1: Write a C Program to perform following operations on a threaded binary tree

* DISPLAY
* INSERT
* DELETE

Program:

#include <stdio.h>

#include <stdlib.h>

typedef enum {false,true} boolean;

struct node \*in\_succ(struct node \*p);

struct node \*in\_pred(struct node \*p);

struct node \*insert(struct node \*root, int ikey);

struct node \*del(struct node \*root, int dkey);

struct node \*case\_a(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\_c(struct node \*root, struct node \*par,struct node \*ptr);

void inorder( struct node \*root);

void preorder( struct node \*root);

struct node

{

struct node \*left;

boolean lthread;

int info;

boolean rthread;

struct node \*right;

};

int main( )

{

int choice,num;

struct node \*root=NULL;

while(1)

{

printf("\n");

printf("1.Insert\n");

printf("2.Delete\n");

printf("3.Inorder Traversal\n");

printf("4.Preorder Traversal\n");

printf("5.Quit\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("Enter the number to be inserted : ");

scanf("%d",&num);

root = insert(root,num);

break;

case 2:

printf("Enter the number to be deleted : ");

scanf("%d",&num);

root = del(root,num);

break;

case 3:

inorder(root);

break;

case 4:

preorder(root);

break;

case 5:

exit(1);

default:

printf("Wrong choice\n");

}

}

}

struct node \*insert(struct node \*root, int ikey)

{

struct node \*tmp,\*par,\*ptr;

int found=0;

ptr = root;

par = NULL;

while( ptr!=NULL )

{

if( ikey == ptr->info)

{

found =1;

break;

}

par = ptr;

if(ikey<ptr->info)

{

if(ptr->lthread == false)

ptr = ptr->left;

else

break;

}

else

{

if(ptr->rthread == false)

ptr = ptr->right;

else

break;

}

}

if(found)

printf("Duplicate key");

else

{

tmp=(struct node \*)malloc(sizeof(struct node));

tmp->info=ikey;

tmp->lthread = true;

tmp->rthread = true;

if(par==NULL)

{

root=tmp;

tmp->left=NULL;

tmp->right=NULL;

}

else if( ikey< par->info )

{

tmp->left=par->left;

tmp->right=par;

par->lthread=false;

par->left=tmp;

}

else

{

tmp->left=par;

tmp->right=par->right;

par->rthread=false;

par->right=tmp;

}

}

return root;

}

struct node \*del(struct node \*root, int dkey)

{

struct node \*par,\*ptr;

int found=0;

ptr = root;

par = NULL;

while( ptr!=NULL)

{

if( dkey == ptr->info)

{

found =1;

break;

}

par = ptr;

if(dkey<ptr->info)

{

if(ptr->lthread == false)

ptr = ptr->left;

else

break;

}

else

{

if(ptr->rthread == false)

ptr = ptr->right;

else

break;

}

}

if(found==0)

printf("dkey not present in tree");

else if(ptr->lthread==false &&ptr->rthread==false)/\*2 children\*/

root = case\_c(root,par,ptr);

else if(ptr->lthread==false )/\*only left child\*/

root = case\_b(root, par,ptr);

else if(ptr->rthread==false)/\*only right child\*/

root = case\_b(root, par,ptr);

else /\*no child\*/

root = case\_a(root,par,ptr);

return root;

}

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->left)

{

par->lthread=true;

par->left=ptr->left;

}

else

{

par->rthread=true;

par->right=ptr->right;

}

free(ptr);

return root;

}

struct node \*case\_b(struct node \*root,struct node \*par,struct node \*ptr)

{

struct node \*child,\*s,\*p; /\*Initialize child\*/

if(ptr->lthread==false) /\*node to be deleted has left child \*/

child=ptr->left;

else /\*node to be deleted has right child \*/

child=ptr->right;

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

root=child;

else if( ptr==par->left) /\*node is left child of its parent\*/

par->left=child;

else /\*node is right child of its parent\*/

par->right=child;

s=in\_succ(ptr);

p=in\_pred(ptr);

if(ptr->lthread==false) /\*if ptr has left subtree \*/

p->right=s;

else

{

if(ptr->rthread==false) /\*if ptr has right subtree\*/

s->left=p;

