```c
/*Aim 13B:-  C Program for Non recursive operations (including preorder traversal) in Binary Search Tree
*/

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

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

struct node *search_nrec(struct node *root, int skey);
struct node *min_nrec(struct node *root);
struct node *max_nrec(struct node *root);
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 nrec_pre(struct node *root);
void nrec_in(struct node *root);
void nrec_post(struct node *root);
void level_trav(struct node *root);
void display(struct node *ptr,int level);


struct node *queue[MAX];
int front=-1,rear=-1;
void insert_queue(struct node *item);
struct node *del_queue();
int queue_empty();

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.Search\n");
    printf("2.Insert\n");
    printf("3.Delete\n");
    printf("4.Preorder Traversal\n");
    printf("5.Inorder Traversal\n");
    printf("6.Postorder Traversal\n");
    printf("7.Level order traversal\n");
    printf("8.Find minimum and maximum\n");
    printf("9.Display\n");
    printf("10.Quit\n");
    printf("\nEnter your choice : ");
    scanf("%d",&choice);

    switch(choice)
    {

    case 1:
        printf("\nEnter the key to be searched : ");
        scanf("%d",&k);
        ptr = search_nrec(root, k);
        if(ptr==NULL)
            printf("\nKey not present\n");
        else
            printf("\nKey present\n");
        break;

    case 2:
        printf("\nEnter the key to be inserted : ");
        scanf("%d",&k);
        root = insert_nrec(root, k);
        break;

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

    case 4:
        nrec_pre(root);
        break;

    case 5:
        nrec_in(root);
        break;
        case 6:
            nrec_post(root);
            break;

        case 7:
            level_trav(root);
            break;

        case 8:
            ptr = min_nrec(root);
            if(ptr!=NULL)
                printf("\nMinimum key is %d\n", ptr->info );
            ptr = max_nrec(root);
            if(ptr!=NULL)
                printf("\nMaximum key is %d\n", ptr->info );
            break;

    case 9:
            printf("\n");
            display(root,0);
            printf("\n");
            break;

        case 10:
            exit(1);
        default:
            printf("\nWrong choice\n");
        }/*End of switch*/
    }/*End of while */

    return 0;

}/*End of main( )*/

struct node *search_nrec(struct node *ptr, int skey)
{
    while(ptr!=NULL)
    {
        if(skey < ptr->info)
            ptr = ptr->lchild; /*Move to left child*/
        else if(skey > ptr->info)
            ptr = ptr->rchild;  /*Move to right child */
        else    /*skey found*/
            return ptr;
    }
    return NULL;
}/*End of search_nrec()*/
```

```
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;
}/*End of insert_nrec( )*/

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;
    }

    /*Case C: 2 children*/
    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;
    }

    /*Case B and Case A : 1 or no child*/
    if(ptr->lchild!=NULL) /*node to be deleted has left child */
        child=ptr->lchild;
    else            /*node to be deleted has right child */
        child=ptr->rchild;

    if(par==NULL )   /*node to be deleted is root node*/
        root=child;
    else if( ptr==par->lchild)/*node is left child of its parent*/
        par->lchild=child;
    else      /*node is right child of its parent*/
        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)/*2 children*/
                root = case_c(root,par,ptr);
        else if(ptr->lchild!=NULL )/*only left child*/
        root = case_b(root, par,ptr);
        else if(ptr->rchild!=NULL)/*only right child*/
        root = case_b(root, par,ptr);
        else /*no child*/
                root = case_a(root,par,ptr);

        return root;
}/*End of del_nrec( )*/

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;

        /*Initialize child*/
        if(ptr->lchild!=NULL) /*node to be deleted has left child */
                child=ptr->lchild;
        else              /*node to be deleted has right child */
                child=ptr->rchild;

        if(par==NULL )   /*node to be deleted is root node*/
```

```
                root=child;
        else if( ptr==par->lchild)   /*node is left child of its parent*/
                par->lchild=child;
        else                 /*node is right child of its parent*/
                par->rchild=child;
        free(ptr);
        return root;
}/*End of case_b( )*/

struct node *case_c(struct node *root, struct node *par,struct node *ptr)
{
        struct node *succ,*parsucc;

        /*Find inorder successor and its parent*/
        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;
}/*End of case_c( )*/

struct node *min_nrec(struct node *ptr)
{
        if(ptr!=NULL)
                while(ptr->lchild!=NULL)
                        ptr=ptr->lchild;
        return ptr;
}/*End of min_nrec()*/

struct node *max_nrec(struct node *ptr)
{
        if(ptr!=NULL)
                while(ptr->rchild!=NULL)
                        ptr=ptr->rchild;
        return ptr;
}/*End of max_nrec()*/

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");
}/*End of nrec_pre*/

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");
```

```
}/*End of nrec_in( )*/

void nrec_post(struct node *root)
{
        struct node *ptr = root;
        struct node *q;

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

                while( ptr->rchild==NULL || ptr->rchild==q )
                {
                        printf("%d  ",ptr->info);
                        q = ptr;
                        if( stack_empty() )
                                return;
                        ptr = pop_stack();
                }
                push_stack(ptr);
                ptr = ptr->rchild;
        }
        printf("\n");
}/*End of nrec_post( )*/

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

        if( ptr==NULL )
        {
                printf("Tree is empty\n");
                return;
        }

        insert_queue(ptr);

        while( !queue_empty() ) /*Loop until queue is not empty*/
```

```
    {
        ptr=del_queue();
        printf("%d ",ptr->info);
        if(ptr->lchild!=NULL)
            insert_queue(ptr->lchild);
        if(ptr->rchild!=NULL)
            insert_queue(ptr->rchild);
    }
    printf("\n");
}/*End of level_trav( )*/

/*Functions for implementation of queue*/
void insert_queue(struct node *item)
{
    if(rear==MAX-1)
    {
        printf("Queue Overflow\n");
        return;
    }
    if(front==-1)  /*If queue is initially empty*/
        front=0;
    rear=rear+1;
    queue[rear]=item ;
}/*End of insert()*/

struct node *del_queue()
{
    struct node *item;
    if(front==-1 || front==rear+1)
    {
        printf("Queue Underflow\n");
        return 0;
    }
    item=queue[front];
    front=front+1;
    return item;
}/*End of del_queue()*/

int queue_empty()
{
    if(front==-1 || front==rear+1)
        return 1;
    else
        return 0;
}

/*Functions for implementation of stack*/
void push_stack(struct node *item)
```

