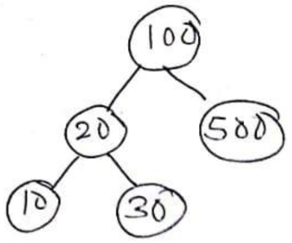
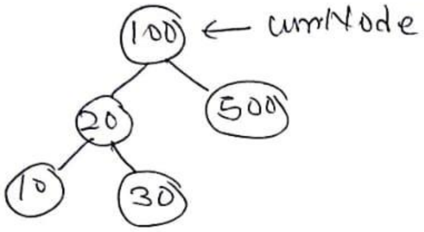
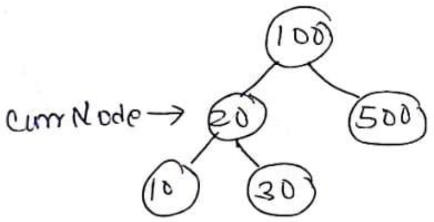
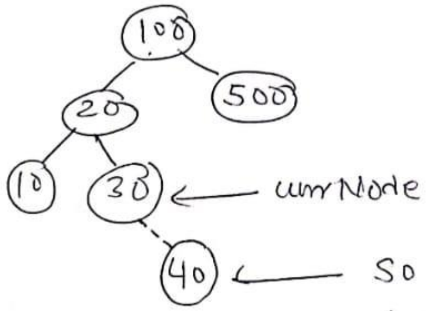
→ 20, 40, 30, 60, 80, 70, 50

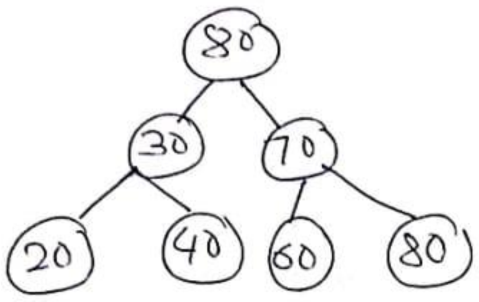
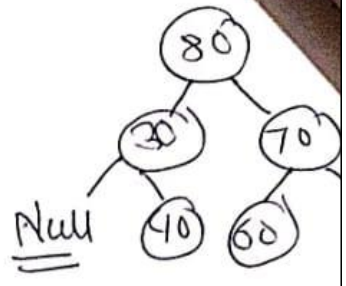
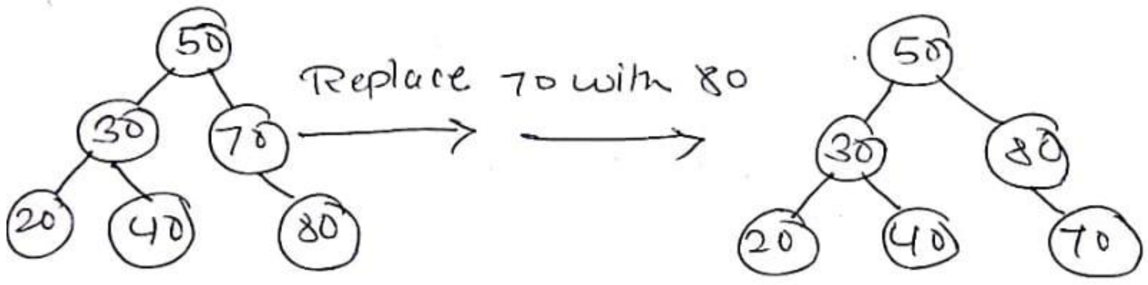
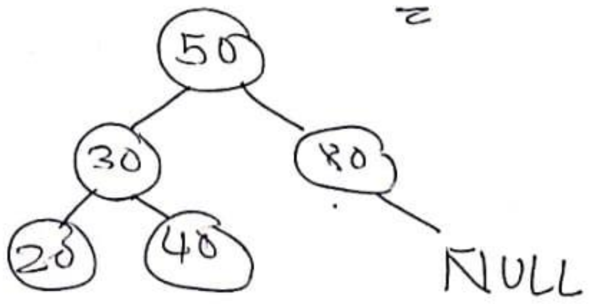


c) Inorder :- [left - Root - Right]

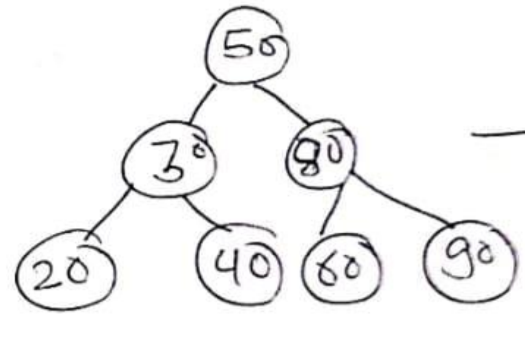
→ 20, 30, 40, 50, 70, 60, 80



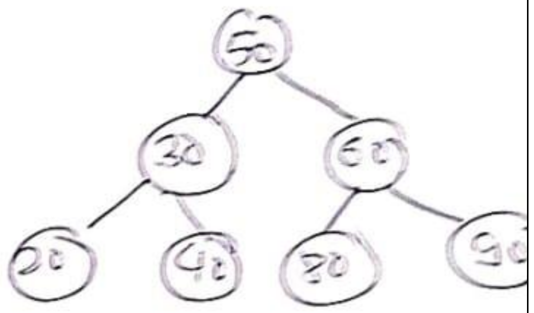
Insertion in BST Manual Explanation :-	 key = 40 (node to be inserted) Step 1 Comparing key with root node  root->key > key comparing key with left child root node  since 20 > 40 [key] comparing key with the right child of 20 30 < 40 [key]  So 40 will be inserted here or right side of 30