}

free(ptr);

return root;

}

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->right;

while(succ->left!=NULL)

{

parsucc = succ;

succ = succ->left;

}

ptr->info = succ->info;

if(succ->lthread==true &&succ->rthread==true)

root = case\_a(root, parsucc,succ);

else

root = case\_b(root, parsucc,succ);

return root;

}

struct node \*in\_succ(struct node \*ptr)

{

if(ptr->rthread==true)

return ptr->right;

else

{

ptr=ptr->right;

while(ptr->lthread==false)

ptr=ptr->left;

return ptr;

}

}

struct node \*in\_pred(struct node \*ptr)

{

if(ptr->lthread==true)

return ptr->left; else

{

ptr=ptr->left;

while(ptr->rthread==false)

ptr=ptr->right;

return ptr;

}

}

void inorder( struct node \*root)

{

struct node \*ptr; if(root == NULL )

{

printf("Tree is empty");

return;

}

ptr=root; /\*Find the leftmost node \*/ while(ptr->lthread==false)

ptr=ptr->left;

while( ptr!=NULL )

{

printf("%d ",ptr->info);

ptr=in\_succ(ptr);

}

printf("\n\n");

}

void preorder(struct node \*root )

{

struct node \*ptr; if(root==NULL)

{

printf("Tree is empty");

return;

}

ptr=root;

while(ptr!=NULL)

{

printf("%d ",ptr->info);

if(ptr->lthread==false)

ptr=ptr->left;

else if(ptr->rthread==false)

ptr=ptr->right;

else

{

while(ptr!=NULL &&ptr->rthread==true)

ptr=ptr->right;

if(ptr!=NULL)

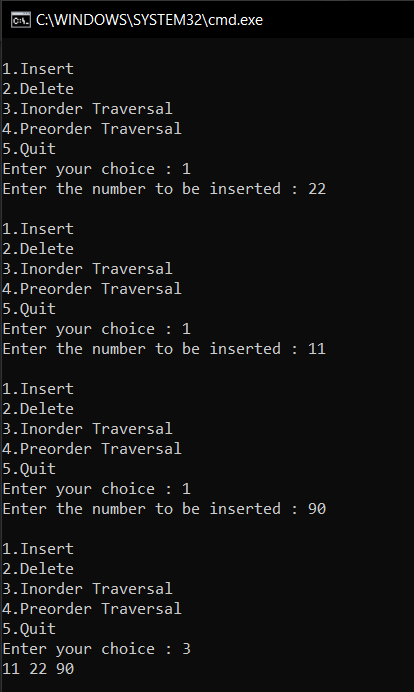
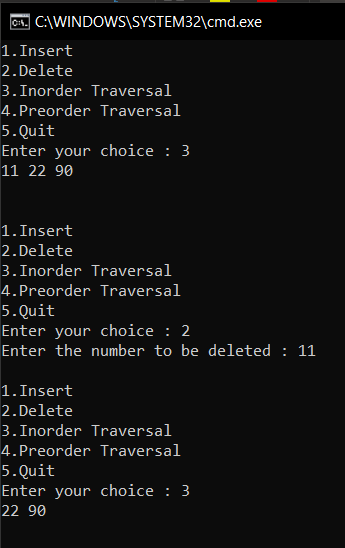
ptr=ptr->right;

}

}

}

Output:

2. Write a program to implement following traversals in an in-threaded binary tree

* In order
* Pre Order

Program:

#include <stdio.h>

#include <stdlib.h>

typedef enum {false,true} boolean;

struct node \*in\_succ(struct node \*p);

struct node \*in\_pred(struct node \*p);

struct node \*insert(struct node \*root, int ikey);

struct node \*del(struct node \*root, int dkey);

struct node \*case\_a(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\_c(struct node \*root, struct node \*par,struct node \*ptr);

void inorder( struct node \*root);

void preorder( struct node \*root);

struct node

{

struct node \*left;

boolean lthread;

int info;

boolean rthread;

struct node \*right;