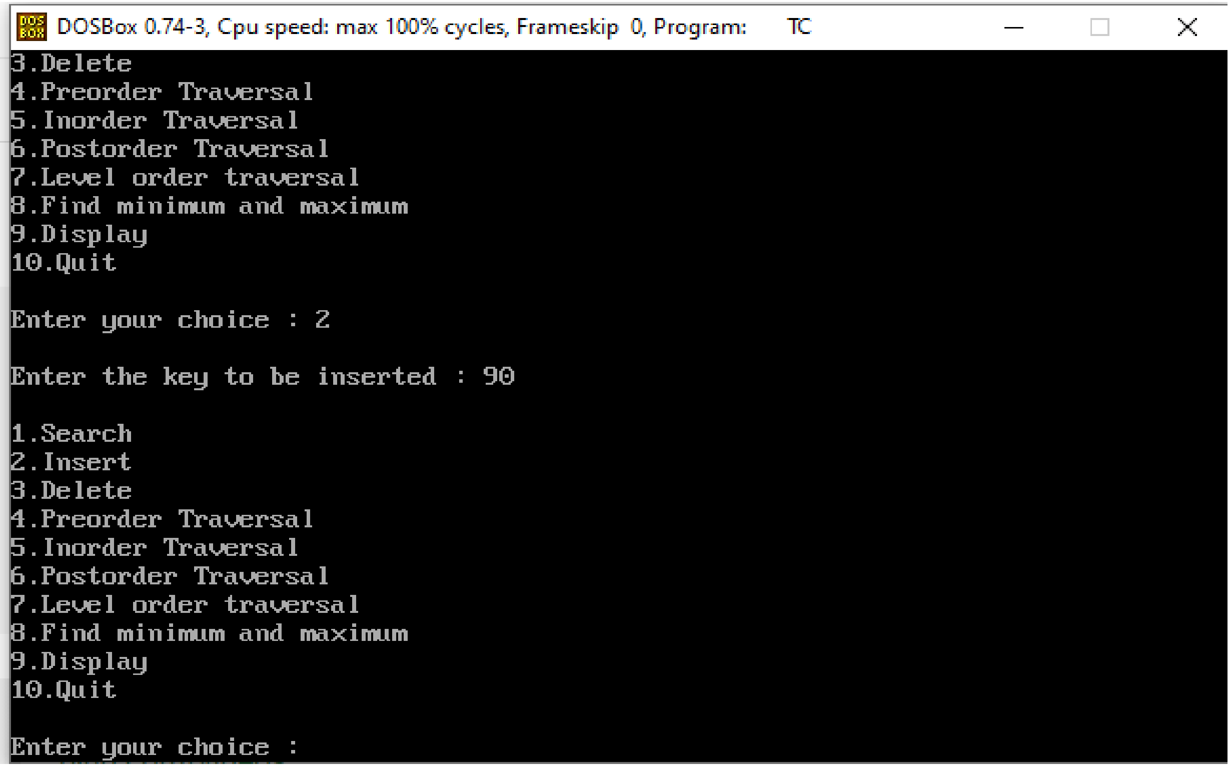
```
{
    if(top==(MAX-1))
    {
        printf("Stack Overflow\n");
        return;
    }
    top=top+1;
    stack[top]=item;
}/*End of push_stack()*/

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

int stack_empty()
{
    if(top==-1)
        return 1;
    else
        return 0;
} /*End of stack_empty*/

void display(struct node *ptr,int level)
{
    int i;
    if(ptr == NULL )/*Base Case*/
        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);
    }
}/*End of display()*/
```

