NAME: GAURAV KISHOR PATIL

DIV: 2 ROLL NO:54

BATCH: C

Experiment No.8
Implement Binary Search Tree ADT using Linked List.
Date of Performance:
Date of Submission:

Experiment No. 8: Binary Search Tree Operations

Aim : Implementation of Binary Search Tree ADT using Linked List.

Objective:

1) Understand how to implement a BST using a predefined BST ADT.

2) Understand the method of counting the number of nodes of a binary tree.

Theory:

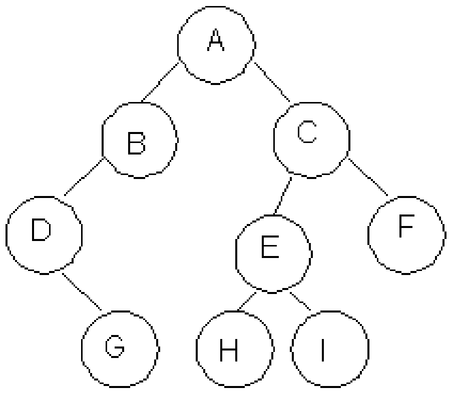
A binary tree is a finite set of elements that is either empty or partitioned into disjoint subsets. In other words node in a binary tree has at most two children and each child node is referred as left or right child.

Traversals in tree can be in one of the three ways : preorder, postorder, inorder.

Preorder Traversal

Here the following strategy is followed in sequence

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



Description	Output
Visit Root	A
Traverse left sub tree – step to B then D	ABD
Traverse right sub tree – step to G	ABDG
As left subtree is over. Visit root , which is already visited so go for right subtree	ABDGC
Traverse the left subtree	ABDGCEH
Traverse the right sub tree	ABDGCEHIF

Inorder Traversal

Here the following strategy is followed in sequence

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

Description	Output
Start with root and traverse left sub tree from A-B-D	D
As D doesn't have left child visit D and go for right subtree of D which is G so visit this.	DG
Backtrack to D and then to B and visit it.	DGB
Backtract to A and visit it	DGBA

Start with right sub tree from C-E-H and visit H	DGBAH
Now traverse through parent of H which is E and then I	DGBAHEI
Backtrack to C and visit it and then right subtree of E which is F	DGBAHEICF

Postorder Traversal

Here the following strategy is followed in sequence

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

Description	**Output**
Start with left sub tree from A-B-D and then traverse right sub tree to get G	G
Now Backtrack to D and visit it then to B and visit it.	GD
Now as the left sub tree is over go for right sub tree	GDB
In right sub tree start with leftmost child to visit H followed by I	GDBHI
Visit its root as E and then go for right sibling of C as F	GDBHIEF
Traverse its root as C	GDBHIEFC
Finally a root of tree as A	GDBHIEFCA

Algorithm

Algorithm: PREORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in preorder fashion.

Description : Linked structure of binary tree

```
1.    ptr=ROOT
2.    if ptr!=NULL then
          visit(ptr)
          PREORDER(LSON(ptr))\
          PREORDER(RSON(ptr))
      End if
```

```
3.      Stop
```

Algorithm: INORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in inorder fashion.

Description : Linked structure of binary tree

```
1. ptr=ROOT
2. if ptr!=NULL then
   INORDER (LSON(ptr))
   visit(ptr)
   INORDER (RSON(ptr))
   End if
3. Stop
```

Algorithm: POSTORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in postorder fashion.

Description : Linked structure of binary tree

```
1. ptr=ROOT
2. if ptr!=NULL then
   PREORDER(LSON(ptr))
   PREORDER(RSON(ptr))
   visit(ptr)
   End if
3. Stop
```