};

int main( )

{

int choice,num;

struct node \*root=NULL;

while(1)

{

printf("\n");

printf("1.Insert\n");

printf("2.Inorder Traversal\n");

printf("3.Preorder Traversal\n");

printf("4.Quit\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("Enter the number to be inserted : ");

scanf("%d",&num);

root = insert(root,num);

break;

case 2:

inorder(root);

break;

case 3:

preorder(root);

break;

case 4:

exit(1);

default:

printf("Wrong choice\n");

}

}

}

struct node \*insert(struct node \*root, int ikey)

{

struct node \*tmp,\*par,\*ptr;

int found=0;

ptr = root;

par = NULL;

while( ptr!=NULL )

{

if( ikey == ptr->info)

{

found =1;

break;

}

par = ptr;

if(ikey<ptr->info)

{

if(ptr->lthread == false)

ptr = ptr->left;

else

break;

}

else

{

if(ptr->rthread == false)

ptr = ptr->right;

else

break;

}

}

if(found)

printf("Duplicate key");

else

{

tmp=(struct node \*)malloc(sizeof(struct node));

tmp->info=ikey;

tmp->lthread = true;

tmp->rthread = true;

if(par==NULL)

{

root=tmp;

tmp->left=NULL;

tmp->right=NULL;

}

else if( ikey< par->info )

{

tmp->left=par->left;

tmp->right=par;

par->lthread=false;

par->left=tmp;

}

else

{

tmp->left=par;

tmp->right=par->right;

par->rthread=false;

par->right=tmp;

}

}

return root;

}

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->left)

{

par->lthread=true;

par->left=ptr->left;

}

else

{

par->rthread=true;

par->right=ptr->right;

}

free(ptr);

return root;

}

struct node \*case\_b(struct node \*root,struct node \*par,struct node \*ptr)

{

struct node \*child,\*s,\*p; /\*Initialize child\*/

if(ptr->lthread==false) /\*node to be deleted has left child \*/

child=ptr->left;

else /\*node to be deleted has right child \*/

child=ptr->right;

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

root=child;

else if( ptr==par->left) /\*node is left child of its parent\*/

par->left=child;

else /\*node is right child of its parent\*/

par->right=child;

s=in\_succ(ptr);

p=in\_pred(ptr);

if(ptr->lthread==false) /\*if ptr has left subtree \*/

p->right=s;

else

{

if(ptr->rthread==false) /\*if ptr has right subtree\*/

s->left=p;

}

free(ptr);

return root;

}

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->right;

while(succ->left!=NULL)

{

parsucc = succ;

succ = succ->left;

}

ptr->info = succ->info;

if(succ->lthread==true &&succ->rthread==true)

root = case\_a(root, parsucc,succ);

else

root = case\_b(root, parsucc,succ);

return root;

}

struct node \*in\_succ(struct node \*ptr)

{

if(ptr->rthread==true)

return ptr->right;

else

{

ptr=ptr->right;

while(ptr->lthread==false)

ptr=ptr->left;

return ptr;

}

}

struct node \*in\_pred(struct node \*ptr)

{

if(ptr->lthread==true)

return ptr->left; else

{

ptr=ptr->left;

while(ptr->rthread==false)

ptr=ptr->right;

return ptr;

}

}

void inorder( struct node \*root)

{

struct node \*ptr; if(root == NULL )

{

printf("Tree is empty");

return;

}

ptr=root; /\*Find the leftmost node \*/ while(ptr->lthread==false)

ptr=ptr->left;

while( ptr!=NULL )

{

printf("%d ",ptr->info);

ptr=in\_succ(ptr);

}

}

void preorder(struct node \*root )

{

struct node \*ptr; if(root==NULL)

{

printf("Tree is empty");

return;

}

ptr=root;

while(ptr!=NULL)

{

printf("%d ",ptr->info);

if(ptr->lthread==false)

ptr=ptr->left;

else if(ptr->rthread==false)

ptr=ptr->right;

else

{

while(ptr!=NULL &&ptr->rthread==true)

ptr=ptr->right;

if(ptr!=NULL)