OUTPUT13B:-


DOSBox 0.74-3, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

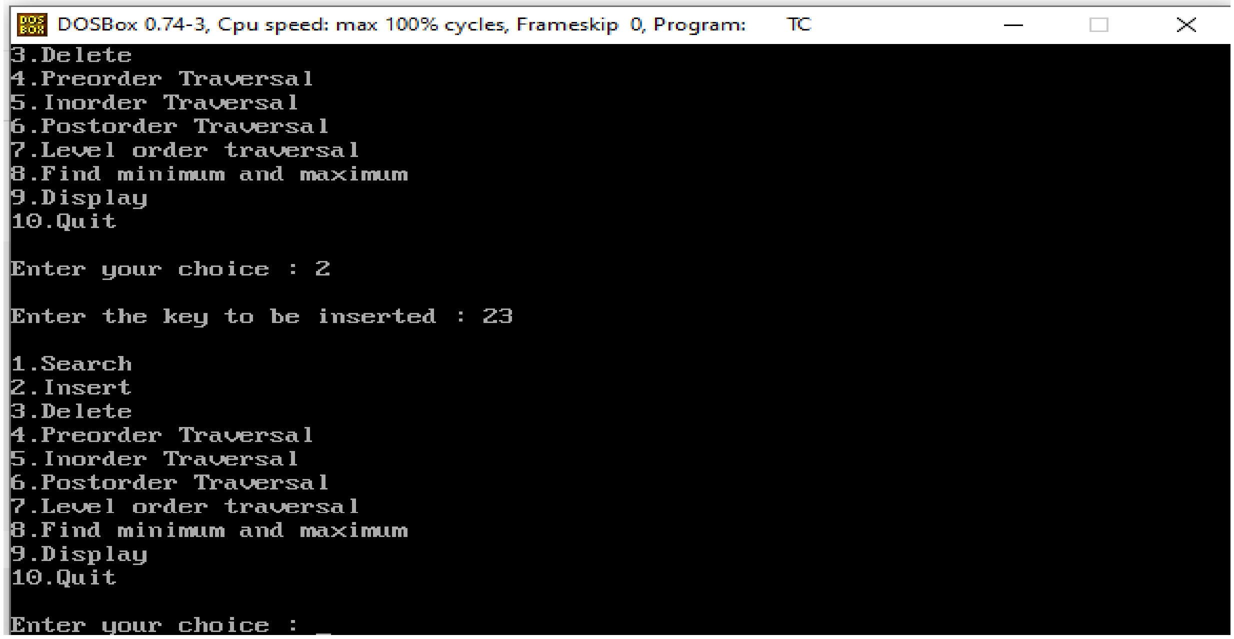
```
3.Delete
4.Preorder Traversal
5.Inorder Traversal
6.Postorder Traversal
7.Level order traversal
8.Find minimum and maximum
9.Display
10.Quit

Enter your choice : 2

Enter the key to be inserted : 90

1.Search
2.Insert
3.Delete
4.Preorder Traversal
5.Inorder Traversal
6.Postorder Traversal
7.Level order traversal
8.Find minimum and maximum
9.Display
10.Quit

Enter your choice :
```