Deletion in BST Manual Explanation:-	Deletion in BST :- case 1 - Delete a leaf Node in BST.  Delet Node 20  Assign Node to case 2 - Delete Node with single Child in BST.  now delete leaf Node 70 

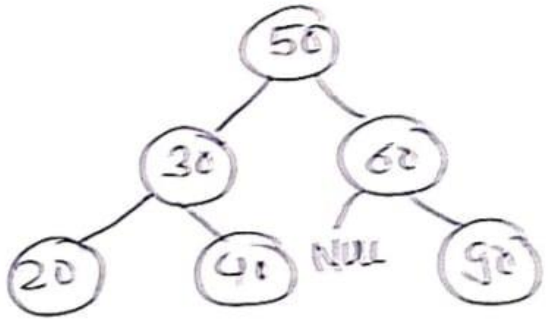
case 3 - Delete Node with Both children in BST



Delete Node 80 and replace with Node 60



After replacing delete Node 80



PROGRAM :-

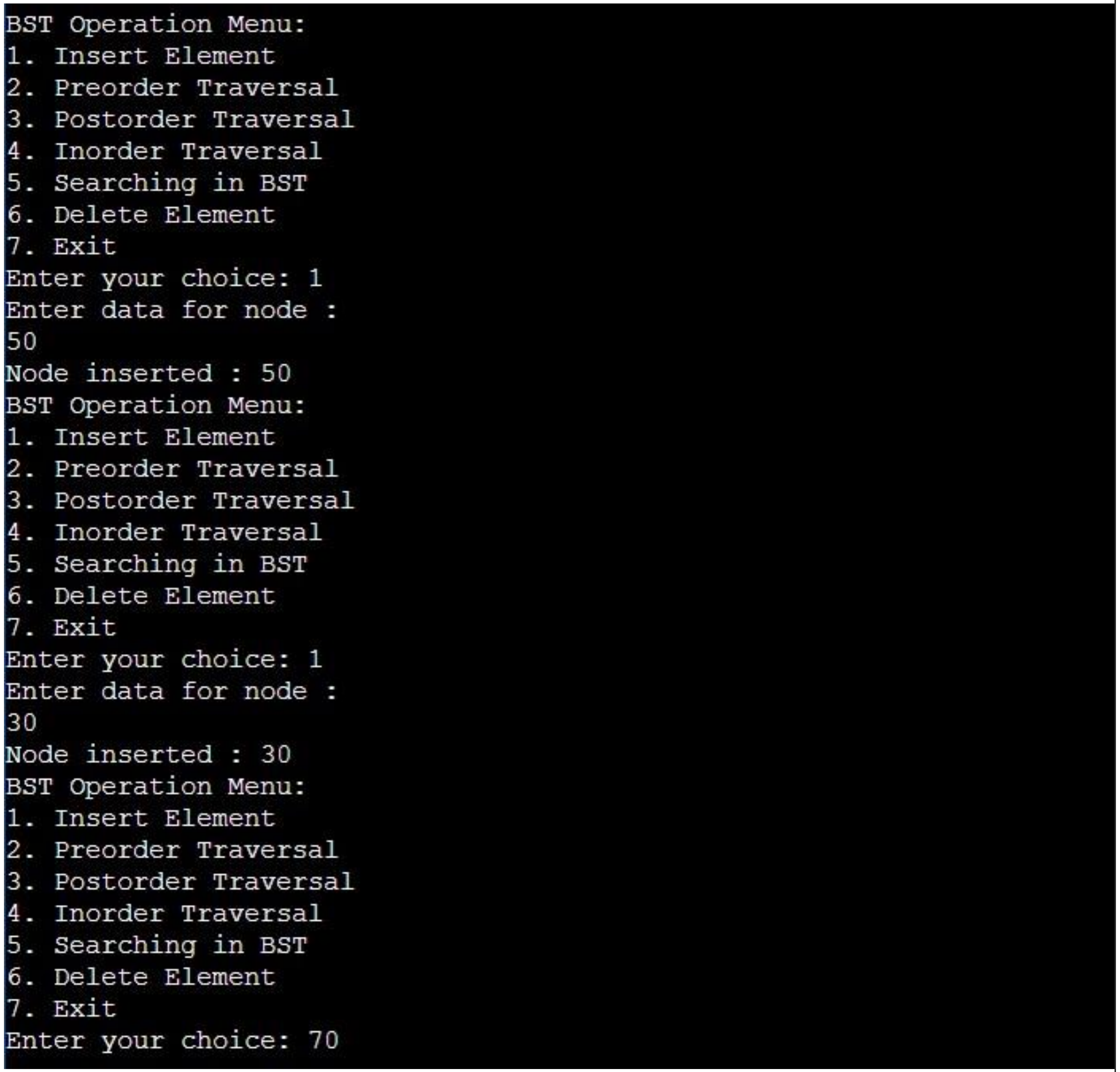
```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int key;
    struct Node *left;
    struct Node *right;
};

struct Node *createNode(int data){
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = data;
    newNode->left = NULL;
    newNode->right = NULL;

    return newNode;
}

struct Node *insertBST(struct Node* root,int key){
    if(root == NULL){
        return createNode(key);
    }

    if(key < root->key){
        root->left = insertBST(root->left,key);
    }else{
        root->right = insertBST(root->right,key);
    }

    return root;
}

struct Node *searchNode(struct Node *root,int key){
    if(root == NULL || root->key == key){
        return root;
    }
    if(key < root->key){
        searchNode(root->left,key);
    }else{
        searchNode(root->right,key);
    }
}

// Function to find the in-order predecessor (largest node in the left subtree)
struct Node *predecessor(struct Node *root){
    struct Node *curr = root->left;
    while(!curr && !curr->right){
        curr = curr->right;
    }

    return curr;
}

// Function to find the in-order successor (smallest node in the right subtree)
struct Node *successor(struct Node *root){
    struct Node *curr = root->right;
    while(!curr && !curr->left){
        curr = curr->left;
    }

    return curr;
}

struct Node *DeleteNode(struct Node* root, int key){

    //base condition...
    if(root == NULL){
        return root;
    }

    //search in subtree..
    if(key < root->key){
        root->left = DeleteNode(root->left,key);
```

	```
	    }else if(key > root->key){
	        root->right = DeleteNode(root->right,key);
	    }else{
	        //case when root has no child or only right child...
	        if(root->left == NULL){
	            struct Node *temp = root->right;
	            free(root);
	            return temp;
	        }

	        //case when root has no child or only left child.
	        if(root->right == NULL){
	            struct Node *temp = root->left;
	            free(root);
	            return temp;
	        }

	        struct Node *succ = successor(root);
	        root->key = succ->key;
	        root->right = DeleteNode(root->right,succ->key);
	    }

	    return root;
	}

	void Preorder(struct Node *root){
	    if(root == NULL){
	        return;
	    }

	    printf("%d ",root->key);
	    Preorder(root->left);
	    Preorder(root->right);
	}

	void Postorder(struct Node *root){
	    if(root == NULL){
	```