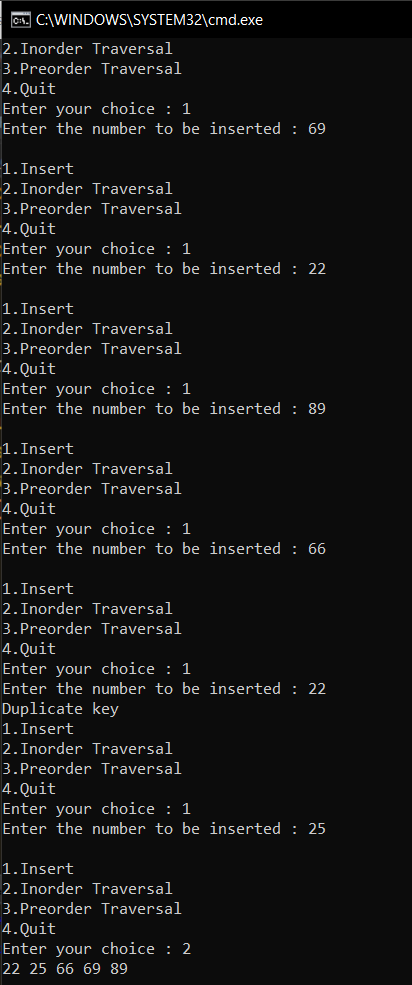
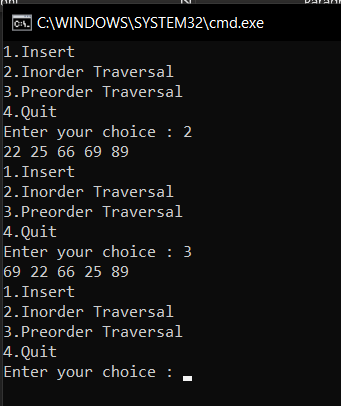
ptr=ptr->right;

}

}

}

In order and Pre-order output:

3. Write a program to implement following traversals in AVL trees

* Insertion in Left sub tree
* Insertion in Right Sub Tree

Program:

#include <stdio.h>

#include <stdlib.h>

#define FALSE 0

#define TRUE 1

struct node

{

struct node \*lchild;

int info; struct node \*rchild;

int balance;

};

void inorder(struct node \*ptr);

struct node \*RotateLeft(struct node \*pptr);

struct node \*RotateRight(struct node \*pptr);

struct node \*insert(struct node \*pptr, int ikey);

struct node \*insert\_left\_check(struct node \*pptr, int \*ptaller);

struct node \*insert\_right\_check(struct node \*pptr, int \*ptaller);

struct node \*insert\_LeftBalance(struct node \*pptr);

struct node \*insert\_RightBalance(struct node \*pptr);

int main()

{

int choice,key;

struct node \*root = NULL;

while(1)

{

printf("\n");

printf("1.Insert into subtree\n");

printf("2.Inorder Traversal\n");

printf("3.Quit\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("Enter the key to be inserted : ");

scanf("%d",&key);

root = insert(root,key);

break;

case 2:

inorder(root);

break;

case 3:

exit(1);

default:

printf("Wrong choice\n");

}

}

}

struct node \*insert(struct node \*pptr, int ikey)

{

static int taller;

if(pptr==NULL) /\*Base case\*/

{

pptr = (struct node \*) malloc(sizeof(struct node));

pptr->info = ikey;

pptr->lchild = NULL;

pptr->rchild = NULL;

pptr->balance = 0;

taller = TRUE;

}

else if(ikey<pptr->info) /\*Insertion in left subtree\*/

{

pptr->lchild = insert(pptr->lchild, ikey);

if(taller==TRUE)

pptr = insert\_left\_check( pptr, &taller );

}

else if(ikey>pptr->info) /\*Insertion in right subtree \*/

{

pptr->rchild = insert(pptr->rchild, ikey);

if(taller==TRUE)

pptr = insert\_right\_check(pptr, &taller);

}

else /\*Base Case\*/

{

printf("Duplicate key\n");

taller = FALSE;