DOSBox 0.74-3, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

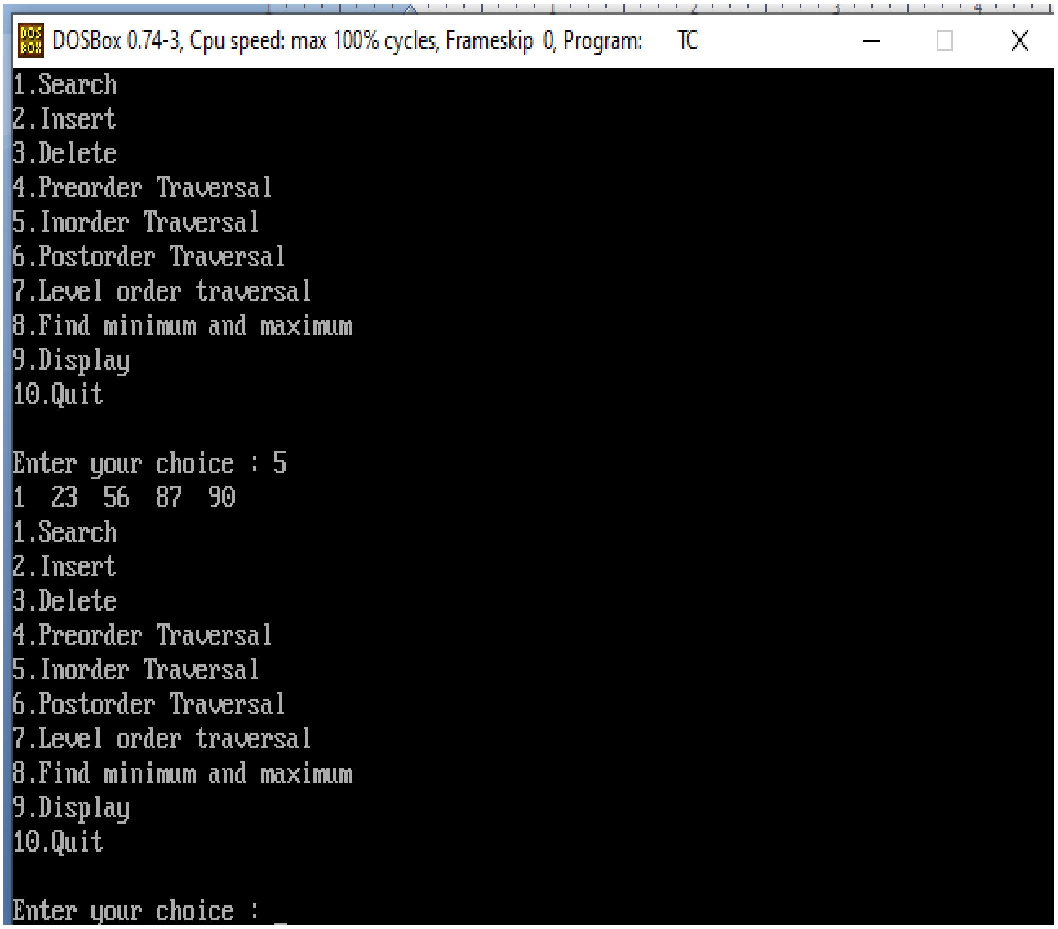
```
3.Delete
4.Preorder Traversal
5.Inorder Traversal
6.Postorder Traversal
7.Level order traversal
8.Find minimum and maximum
9.Display
10.Quit

Enter your choice : 2

Enter the key to be inserted : 23

1.Search
2.Insert
3.Delete
4.Preorder Traversal
5.Inorder Traversal
6.Postorder Traversal
7.Level order traversal
8.Find minimum and maximum
9.Display
10.Quit

Enter your choice : _
```


DOSBox 0.74-3, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
1.Search
2.Insert
3.Delete
4.Preorder Traversal
5.Inorder Traversal
6.Postorder Traversal
7.Level order traversal
8.Find minimum and maximum
9.Display
10.Quit

Enter your choice : 5
1  23  56  87  90
1.Search
2.Insert
3.Delete
4.Preorder Traversal
5.Inorder Traversal
6.Postorder Traversal
7.Level order traversal
8.Find minimum and maximum
9.Display
10.Quit

Enter your choice : _
```