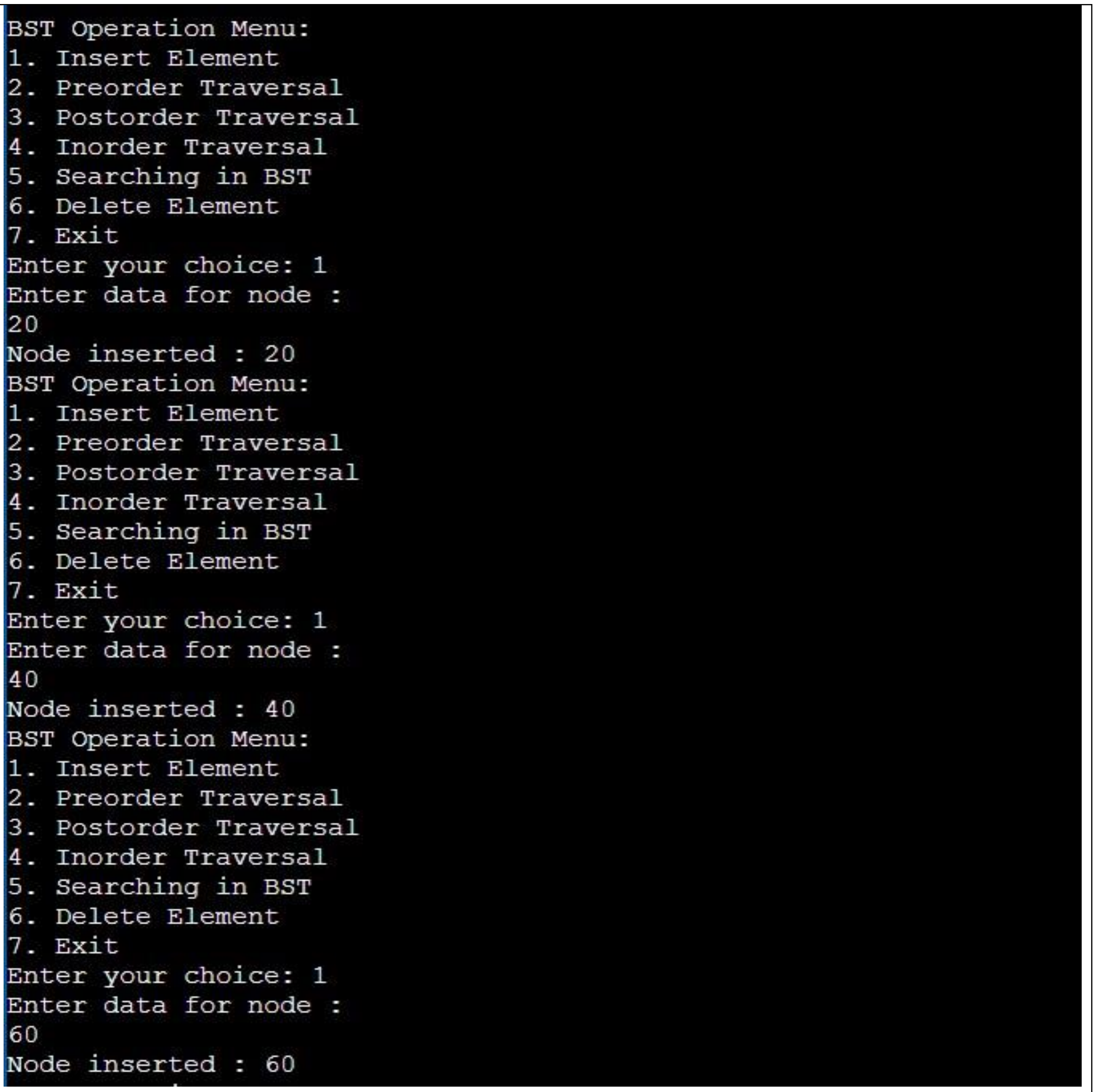
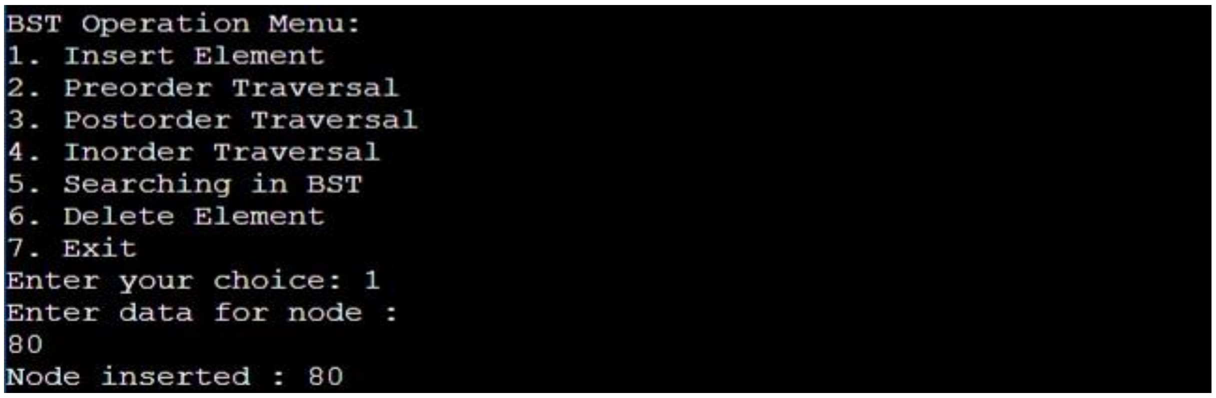
```
        return;
    }

    Postorder(root->left);
    Postorder(root->right);
    printf("%d ",root->key);
}

void InOrder(struct Node *root){
    if(root == NULL){
        return;
    }

    InOrder(root->left);
    printf("%d ",root->key);
    InOrder(root->right);
}

int main(){
    int choice;
    struct Node *root = NULL;



    do{
        printf("\nBST Operation Menu:\n");
        printf("1. Insert Element\n");
        printf("2. Preorder Traversal\n");
        printf("3. Postorder Traversal\n");
        printf("4. Inorder Traversal\n");
        printf("5. Searching in BST\n");
        printf("6. Delete Element\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
```

