VISHWAKARMA INSTITUTE OF TECHNOLOGY

NAME	Arpit Sudhir Vidhale
ROLL NO.	60
DIVISION	CS-D
BATCH	B3
PRN NO.	12111229

DS LAB ASSIGNMENT 6

Question:
Write a Program to create a Binary Search Tree and perform following nonrecursive operations on it. a. Preorder Traversal b. Inorder Traversal c. Display Number of Leaf Nodes d. Mirror Image

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

struct node
{
   struct node *lchild;
   int info;
   struct node *rchild;
};

void nrec_pre(struct node *root);
void nrec_in(struct node *root);
struct node *insert_nrec(struct node *root, int ikey);
void display(struct node *ptr, int level);
void mirror(struct node *node);
unsigned int getLeafCount(struct node *node);

struct node *stack[MAX];
int top = -1;
void push_stack(struct node *item);
```

```
struct node *pop_stack();
int stack_empty();
int main()
{
    struct node *root = NULL, *ptr;
    int choice, k;

   while (1)
   {
      printf("\n");
      printf("1.Insert\n");
      printf("2.Display\n");
      printf("3.Preorder Traversal\n");
      printf("4.Inorder Traversal\n");
      printf("5.mirror\n");
      printf("6.number of leaf nodes\n");
      printf("7.Quit\n");
      printf("\nEnter your choice : ");
      scanf("%d", &choice);

      switch (choice)
      {

      case 1:
         printf("\nEnter the key to be inserted :
         "); scanf("%d", &k);
         root = insert_nrec(root, k);
         break;

      case 2:
         printf("\n");
         display(root, 0);
         printf("\n");
         break;
      case 3:
         printf("\n Preorder Traversal ->");
         nrec_pre(root);
         printf("\n");
         break;

      case 4:
         printf("\n Inorder Traversal ->");
```

```
            nrec_in(root);
            printf("\n");
            break;
        case 5:
            printf("\n");
            mirror(root);
            display(root, 0);
            printf("\n");
            break;

        case 6:
            printf("\n");
            printf("Leaf count of the tree is %d",
            getLeafCount(root)); printf("\n");
            break;

        case 7:
            exit(1);

        default:
            printf("\nWrong choice\n");
        }
    }
}

struct node *insert_nrec(struct node *root, int ikey)
{
    struct node *tmp, *par, *ptr;

    ptr = root;
    par = NULL;

    while (ptr != NULL)
    {
        par = ptr;
        if (ikey < ptr->info)
            ptr = ptr->lchild;
        else if (ikey > ptr->info)
            ptr = ptr->rchild;
        else
        {
            printf("\nDuplicate key");
```

```
            return root;
        }
    }
    tmp = (struct node *)malloc(sizeof(struct
    node)); tmp->info = ikey;
    tmp->lchild = NULL;
    tmp->rchild = NULL;

    if (par == NULL)
        root = tmp;
    else if (ikey < par->info)
        par->lchild = tmp;
    else
        par->rchild = tmp;

    return root;
}

void display(struct node *ptr, int level)
{
    int i;
    if (ptr == NULL)
        return;
    else
    {
        display(ptr->rchild, level + 1);
        printf("\n");
        for (i = 0; i < level; i++)
            printf(" ");
        printf("%d", ptr->info);
        display(ptr->lchild, level + 1);
    }
}

void nrec_pre(struct node *root)
{
    struct node *ptr = root;
    if (ptr == NULL)
    {
        printf("Tree is empty\n");
        return;
    }
```

```
    push_stack(ptr);
    while (!stack_empty())
    {
        ptr = pop_stack();
        printf("%d ", ptr->info);
        if (ptr->rchild != NULL)
            push_stack(ptr->rchild);
        if (ptr->lchild != NULL)
            push_stack(ptr->lchild);
    }
    printf("\n");
}

void nrec_in(struct node *root)
{
    struct node *ptr = root;

    if (ptr == NULL)
    {
        printf("Tree is empty\n");
        return;
    }
    while (1)
    {
        while (ptr->lchild != NULL)
        {
            push_stack(ptr);
            ptr = ptr->lchild;
        }

        while (ptr->rchild == NULL)
        {
            printf("%d ", ptr->info);
            if (stack_empty())
                return;
            ptr = pop_stack();
        }
        printf("%d ", ptr->info);
        ptr = ptr->rchild;
    }
    printf("\n");
}
```