}

return pptr;

}

struct node \*insert\_left\_check(struct node \*pptr, int \*ptaller )

{

switch(pptr->balance)

{

case 0: /\* Case L\_A : was balanced \*/

pptr->balance = 1; /\* now left heavy \*/

break;

case -1: /\* Case L\_B: was right heavy \*/

pptr->balance = 0; /\* now balanced \*/

\*ptaller = FALSE;

break;

case 1: /\* Case L\_C: was left heavy \*/ pptr = insert\_LeftBalance(pptr); /\* Left Balancing \*/

\*ptaller = FALSE;

}

return pptr;

}

struct node \*insert\_right\_check(struct node \*pptr, int \*ptaller )

{

switch(pptr->balance)

{

case 0: /\* Case R\_A : was balanced \*/

pptr->balance = -1; /\* now right heavy \*/

break;

case 1: /\* Case R\_B : was left heavy \*/

pptr->balance = 0; /\* now balanced \*/

\*ptaller = FALSE;

break;

case -1: /\* Case R\_C: Right heavy \*/

pptr = insert\_RightBalance(pptr); /\* Right Balancing \*/

\*ptaller = FALSE;

}

return pptr;

}

struct node \*insert\_LeftBalance(struct node \*pptr)

{

struct node \*aptr, \*bptr;

aptr = pptr->lchild;

if(aptr->balance == 1) /\* Case L\_C1 : Insertion in AL \*/

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateRight(pptr);

}

else /\* Case L\_C2 : Insertion in AR \*/

{

bptr = aptr->rchild;

switch(bptr->balance)

{

case -1: /\* Case L\_C2a : Insertion in BR \*/

pptr->balance = 0;

aptr->balance = 1;

break;

case 1: /\* Case L\_C2b : Insertion in BL \*/

pptr->balance = -1;

aptr->balance = 0;

break;

case 0: /\* Case L\_C2c : B is the newly inserted node \*/

pptr->balance = 0;

aptr->balance = 0;

}

bptr->balance = 0;

pptr->lchild = RotateLeft(aptr);

pptr = RotateRight(pptr);

}

return pptr;

}

struct node \*insert\_RightBalance(struct node \*pptr)

{

struct node \*aptr, \*bptr;

aptr = pptr->rchild;

if(aptr->balance == -1) /\* Case R\_C1 : Insertion in AR \*/

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateLeft(pptr);

}

else /\* Case R\_C2 : Insertion in AL \*/

{

bptr = aptr->lchild;

switch(bptr->balance)

{

case -1: /\* Case R\_C2a : Insertion in BR \*/

pptr->balance = 1;

aptr->balance = 0;

break;

case 1: /\* Case R\_C2b : Insertion in BL \*/

pptr->balance = 0;

aptr->balance = -1;

break;

case 0: /\* Case R\_C2c : B is the newly inserted node \*/

pptr->balance = 0;

aptr->balance = 0;

}

bptr->balance = 0;

pptr->rchild = RotateRight(aptr);

pptr = RotateLeft(pptr);

}

return pptr;

}

struct node \*RotateLeft(struct node \*pptr)

{

struct node \*aptr;

aptr = pptr->rchild; /\*A is right child of P\*/

pptr->rchild = aptr->lchild; /\*Left child of A becomes right child of P \*/

aptr->lchild = pptr; /\*P becomes left child of A\*/

return aptr; /\*A is the new root of the subtree initially rooted at P\*/

}

struct node \*RotateRight(struct node \*pptr)

{

struct node \*aptr;

aptr = pptr->lchild; /\*A is left child of P \*/

pptr->lchild = aptr->rchild; /\*Right child of A becomes left child of P\*/

aptr->rchild = pptr; /\*P becomes right child of A\*/

return aptr; /\*A is the new root of the subtree initially rooted at P\*/

}

void inorder(struct node \*ptr)

{

if(ptr!=NULL)