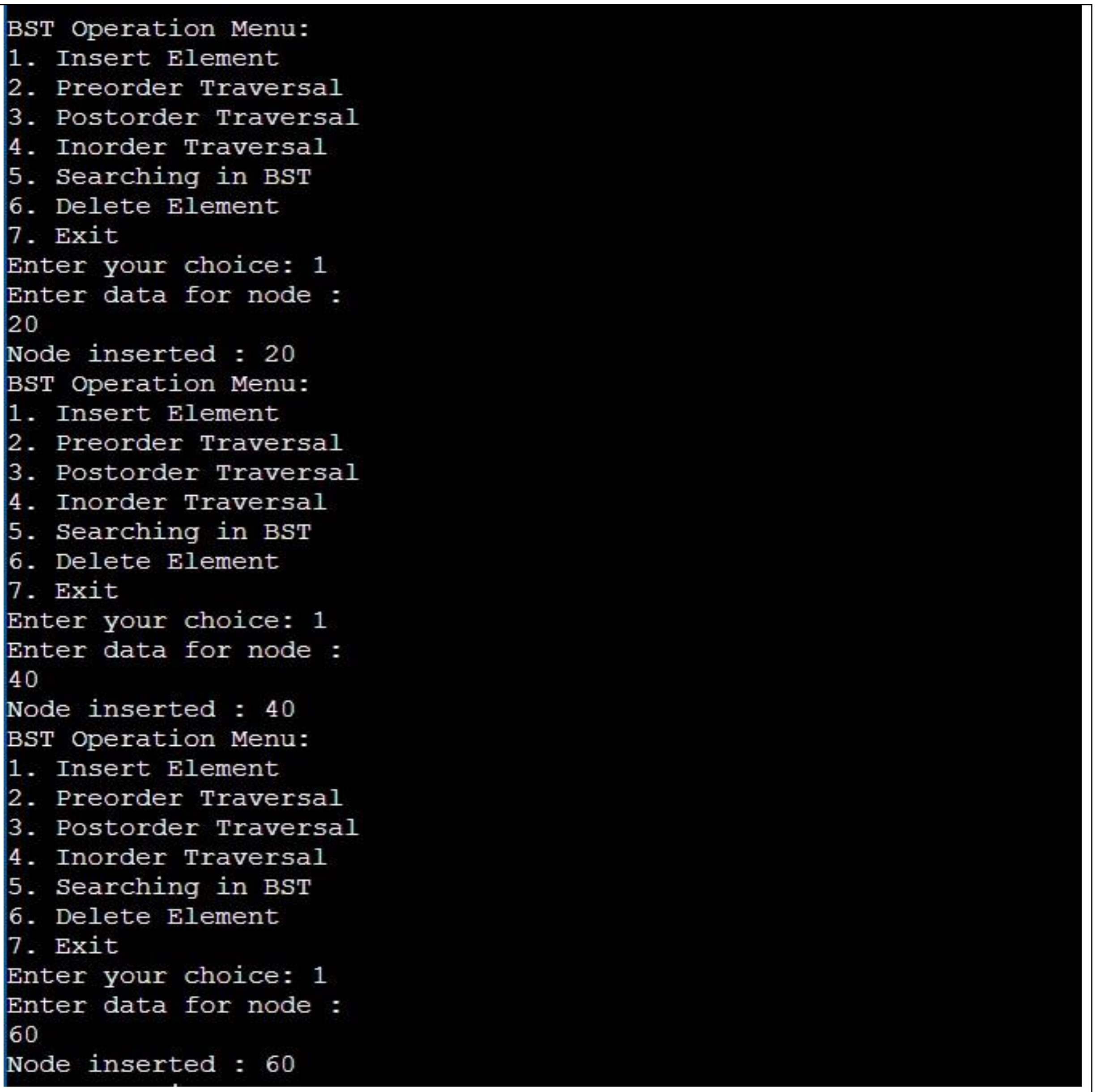
```
case 1:
{
    int key;
    printf("Enter data for node :\n");
    scanf("%d",&key);
    root = insertBST(root,key);
    printf("Node inserted : %d",key);
}
    break;
case 2:
    printf("Preordered traversed..\n");
    Preorder(root);
    break;
case 3:
    printf("Postordered traversed..\n");
    Postorder(root);
    break;
case 4:
    printf("Inordered Traversed..\n");
    InOrder(root);
    break;
case 5:
{
    int element;
    printf("Enter element you want check present or not in BST..\n");
    scanf("%d",&element);
    struct Node *result = searchNode(root,element);
    if(result != NULL){
        printf("Element found in BST  : %d",element);
    }else{
        printf("Element not found in BST : %d",element);
    }
}
    break;
case 6:
{
    int element;
```

```
            printf("Enter element you want to delete :\n");
            scanf("%d",&element);
            DeleteNode(root,element);
        }
            break;
        case 7:
            break;
        default:
            break;
        }
    }while (choice != 7);

    return 0;
}
```