```
unsigned int getLeafCount(struct node
*node) {
    if (node == NULL)
        return 0;
    if (node->lchild == NULL && node->rchild ==
        NULL) return 1;
    else
        return getLeafCount(node->lchild) +
              getLeafCount(node->rchild);
}

void mirror(struct node *node)
{
    if (node == NULL)
        return;
    else
    {
        struct node *temp;

        mirror(node->lchild);
        mirror(node->rchild);

        temp = node->lchild;
        node->lchild = node->rchild;
        node->rchild = temp;
    }
}

void push_stack(struct node *item)
{
    if (top == (MAX - 1))
    {
        printf("Stack Overflow\n");
        return;
    }
    top = top + 1;
    stack[top] = item;
}

struct node *pop_stack()
{
```

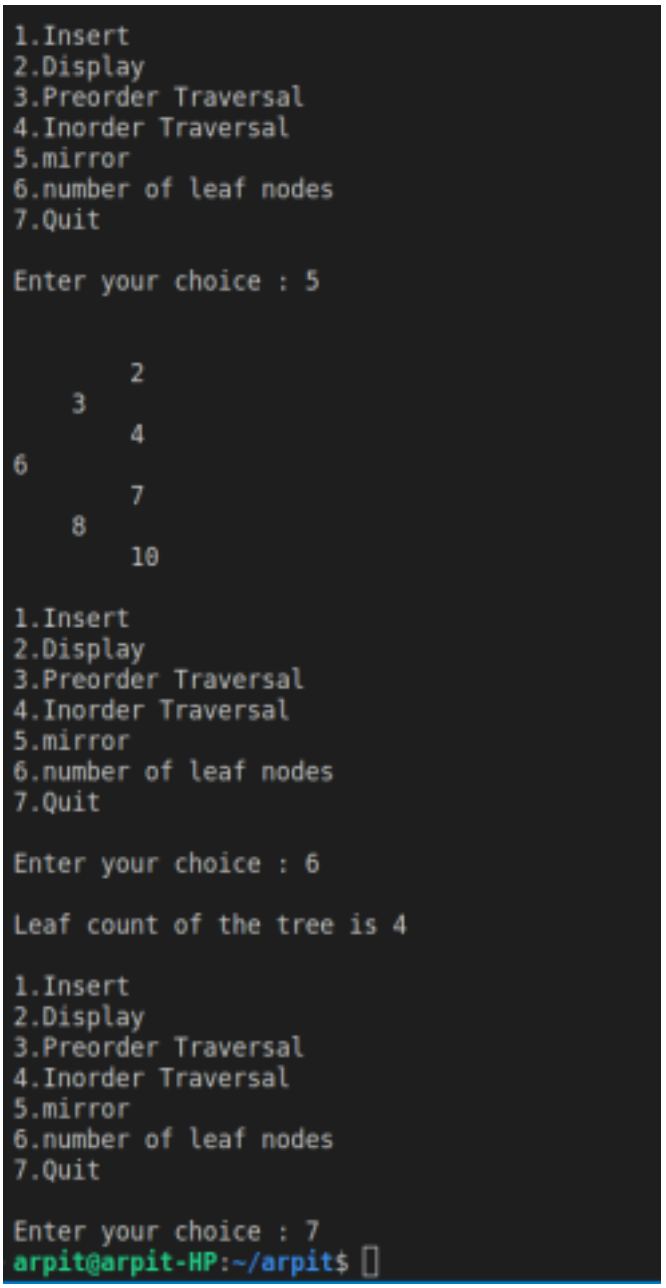
```
    struct node *item;
    if (top == -1)
    {
        printf("Stack Underflow....\n");
        exit(1);
    }
    item = stack[top];
    top = top - 1;
    return item;
}

int stack_empty()
{
    if (top == -1)
        return 1;
    else
        return 0;
}
```

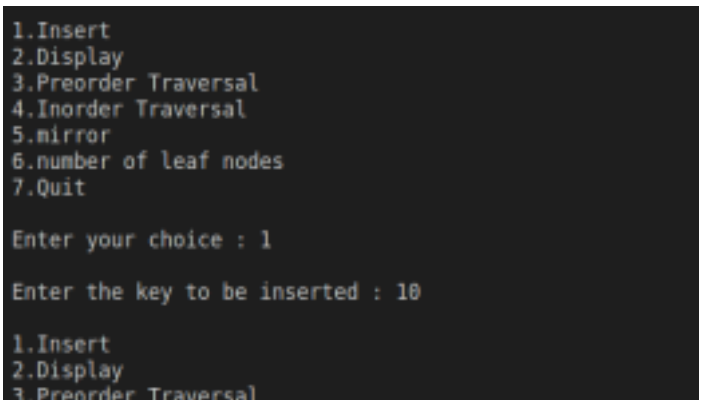
Output:

```
1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit

Enter your choice : 1

Enter the key to be inserted : 6

1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit

Enter your choice : 1

Enter the key to be inserted : 3

1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit

Enter your choice : 1

Enter the key to be inserted : 4
```

```
1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit

Enter your choice : 1

Enter the key to be inserted : 2

1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit

Enter your choice : 1

Enter the key to be inserted : 8

1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit

Enter your choice : 1

Enter the key to be inserted : 7
```

Preorder and inorder traversal :


1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit
Enter your choice : 5
2
3
4
6
7
8
10
1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit
Enter your choice : 6
Leaf count of the tree is 4
1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit
Enter your choice : 7
arpit@arpit-HP:~/arpit$


1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.mirror
6.number of leaf nodes
7.Quit
Enter your choice : 1
Enter the key to be inserted : 10
1.Insert
2.Display
3.Preorder Traversal

Mirror Image and number of leaf nodes:

Question:

Create BST and perform following operations on it.

A. Insertion.
B. Delete.
C. Level wise Display.
D. Mirror Image.
E. Height of the Tree.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 50

struct node
{
        struct node *lchild;
        int info;
        struct node *rchild;
};
struct node *insert_nrec(struct node *root, int ikey );
struct node *del_nrec(struct node *root, int dkey);
struct node *case_c(struct node *root, struct node *par,struct node
*ptr); struct node *case_b(struct node *root,struct node *par,struct node
*ptr); struct node *case_a(struct node *root, struct node *par,struct
node *ptr );
struct node *del_nrec1(struct node *root, int item);
void mirror(struct node *node);
void display(struct node *ptr,int level);
int height(struct node* node);
struct node *stack[MAX];
int top=-1;
void push_stack(struct node *item);
struct node *pop_stack();
int stack_empty();

int main( )
{
        struct node *root=NULL, *ptr;
        int choice,k;
```

```
while(1)
{
        printf("\n");
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Mirror Image\n");
        printf("5.Height\n");
        printf("6.Quit\n");
        printf("\nEnter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {

        case 1:
                printf("\nEnter the key to be inserted : ");
                scanf("%d",&k);
                root = insert_nrec(root, k);
                break;

        case 2:
                printf("\nEnter the key to be deleted :
                "); scanf("%d",&k);
                root = del_nrec(root, k);
                break;
                break;
        case 3:
                printf("\n");
                display(root,0);
                printf("\n");
                break;

        case 4:
                printf("\n");
                mirror(root);
                display(root, 0);
                printf("\n");
                break;
        case 5:
```

```
                printf("\n");
                printf("Height of tree is %d", height(root));
                printf("\n");
                break;
            case 6:
                exit(1);
            default:
                printf("\nWrong choice\n");
            }
        }

        return 0;
}
struct node *insert_nrec(struct node *root, int ikey)
{
        struct node *tmp,*par,*ptr;

        ptr = root;
        par = NULL;

        while( ptr!=NULL)
        {
            par = ptr;
            if(ikey < ptr->info)
                ptr = ptr->lchild;
            else if( ikey > ptr->info )
                ptr = ptr->rchild;
            else
            {
                printf("\nDuplicate key");
                return root;
            }
        }
        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=ikey;
        tmp->lchild=NULL;
        tmp->rchild=NULL;

        if(par==NULL)
            root=tmp;
```

```
        else if( ikey < par->info )
                par->lchild=tmp;
        else
                par->rchild=tmp;

        return root;
}
struct node *del_nrec1(struct node *root, int dkey)
{
        struct node *par,*ptr, *child, *succ, *parsucc;

        ptr = root;
        par = NULL;
        while( ptr!=NULL)
        {
                if( dkey == ptr->info)
                        break;
                par = ptr;
                if(dkey < ptr->info)
                        ptr = ptr->lchild;
                else
                        ptr = ptr->rchild;
        }
        if(ptr==NULL)
        {
                printf("\ndkey not present in tree");
                return root;
        }

        if(ptr->lchild!=NULL &&
        ptr->rchild!=NULL) {
                parsucc = ptr;
                succ = ptr->rchild;
                while(succ->lchild!=NULL)
                {
                        parsucc = succ;
                        succ = succ->lchild;
                }
                ptr->info = succ->info;
                ptr = succ;
```

```
            par = parsucc;
        }

        if(ptr->lchild!=NULL)
            child=ptr->lchild;
        else
            child=ptr->rchild;

        if(par==NULL )
            root=child;
        else if( ptr==par->lchild)
            par->lchild=child;
        else
            par->rchild=child;
        free(ptr);
        return root;
}
struct node *del_nrec(struct node *root, int dkey)
{
        struct node *par,*ptr;

        ptr = root;
        par = NULL;
        while(ptr!=NULL)
        {
            if( dkey == ptr->info)
                break;
            par = ptr;
            if(dkey < ptr->info)
                ptr = ptr->lchild;
            else
                ptr = ptr->rchild;
        }

        if(ptr==NULL)
            printf("dkey not present in tree\n");
        else if(ptr->lchild!=NULL &&
            ptr->rchild!=NULL) root =
            case_c(root,par,ptr);
        else if(ptr->lchild!=NULL )
```

```
        root = case_b(root, par,ptr);
        else if(ptr->rchild!=NULL)
        root = case_b(root, par,ptr);
        else
                root = case_a(root,par,ptr);

        return root;
}
void mirror(struct node *node)
{
    if (node == NULL)
        return;
    else
    {
        struct node *temp;

        mirror(node->lchild);
        mirror(node->rchild);

        temp = node->lchild;
        node->lchild = node->rchild;
        node->rchild = temp;
    }
}
int height(struct node* node)
{
    if (node == NULL)
        return 0;
    else {

        int lheight = height(node->lchild);
        int rheight = height(node->rchild);

        if (lheight > rheight)
            return (lheight + 1);
        else
            return (rheight + 1);
    }
}
struct node *case_a(struct node *root, struct node *par,struct node *ptr )
```