{

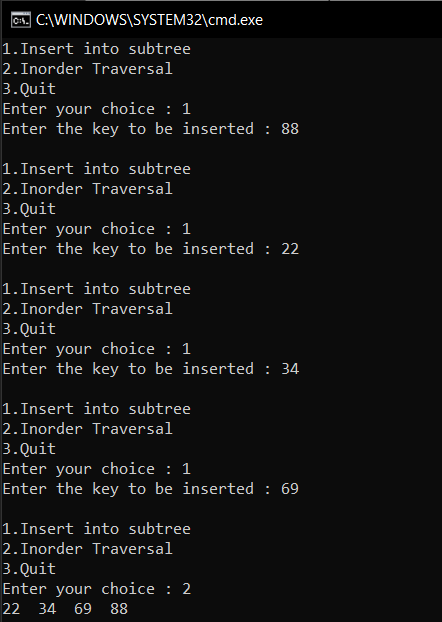
inorder(ptr->lchild);

printf("%d ",ptr->info);

inorder(ptr->rchild);

}

}



4. Write a program to implement following traversals in AVL trees

* Deletion in Left sub tree
* Deletion in Right Sub Tree

Program:

#include <stdio.h>

#include <stdlib.h>

#define FALSE 0

#define TRUE 1

struct node

{

struct node \*lchild;

int info; struct node \*rchild;

int balance;

};

void inorder(struct node \*ptr);

struct node \*RotateLeft(struct node \*pptr);

struct node \*RotateRight(struct node \*pptr);

struct node \*insert(struct node \*pptr, int ikey);

struct node \*insert\_left\_check(struct node \*pptr, int \*ptaller);

struct node \*insert\_right\_check(struct node \*pptr, int \*ptaller);

struct node \*insert\_LeftBalance(struct node \*pptr);

struct node \*insert\_RightBalance(struct node \*pptr);

struct node \*del(struct node \*pptr, int dkey);

struct node \*del\_left\_check(struct node \*pptr, int \*pshorter);

struct node \*del\_right\_check(struct node \*pptr, int \*pshorter);

struct node \*del\_LeftBalance(struct node \*pptr,int \*pshorter);

struct node \*del\_RightBalance(struct node \*pptr,int \*pshorter);

int main()

{

int choice,key;

struct node \*root = NULL;

while(1)

{

printf("\n");

printf("1.Insert into subtree\n");

printf("2.Delete from sub tree\n");

printf("3.Inorder Traversal\n");

printf("4.Quit\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("Enter the key to be inserted : ");

scanf("%d",&key);

root = insert(root,key);

break;

case 2:

printf("Enter the key to be deleted : ");

scanf("%d",&key);

root = del(root,key);

break;

case 3:

inorder(root);

break;

case 4:

exit(1);

default:

printf("Wrong choice\n");

}

}

}

struct node \*insert(struct node \*pptr, int ikey)

{

static int taller;

if(pptr==NULL) /\*Base case\*/

{

pptr = (struct node \*) malloc(sizeof(struct node));

pptr->info = ikey;

pptr->lchild = NULL;

pptr->rchild = NULL;

pptr->balance = 0;

taller = TRUE;

}

else if(ikey<pptr->info) /\*Insertion in left subtree\*/

{

pptr->lchild = insert(pptr->lchild, ikey);

if(taller==TRUE)

pptr = insert\_left\_check( pptr, &taller );

}

else if(ikey>pptr->info) /\*Insertion in right subtree \*/

{

pptr->rchild = insert(pptr->rchild, ikey);

if(taller==TRUE)

pptr = insert\_right\_check(pptr, &taller);

}

else /\*Base Case\*/

{

printf("Duplicate key\n");

taller = FALSE;

}

return pptr;

}

struct node \*insert\_left\_check(struct node \*pptr, int \*ptaller )

{

switch(pptr->balance)

{

case 0: /\* Case L\_A : was balanced \*/

pptr->balance = 1; /\* now left heavy \*/

break;

case -1: /\* Case L\_B: was right heavy \*/

pptr->balance = 0; /\* now balanced \*/

\*ptaller = FALSE;

break;

case 1: /\* Case L\_C: was left heavy \*/ pptr = insert\_LeftBalance(pptr); /\* Left Balancing \*/