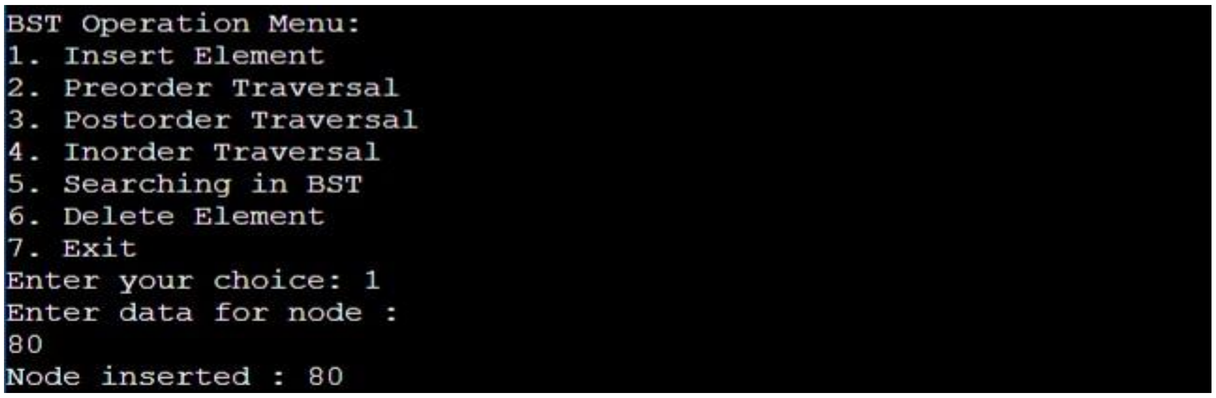
RESULT :-	1) Insert element in tree – 50 30 70 20 40 60 80 

```
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 1
Enter data for node :
50
Node inserted : 50
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 1
Enter data for node :
30
Node inserted : 30
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 70
```