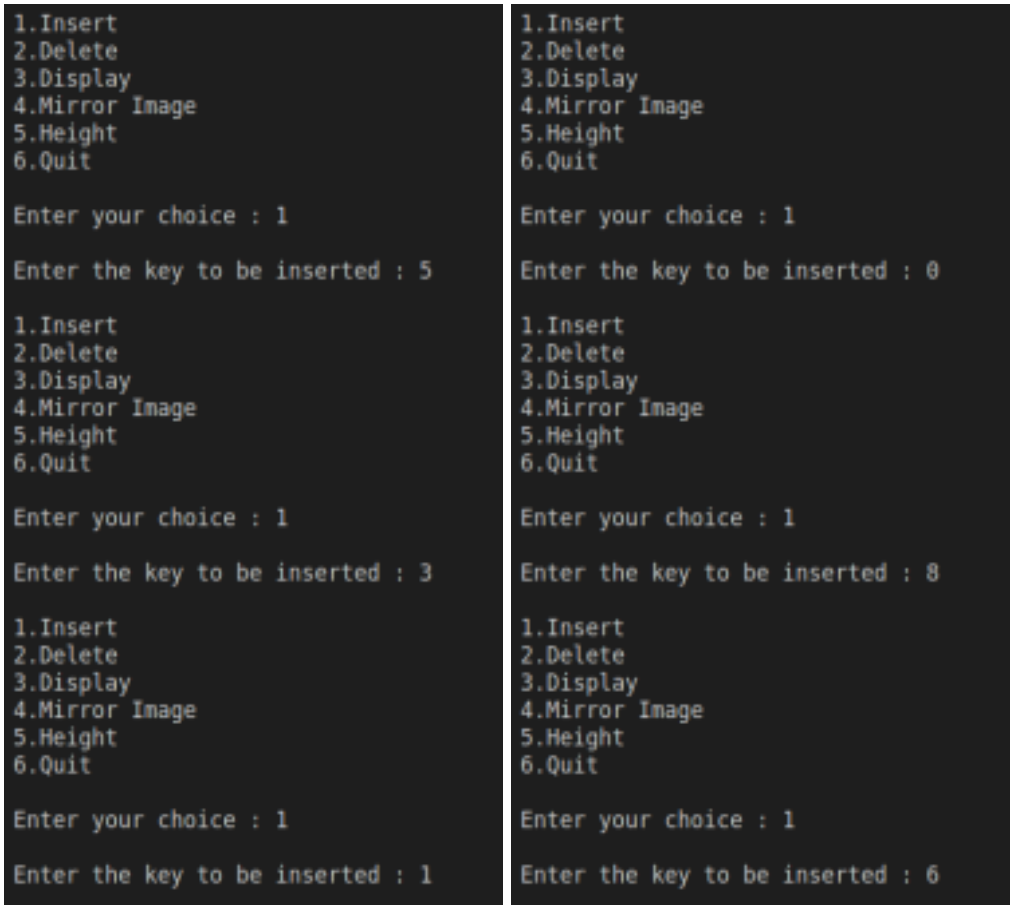
```
{
        if(par==NULL) /*root node to be deleted*/
                root=NULL;
        else if(ptr==par->lchild)
                par->lchild=NULL;
        else
                par->rchild=NULL;
        free(ptr);
        return root;
}/*End of case_a( )*/

struct node *case_b(struct node *root,struct node *par,struct node *ptr)
{
        struct node *child;


        if(ptr->lchild!=NULL)
                child=ptr->lchild;
        else
                child=ptr->rchild;

        if(par==NULL )
                root=child;
        else if( ptr==par->lchild)
                par->lchild=child;
        else
                par->rchild=child;
        free(ptr);
        return root;
}

struct node *case_c(struct node *root, struct node *par,struct node *ptr)
{
        struct node *succ,*parsucc;
        parsucc = ptr;
        succ = ptr->rchild;
        while(succ->lchild!=NULL)
        {
                parsucc = succ;
                succ = succ->lchild;
```

```
        }
        ptr->info = succ->info;

                        if(succ->lchild==NULL &&
            succ->rchild==NULL) root = case_a(root,
                                parsucc,succ);
        else
                root = case_b(root, parsucc,succ);
        return root;
}

void push_stack(struct node *item)
{
        if(top==(MAX-1))
        {
                printf("Stack Overflow\n");
                return;
        }
        top=top+1;
        stack[top]=item;
}

struct node *pop_stack()
{
        struct node *item;
        if(top==-1)
        {
                printf("Stack Underflow....\n");
                exit(1);
        }
        item=stack[top];
        top=top-1;
        return item;
}
int stack_empty()
{
        if(top==-1)
                return 1;
        else
                return 0;
```

```
}
void display(struct node *ptr,int level)
{
        int i;
        if(ptr == NULL )
                return;
        else
    {
                display(ptr->rchild, level+1);
                printf("\n");
                for (i=0; i<level; i++)
                        printf(" ");
                printf("%d", ptr->info);
                display(ptr->lchild, level+1);
        }
}
```