\*ptaller = FALSE;

}

return pptr;

}

struct node \*insert\_right\_check(struct node \*pptr, int \*ptaller )

{

switch(pptr->balance)

{

case 0: /\* Case R\_A : was balanced \*/

pptr->balance = -1; /\* now right heavy \*/

break;

case 1: /\* Case R\_B : was left heavy \*/

pptr->balance = 0; /\* now balanced \*/

\*ptaller = FALSE;

break;

case -1: /\* Case R\_C: Right heavy \*/

pptr = insert\_RightBalance(pptr); /\* Right Balancing \*/

\*ptaller = FALSE;

}

return pptr;

}

struct node \*insert\_LeftBalance(struct node \*pptr)

{

struct node \*aptr, \*bptr;

aptr = pptr->lchild;

if(aptr->balance == 1) /\* Case L\_C1 : Insertion in AL \*/

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateRight(pptr);

}

else /\* Case L\_C2 : Insertion in AR \*/

{

bptr = aptr->rchild;

switch(bptr->balance)

{

case -1: /\* Case L\_C2a : Insertion in BR \*/

pptr->balance = 0;

aptr->balance = 1;

break;

case 1: /\* Case L\_C2b : Insertion in BL \*/

pptr->balance = -1;

aptr->balance = 0;

break;

case 0: /\* Case L\_C2c : B is the newly inserted node \*/

pptr->balance = 0;

aptr->balance = 0;

}

bptr->balance = 0;

pptr->lchild = RotateLeft(aptr);

pptr = RotateRight(pptr);

}

return pptr;

}

struct node \*insert\_RightBalance(struct node \*pptr)

{

struct node \*aptr, \*bptr;

aptr = pptr->rchild;

if(aptr->balance == -1) /\* Case R\_C1 : Insertion in AR \*/

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateLeft(pptr);

}

else /\* Case R\_C2 : Insertion in AL \*/

{

bptr = aptr->lchild;

switch(bptr->balance)

{

case -1: /\* Case R\_C2a : Insertion in BR \*/

pptr->balance = 1;

aptr->balance = 0;

break;

case 1: /\* Case R\_C2b : Insertion in BL \*/

pptr->balance = 0;

aptr->balance = -1;

break;

case 0: /\* Case R\_C2c : B is the newly inserted node \*/

pptr->balance = 0;

aptr->balance = 0;

}

bptr->balance = 0;

pptr->rchild = RotateRight(aptr);

pptr = RotateLeft(pptr);

}

return pptr;

}

struct node \*RotateLeft(struct node \*pptr)

{

struct node \*aptr;

aptr = pptr->rchild; /\*A is right child of P\*/

pptr->rchild = aptr->lchild; /\*Left child of A becomes right child of P \*/

aptr->lchild = pptr; /\*P becomes left child of A\*/

return aptr; /\*A is the new root of the subtree initially rooted at P\*/

}

struct node \*RotateRight(struct node \*pptr)

{

struct node \*aptr;

aptr = pptr->lchild; /\*A is left child of P \*/

pptr->lchild = aptr->rchild; /\*Right child of A becomes left child of P\*/

aptr->rchild = pptr; /\*P becomes right child of A\*/

return aptr; /\*A is the new root of the subtree initially rooted at P\*/

}

struct node \*del(struct node \*pptr, int dkey)

{

struct node \*tmp, \*succ;

static int shorter;

if( pptr == NULL) /\*Base Case\*/

{

printf("Key not present \n");

shorter = FALSE;

return(pptr);

}

if( dkey<pptr->info )

{

pptr->lchild = del(pptr->lchild, dkey);

if(shorter == TRUE)

pptr = del\_left\_check(pptr, &shorter);

}

else if( dkey>pptr->info )

{

pptr->rchild = del(pptr->rchild, dkey);

if(shorter==TRUE)

pptr = del\_right\_check(pptr, &shorter);

}

else /\* dkey == pptr->info, Base Case\*/

{

/\*pptr has 2 children\*/

if( pptr->lchild!=NULL &&pptr->rchild!=NULL )

