Date : Page No :

Assignment Number

Problem Statement

Program in C to perform elementary operations on a Binary Search Tree (BST)

Theory

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Algorithm

Input : An user defined structure say node consist of a data part say, data and two link parts say, left and right.

Output : Successful insertion, deletion and preorder, postorder or inorder traversal of the BST.

Steps:

Algorithm_Create_BST()

Step 1 : If(root = NULL)
Then
1. set root = GetNode()
2. set root->left = root->right = NULL
3. set root ->data = elem
4. return root

Step 2 : Else
1. If(elem<root->data)
Then
1. set root->left = Create_BST()
2. Else
1. If(elem > root->data)
Then
1. set root->right = Create_BST()
2. Else
1. Print “Duplicate Element !! Not Allowed !!!”
EndIf
EndIf
EndIf

Step 3 : Return root

Algorithm_Inorder(root)

Step 1 : If (root != Null)
a. Inorder(root->left)
b. Print root->data
c. Inorder(root->right)

EndIf

Algorithm_Preorder(root)

Step 1 : If(root != Null)

a. Print root->data
b. Preorder(root->left)
c. Preorder(root->right)

EndIf

Algorithm_Postorder(root)

Step 1 : If(root != Null)

a. Postorder(root->left)
b. Postorder(root->right)
c. Print root→data

EndIf

Algorithm_Delete(root)

Step 1 : If (root == NULL) return root
Step 2 : If (key < root->key)

Then

a. root->left = deleteNode(root->left, key)

Step 3 : Else if (key > root->key) Then

a. root->right = deleteNode(root->right, key)

Step 4 : Else

a. If (root->left == NULL) Then
    i. temp = root->right
    ii. free(root)
    iii. Return temp
b. Else If (root->right == NULL) Then
    i. temp = root->left
    ii. free(root)
    iii. Return temp

Step 5 : temp = minValueNode(root->right)
Step 6 : root->key = temp->key

Step 7 : root->right = deleteNode(root->right, temp->key);
Step 8 : Return root

Source Code

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node{
   int key;
   struct node *left, *right;
}node;
// A utility function to create a new BST node
node *newNode(int item){
   struct node *temp =  (struct node *)malloc(sizeof(struct node));
   temp->key = item;
   temp->left = temp->right = NULL;
   return temp;
}
// A utility function to do inorder traversal of BST
void inorder(struct node *root){
   if (root != NULL){
      inorder(root->left);
      printf("%d \n", root->key);
      inorder(root->right);
   }
}

/* A utility function to insert a new node with given key in BST */
node* insert(struct node* node, int key){
   /* If the tree is empty, return a new node */
   if (node == NULL) return newNode(key);

   /* Otherwise, recur down the tree */
   if (key < node->key)
      node->left  = insert(node->left, key);
   else if (key > node->key)
```

```
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
node * minValueNode(struct node* node){
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}
/* Given a binary search tree and a key, this function deletes the key
   and returns the new root */
node* deleteNode(struct node* root, int key){
         node *temp;
    // base case
    if (root == NULL) return root;
    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    // if key is same as root's key, then This is the node
    // to be deleted
    else{
        // node with only one child or no child
        if (root->left == NULL){
            temp = root->right;
```

```
            free(root);
            return temp;
        }
        else if (root->right == NULL){
            temp = root->left;
            free(root);
            return temp;
        }
        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        temp = minValueNode(root->right);
        // Copy the inorder successor's content to this node
        root->key = temp->key;
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
/* To find the preorder traversal */
void preorder(node *temp){
    if (temp == NULL){
        printf("No elements in a tree to display\n");
        return;
    }
    printf("%d\n", temp->key);
    if (temp->left != NULL)
        preorder(temp->left);
    if (temp->right != NULL)
        preorder(temp->right);
}
/* To find the postorder traversal */
void postorder(node *temp){
    if (temp == NULL){
        printf("No elements in a tree to display\n");
        return;
    }
```

```
    if (temp->left != NULL)
        postorder(temp->left);
    if (temp->right != NULL)
        postorder(temp->right);
    printf("%d\n", temp->key);
}
// Driver Program to test above functions
int main(){
    node *root = NULL;
    int ch,key;
    while(1){       printf("1.Insert\n2.Delete\n3.Inorder\n4.Preorder\
n5.Postorder\n6.Exit\nEnter Your choice: ");
        scanf("%d",&ch);
        switch(ch){
                case 1: printf("Enter the element you want to insert: ");
                        scanf("%d",&key);
                        root=insert(root,key);
                        break;
                case 2: printf("Enter the element you want to delete: ");
                        scanf("%d",&key);
                        root=deleteNode(root,key);
                        break;
                case 3: inorder(root);
                        break;
                case 4: preorder(root);
                        break;
                case 5: postorder(root);
                        break;
                case 6: return 0;
            }
        }
return 0;
}
```

Input and Output

```
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 1
Enter the element you want to insert: 234
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 1
Enter the element you want to insert: 231
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 1
Enter the element you want to insert: 2321
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 3
231
234
2321
```

```
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 4
234
231
2321
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 5
231
2321
234
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 2
Enter the element you want to delete: 231
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 4
234
```

```
2321
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 3
234
2321
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 5
2321
234
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 2
Enter the element you want to delete: 234
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 3
2321
1.Insert
```

2.Delete
3.Inorder
4.Preorder
5.Postorder
6.Exit
Enter Your choice: 6

Discussion

1. The cost of insert(), delete(), inorder(), preorder and postorder can be kept to O(logN) where N is the number of nodes in the tree - so the benefit really is that lookups can be done in logarithmic time which matters a lot when N is large.
2. The keys stored in the tree are ordered in a manner. Any time to traverse the increasing (or decreasing) order of keys, the in-order (and reverse in-order) traversal just needed on the tree.
3. Order statistics can be implemented with binary search tree - Nth smallest, Nth largest element. This is because it is possible to look at the data structure as a sorted array.
4. Range queries can also be done. Like - find keys between N and M (N <= M).
5. BST can also be used in the design of memory allocators to speed up the search of free blocks (chunks of memory), and to implement best fit algorithms where we are interested in finding the smallest free chunk with size greater than or equal to size specified in allocation request.
6. The main disadvantage is that a balanced binary search tree should always be maintained - AVL tree, Red-Black tree, Splay tree. Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.