Output:

Insert

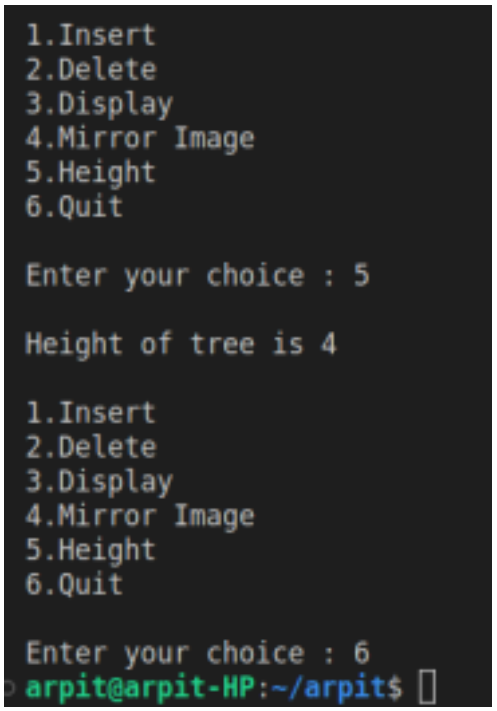


Display and deletion

```
1.Insert
2.Delete
3.Display
4.Mirror Image
5.Height
6.Quit

Enter your choice : 2

Enter the key to be deleted : 6

1.Insert
2.Delete
3.Display
4.Mirror Image
5.Height
6.Quit

Enter your choice : 3


    8
        7
5
    3
        1
            0
```

```
1.Insert
2.Delete
3.Display
4.Mirror Image
5.Height
6.Quit

Enter your choice : 1

Enter the key to be inserted : 7

1.Insert
2.Delete
3.Display
4.Mirror Image
5.Height
6.Quit

Enter your choice : 3


    8
            7
        6
5
    3
        1
            0
```

```
1.Insert
2.Delete
3.Display
4.Mirror Image
5.Height
6.Quit

Enter your choice : 4


            0
        1
    3
5
        7
    8
```



Mirror Image and Height