{

succ = pptr->rchild;

while(succ->lchild)

succ = succ->lchild;

pptr->info = succ->info;

pptr->rchild = del(pptr->rchild, succ->info);

if( shorter == TRUE )

pptr = del\_right\_check(pptr, &shorter);

}

else

{

tmp = pptr;

if( pptr->lchild != NULL ) /\*only left child\*/

pptr = pptr->lchild;

else if( pptr->rchild != NULL) /\*only right child\*/

pptr = pptr->rchild;

else /\* no children \*/

pptr = NULL;

free(tmp);

shorter = TRUE;

}

}

return pptr;

}

struct node \*del\_left\_check(struct node \*pptr, int \*pshorter)

{

switch(pptr->balance)

{

case 0: /\* Case L\_A : was balanced \*/

pptr->balance = -1; /\* now right heavy \*/

\*pshorter = FALSE;

break;

case 1: /\* Case L\_B : was left heavy \*/

pptr->balance = 0; /\* now balanced \*/

break;

case -1: /\* Case L\_C : was right heavy \*/

pptr = del\_RightBalance(pptr, pshorter); /\*Right Balancing\*/

}

return pptr;

}

struct node \*del\_right\_check(struct node \*pptr, int \*pshorter)

{

switch(pptr->balance)

{

case 0: /\* Case R\_A : was balanced \*/ pptr->balance = 1; /\* now left heavy \*/

\*pshorter = FALSE;

break;

case -1: /\* Case R\_B : was right heavy \*/

pptr->balance = 0; /\* now balanced \*/

break;

case 1: /\* Case R\_C : was left heavy \*/ pptr = del\_LeftBalance(pptr, pshorter ); /\*Left Balancing\*/

}

return pptr;

}

struct node \*del\_LeftBalance(struct node \*pptr,int \*pshorter)

{

struct node \*aptr, \*bptr;

aptr = pptr->lchild;

if( aptr->balance == 0) /\* Case R\_C1 \*/

{

pptr->balance = 1;

aptr->balance = -1;

\*pshorter = FALSE;

pptr = RotateRight(pptr);

}

else if(aptr->balance == 1 ) /\* Case R\_C2 \*/

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateRight(pptr);

}

else /\* Case R\_C3 \*/

{

bptr = aptr->rchild;

switch(bptr->balance)

{

case 0: /\* Case R\_C3a \*/

pptr->balance = 0;

aptr->balance = 0;

break;

case 1: /\* Case R\_C3b \*/

pptr->balance = -1;

aptr->balance = 0;

break;

case -1: /\* Case R\_C3c \*/

pptr->balance = 0;

aptr->balance = 1;

}

bptr->balance = 0;

pptr->lchild = RotateLeft(aptr);

pptr = RotateRight(pptr);

}

return pptr;

}

struct node \*del\_RightBalance(struct node \*pptr,int \*pshorter)

{

struct node \*aptr, \*bptr;

aptr = pptr->rchild;

if (aptr->balance == 0) /\* Case L\_C1 \*/

{

pptr->balance = -1;

aptr->balance = 1;

\*pshorter = FALSE;

pptr = RotateLeft(pptr);

}

else if(aptr->balance == -1 ) /\* Case L\_C2 \*/

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateLeft(pptr);

}

else /\* Case L\_C3 \*/

{

bptr = aptr->lchild;

switch(bptr->balance)

{

case 0: /\* Case L\_C3a \*/

pptr->balance = 0;

aptr->balance = 0;

break;

case 1: /\* Case L\_C3b \*/

pptr->balance = 0;

aptr->balance = -1;

break;

case -1: /\* Case L\_C3c \*/

pptr->balance = 1;

aptr->balance = 0;

}

bptr->balance = 0;

pptr->rchild = RotateRight(aptr);

pptr = RotateLeft(pptr);

}

return pptr;

}

void inorder(struct node \*ptr)

{

if(ptr!=NULL)

{

inorder(ptr->lchild);

printf("%d ",ptr->info);

inorder(ptr->rchild);

}

}

