Lab 11: Implementation of 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.

Source Code:

```
#include <stdio.h>
#include <malloc.h>
struct BST
{
    int data;
    struct BST *left;
    struct BST *right;
};
typedef struct BST NodeType;
NodeType* root;
NodeType* create_node(int value);
NodeType* insert(NodeType*,int value);
void PreOrder(NodeType*);
void InOrder(NodeType*);
void PostOrder(NodeType*);
NodeType* remov(NodeType*,int value);
NodeType* Minimum(NodeType* root)
{
    while(root->left != NULL) root = root->left;
    return root;
}
int main()
{
    int choice,element;
    root=NULL;
    do
    {
```

```
        printf("\nBST\n");
        printf("1.INSERT\n2.DELETE\n3.TRAVERSAL\n4.EXIT\n");
        printf("Enter choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
        case 1:
            printf("Enter the element to be INSERTED: ");
            scanf("%d",&element);
            root=insert(root,element);
            printf("SUCCESS");
            break;

        case 2:
            printf("Enter the element to be REMOVED : ");
            scanf("%d",&element);
            root=remov(root,element);
            printf("SUCCESS");
            break;

        case 3:
            printf("\nPREORDER\n");
            PreOrder(root);
            printf("\nINORDER\n");
            InOrder(root);
            printf("\nPOSTORDER\n");
            PostOrder(root);
            break;
        }
    }while(choice!=4);
    return 0;
}
NodeType* create_node(int value)
{
    NodeType* NewNode;
    NewNode=(NodeType*)malloc(sizeof(NodeType));
    NewNode->data=value;
    NewNode->left=NewNode->right=NULL;
    return NewNode;
}
```

```
NodeType* insert(NodeType* rootP,int value)
{
    if(rootP==NULL)
        rootP=create_node(value);
    else if(rootP->data>value)
        rootP->left=insert(rootP->left,value);
    else
        rootP->right=insert(rootP->right,value);
    return rootP;
}
void PreOrder(NodeType* rootP)
{
    if(rootP!=NULL)
    {
    printf(" %d ",rootP->data);
    PreOrder(rootP->left);
    PreOrder(rootP->right);
    }
}
void InOrder(NodeType* rootP)
{
    if(rootP!=NULL)
    {
    InOrder(rootP->left);
    printf(" %d ",rootP->data);
    InOrder(rootP->right);
    }
}
void PostOrder(NodeType* rootP)
{
    if(rootP!=NULL)
    {
    PostOrder(rootP->left);
    PostOrder(rootP->right);
    printf(" %d ",rootP->data);
    }
}
NodeType* remov(NodeType *rootP, int data)
{
    if(rootP == NULL) return rootP;
```

```
    else if(data < rootP->data) rootP->left = remov(rootP->left,data);
    else if (data > rootP->data) rootP->right = remov(rootP->right,data);
    // Wohoo... I found you, Get ready to be deleted
    else {
        // Case 1:  No child
        if(rootP->left == NULL && rootP->right == NULL) {
            free(rootP);
            rootP = NULL;
        }
        //Case 2: One child
        else if(rootP->left == NULL) {
            NodeType *temp = rootP;
            rootP = rootP->right;
            free(temp);
        }
        else if(rootP->right == NULL) {
            NodeType *temp = rootP;
            rootP = rootP->left;
            free(temp);
        }
        // case 3: 2 children
        else {
            NodeType *temp = Minimum(rootP->right);
            rootP->data = temp->data;
            rootP->right = remov(rootP->right,temp->data);
        }
    }
    return rootP;
}
```