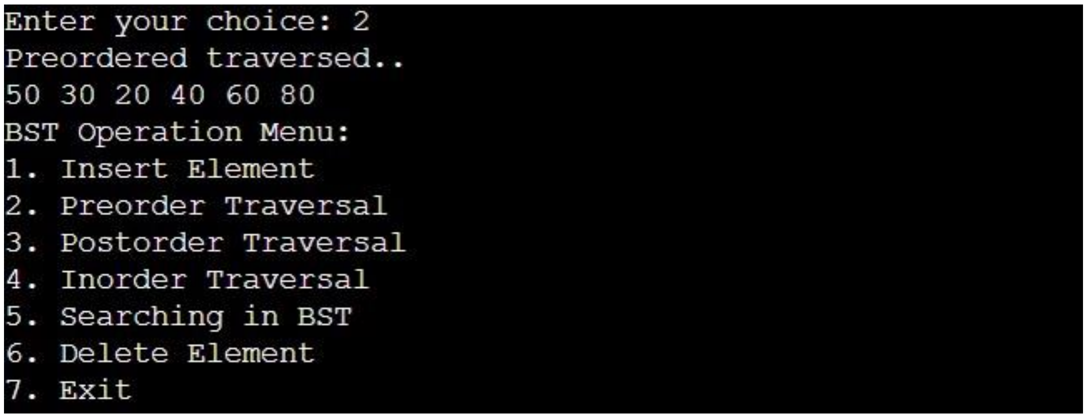
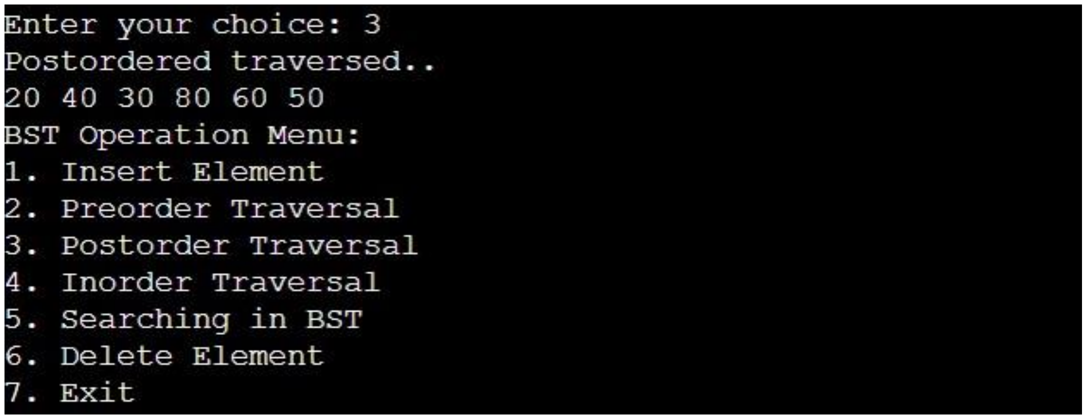
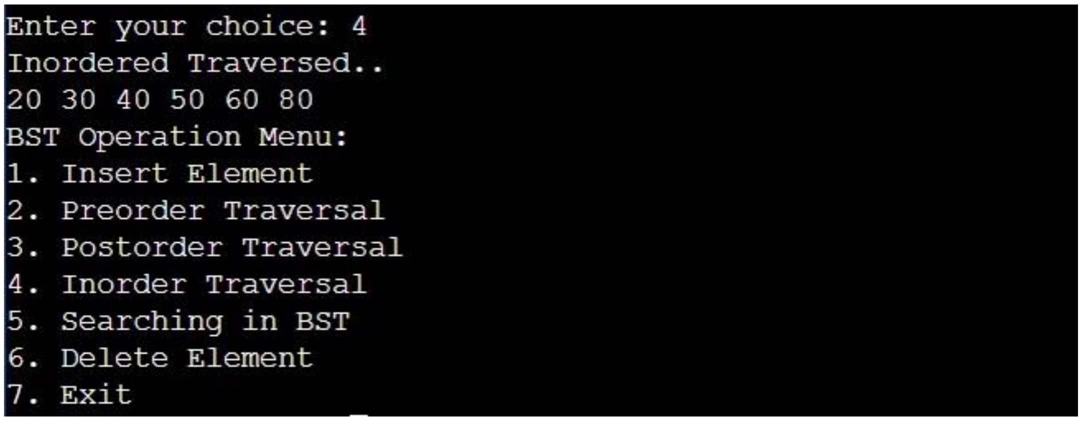
	 