Code:

```
#include <stdio.h>

#include <stdlib.h>
```

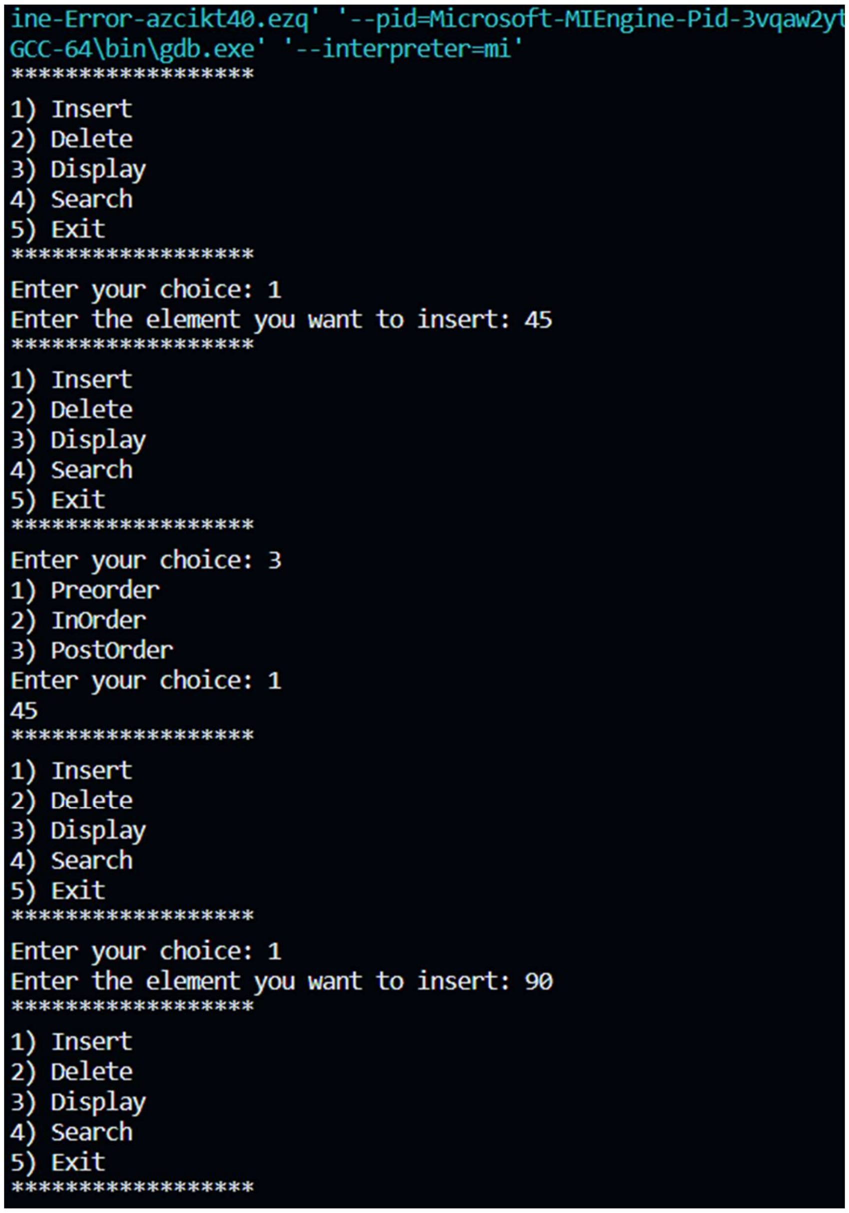
```
struct node {
    int key;
    struct node *right;
    struct node *left;
};



struct node *newnode(int x) {
    struct node *temp = malloc(sizeof(struct node));
    temp->key = x;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}



struct node *insert(struct node *root, int ele) {
    if (root == NULL) {
        return newnode(ele);
    } else if (ele > root->key) {
        root->right = insert(root->right, ele);
    } else {
        root->left = insert(root->left, ele);
    }
    return root;
}
```

```
struct node *getsucc(struct node *root) {
    struct node *curr = root->right;
    while (curr != NULL && curr->left != NULL) {
        curr = curr->left;
    }
    return curr;
}



struct node *delete(struct node *root, int ele) {
    if (root == NULL) {
        return root;
    }
    if (ele > root->key) {
        root->right = delete(root->right, ele);
    } else if (ele < root->key) {
        root->left = delete(root->left, ele);
    } else {
        // Case: Node with only one child or no child
        if (root->left == NULL) {
            struct node *temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
        // Case: Node with two children, get the inorder successor
        struct node *temp = getsucc(root);
        root->key = temp->key;
        root->right = delete(root->right, temp->key);
    }
    return root;
}


void preorder(struct node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preorder(root->left);
        preorder(root->right);
    }
}


void inorder(struct node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}



void postorder(struct node *root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->key);
    }
}



int search(struct node *root, int x) {
    if (root == NULL) {
        return 0;
    } else if (root->key == x) {
        return 1;
    } else if (x > root->key) {
        return search(root->right, x);
    } else {
        return search(root->left, x);
    }
}
```

```
int main() {
    int choice, ele, choice2, f;
    struct node *root=NULL;



    do {
        printf("******************\n");
        printf("1) Insert\n");
        printf("2) Delete\n");
        printf("3) Display\n");
        printf("4) Search\n");
        printf("5) Exit\n");
        printf("******************\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element you want to insert: ");
                scanf("%d", &ele);
                root = insert(root, ele);
                break;

            case 2:
                printf("Enter the element you want to delete: ");
                scanf("%d", &ele);
                root = delete(root, ele);
                break;

            case 3:
                printf("1) Preorder\n2) InOrder\n3) PostOrder\n");
                printf("Enter your choice: ");
                scanf("%d", &choice2);
                switch (choice2) {
                    case 1:
                        preorder(root);
                        break;
                    case 2:
                        inorder(root);
                        break;
                    case 3:
                        postorder(root);
                        break;
                    default:
                        break;
                }
                printf("\n");
                break;

            case 4:
                printf("Element you want to search: ");
                scanf("%d", &ele);
```

```
            f = search(root, ele);
            if (f == 1) {
                printf("Element found\n");
            } else if (f == 0) {
                printf("Element not found\n");
            }
            break;

        case 5:
            printf("Program Closed\n");
            break;

        default:
            printf("Invalid Choice\n");
            break;
        }

    } while (choice != 5);

    return 0;
}
```

Output:



```
ine-Error-azcikt40.ezq' '--pid=Microsoft-MIEngine-Pid-3vqaw2yt
GCC-64\bin\gdb.exe' '--interpreter=mi'
******************
1) Insert
2) Delete
3) Display
4) Search
5) Exit
******************
Enter your choice: 1
Enter the element you want to insert: 45
******************
1) Insert
2) Delete
3) Display
4) Search
5) Exit
******************
Enter your choice: 3
1) Preorder
2) InOrder
3) PostOrder
Enter your choice: 1
45
******************
1) Insert
2) Delete
3) Display
4) Search
5) Exit
******************
Enter your choice: 1
Enter the element you want to insert: 90
******************
1) Insert
2) Delete
3) Display
4) Search
5) Exit
******************
```

Conclusion:

Binary Search Trees (BST) in C language are a fundamental data structure that allow for efficient operations such as lookup, addition, and removal of items. They maintain elements in a sorted order and can dynamically adjust their size. However, to ensure optimal performance, it's crucial to keep the BST balanced. Unbalanced BSTs can lead to slower operations, which is why self-balancing BSTs like AVL trees or Red-Black trees are often used in practice.

Time Complexity: O(H)

Space Complexity: O(H)

Here H represents the height of the node