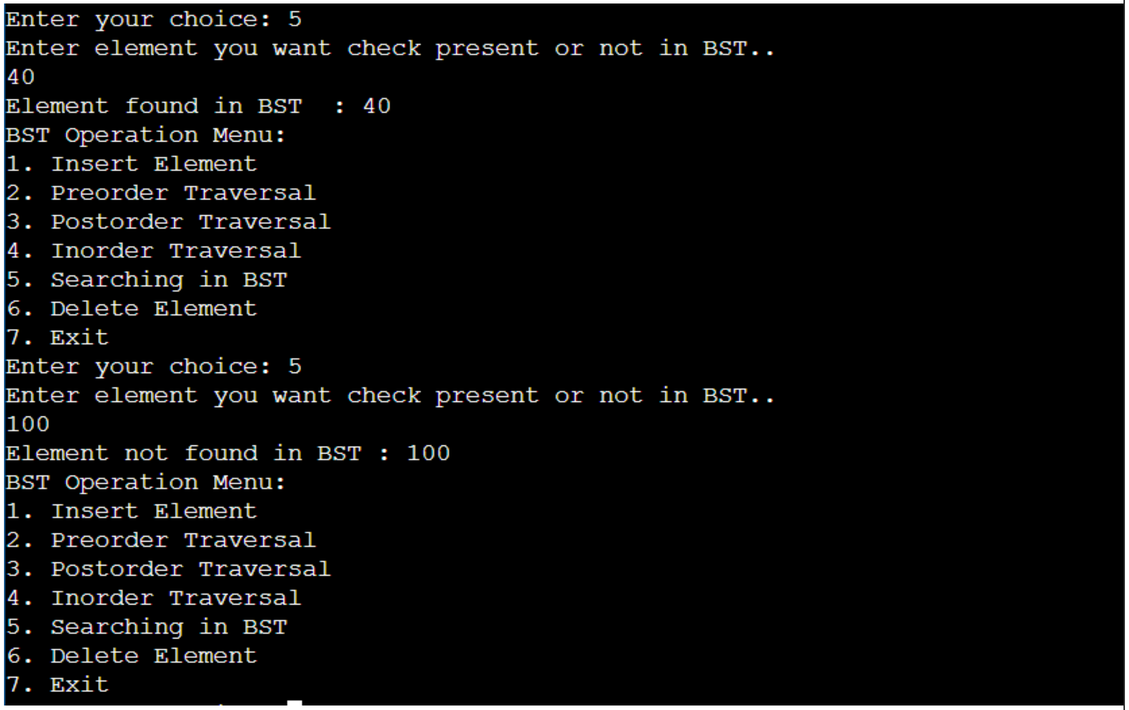
```
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 1
Enter data for node :
20
Node inserted : 20
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 1
Enter data for node :
40
Node inserted : 40
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 1
Enter data for node :
60
Node inserted : 60
```



```
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 1
Enter data for node :
80
Node inserted : 80
```

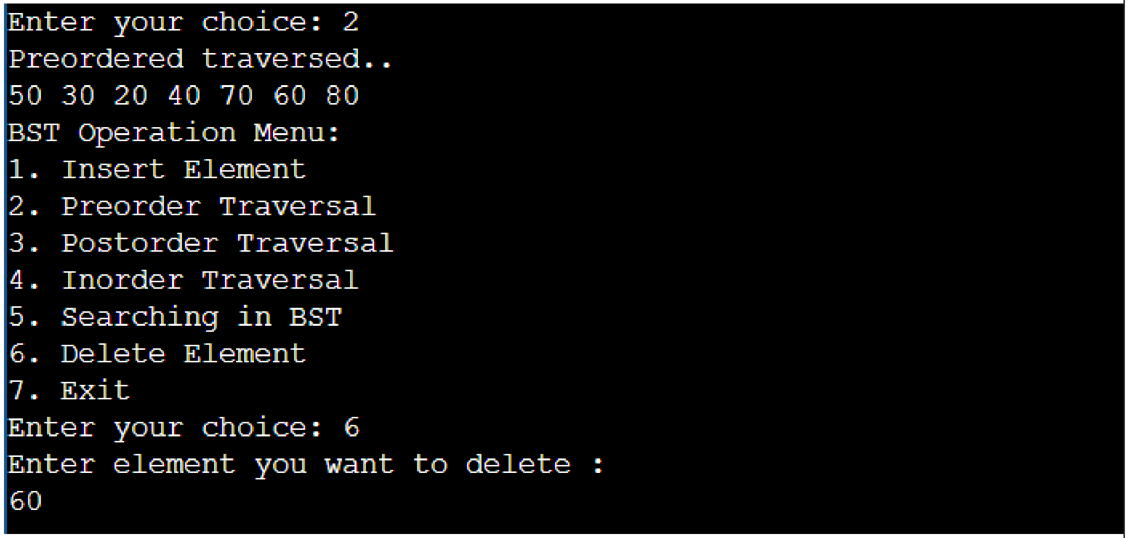
	2) Preorder Traversed output – 
	```
	Enter your choice: 2
	Preordered traversed..
	50 30 20 40 60 80
	BST Operation Menu:
	1. Insert Element
	2. Preorder Traversal
	3. Postorder Traversal
	4. Inorder Traversal
	5. Searching in BST
	6. Delete Element
	7. Exit
	```
	 3) Postorder Traversed output – 
	```
	Enter your choice: 3
	Postordered traversed..
	20 40 30 80 60 50
	BST Operation Menu:
	1. Insert Element
	2. Preorder Traversal
	3. Postorder Traversal
	4. Inorder Traversal
	5. Searching in BST
	6. Delete Element
	7. Exit
	```
	 4) Inordered Traversed output – 
	```
	Enter your choice: 4
	Inordered Traversed..
	20 30 40 50 60 80
	BST Operation Menu:
	1. Insert Element
	2. Preorder Traversal
	3. Postorder Traversal
	4. Inorder Traversal
	5. Searching in BST
	6. Delete Element
	7. Exit
	```

5) BST Searching –

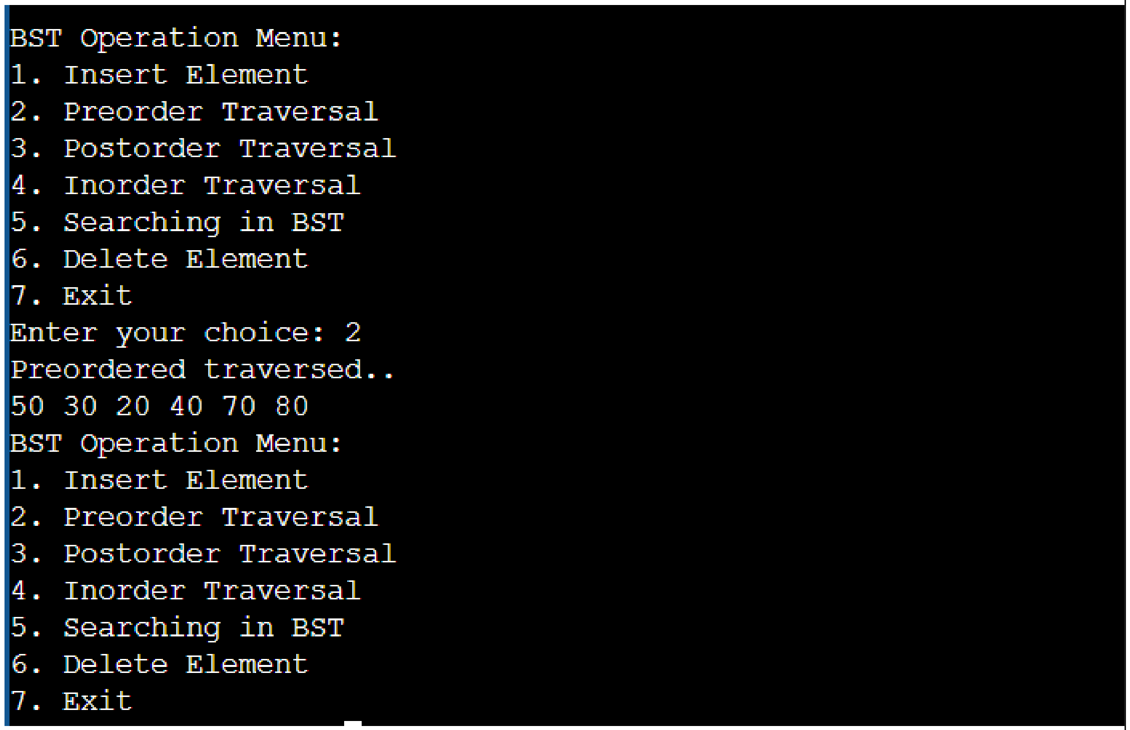


```
Enter your choice: 5
Enter element you want check present or not in BST..
40
Element found in BST  : 40
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 5
Enter element you want check present or not in BST..
100
Element not found in BST : 100
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
```

6) Delete Element Node from Tree –



```
Enter your choice: 2
Preordered traversed..
50 30 20 40 70 60 80
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 6
Enter element you want to delete :
60
```

```
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
Enter your choice: 2
Preordered traversed..
50 30 20 40 70 80
BST Operation Menu:
1. Insert Element
2. Preorder Traversal
3. Postorder Traversal
4. Inorder Traversal
5. Searching in BST
6. Delete Element
7. Exit
```

CONCLUSION : From this experiment with Binary Search Trees (BSTs), I learned how to effectively insert, delete, and traverse nodes while maintaining the tree's structure. Implementing different traversal methods deepened my understanding of tree operations, and handling deletion cases improved my problem-solving with dynamic data structures. This experience has enhanced my ability to work with tree-based algorithms, crucial for efficient searching and data management.