**CODE:-**

#include <stdio.h>

#include <stdlib.h>

#define FALSE 0

#define TRUE 1

struct node

{

struct node \*lchild;

int info;

int balance;

struct node \*rchild;

};

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

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

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

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

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

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

{

struct node \*aptr;

aptr = pptr->lchild;

pptr->lchild = aptr->rchild;

aptr->rchild = pptr;

return aptr;

}

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

{

struct node \*aptr;

aptr = pptr->rchild;

pptr->rchild = aptr->lchild;

aptr->lchild = pptr;

return aptr;

}

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

{

static int taller;

if (pptr == NULL)

{

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)

{

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

if (taller == TRUE)

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

}

else if (ikey > pptr->info)

{

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

if (taller == TRUE)

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

}

else

{

printf("Duplicate key.\n");

taller = FALSE;

}

return pptr;

}

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

{

switch (pptr->balance)

{

case 0:

pptr->balance = 1;

break;

case -1:

pptr->balance = 0;

\*ptaller = FALSE;

break;

case 1:

pptr = insert\_LeftBalance(pptr);

\*ptaller = FALSE;

}

return pptr;

}

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

{

struct node \*aptr, \*bptr;

aptr = pptr->lchild;

if (aptr->balance == 1)

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateRight(pptr);

}

else

{

bptr = aptr->rchild;

switch (bptr->balance)

{

case -1:

pptr->balance = 0;

aptr->balance = 1;

break;

case 1:

pptr->balance = -1;

aptr->balance = 0;

break;

case 0:

pptr->balance = 0;

aptr->balance = 0;

}

bptr->balance = 0;

pptr->lchild = RotateLeft(aptr);

pptr = RotateRight(pptr);

}

return pptr;

}

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

{

switch (pptr->balance)

{

case 0:

pptr->balance = -1;

break;

case 1:

pptr->balance = 0;

\*ptaller = FALSE;

break;

case -1:

pptr = insert\_RightBalance(pptr);

\*ptaller = FALSE;

}

return pptr;

}

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

{

struct node \*aptr, \*bptr;

aptr = pptr->rchild;

if (aptr->balance == -1)

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateLeft(pptr);

}

else

{

bptr = aptr->lchild;

switch (bptr->balance)

{

case -1:

pptr->balance = 1;

aptr->balance = 0;

break;

case 1:

pptr->balance = 0;

aptr->balance = -1;

break;

case 0:

pptr->balance = 0;

aptr->balance = 0;

}

bptr->balance = 0;

pptr->rchild = RotateRight(aptr);

pptr = RotateLeft(pptr);

}

return pptr;

}

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

{

static int shorter;

struct node \*tmp, \*succ;

if (pptr == NULL)

{

printf("Key not found.\n");

shorter = FALSE;

return pptr;

}

else 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

{

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

{

succ = pptr->rchild;

while (succ->lchild != NULL)

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)

pptr = pptr->lchild;

else if (pptr->rchild != NULL)

pptr = pptr->rchild;

else

pptr = NULL;

free(tmp);

shorter = TRUE;

}

}

return pptr;

}

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

{

switch (pptr->balance)

{

case 0:

pptr->balance = -1;

\*pshorter = FALSE;

break;

case 1:

pptr->balance = 0;

break;

case -1:

pptr = del\_RightBalance(pptr, pshorter);

}

return pptr;

}

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

{

struct node \*aptr, \*bptr;

aptr = pptr->rchild;

if (aptr->balance == 0)

{

pptr->balance = -1;

aptr->balance = 1;

\*pshorter = FALSE;

pptr = RotateLeft(pptr);

}

else if (aptr->balance == -1)

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateLeft(pptr);

}

else

{

bptr = aptr->lchild;

switch (bptr->balance)

{

case 0:

pptr->balance = 0;

aptr->balance = 0;

break;

case 1:

pptr->balance = 0;

aptr->balance = -1;

break;

case -1:

pptr->balance = 1;

aptr->balance = 0;

}

bptr->balance = 0;

pptr->rchild = RotateRight(aptr);

pptr = RotateLeft(pptr);

}

return pptr;

}

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

{

switch (pptr->balance)

{

case 0:

pptr->balance = 1;

\*pshorter = FALSE;

break;

case -1:

pptr->balance = 0;

break;

case 1:

pptr = del\_LeftBalance(pptr, pshorter);

}

return pptr;

}

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

{

struct node \*aptr, \*bptr;

aptr = pptr->lchild;

if (aptr->balance == 0)

{

pptr->balance = 1;

aptr->balance = -1;

\*pshorter = FALSE;

pptr = RotateRight(pptr);

}

else if (aptr->balance == 1)

{

pptr->balance = 0;

aptr->balance = 0;

pptr = RotateRight(pptr);

}

else

{

bptr = aptr->rchild;

switch (bptr->balance)

{

case 0:

pptr->balance = 0;

aptr->balance = 0;

break;

case 1:

pptr->balance = -1;

aptr->balance = 0;

break;

case -1:

pptr->balance = 0;

aptr->balance = 1;

}

bptr->balance = 0;

pptr->lchild = RotateLeft(aptr);

pptr = RotateRight(pptr);

}

return pptr;

}

struct node \*search(struct node \*root, int key)

{

if (root == NULL)

{

return NULL;

}

if (key == root->info)

{

return root;

}

else if (key < root->info)

{

return search(root->lchild, key);

}

else

{

return search(root->rchild, key);

}

}

static int max, min, count = 0;

void reverse\_inorder(struct node \*root, int m)

{

if (root != NULL)

min = root->info;

if (root == NULL)

{

return;

}

reverse\_inorder(root->rchild, m);

if (count == 0)

{

max = root->info;

count++;

}

if (m == 1)

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

reverse\_inorder(root->lchild, m);

}

int height(struct node \*ptr)

{

int h\_left, h\_right;

if (ptr == NULL)

return 0;

h\_left = height(ptr->lchild);

h\_right = height(ptr->rchild);

if (h\_left > h\_right)

return 1 + h\_left;

else

return 1 + h\_right;

}

int getWidth(struct node \*root, int level)

{

if (root == NULL)

return 0;

if (level == 1)

return 1;

else if (level > 1)

return getWidth(root->lchild, level - 1) + getWidth(root->rchild, level - 1);

}

int getMaxWidth(struct node \*root)

{

int maxWidth = 0;

int width;

int h = height(root);

int i;

for (i = 1; i <= h; i++)

{

width = getWidth(root, i);

if (width > maxWidth)

maxWidth = width;

}

return maxWidth;

}

int main()

{

struct node \*root = NULL, \*tp;

int ch, elem;

int A[15] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

for (int i = 0; i < 15; i++)

root = insert(root, A[i]);

while (1)

{

printf("Enter 1 to insert new element.\n");

printf("Enter 2 to delete existing element.\n");

printf("Enter 3 to search an element.\n");

printf("Enter 4 to list all elements in descending order.\n");

printf("Enter 5 to find maximum element in the tree.\n");

printf("Enter 6 to find minimum element in the tree.\n");

printf("Enter 7 to find width of the tree.\n");

printf("Enter 8 to exit.\n");

scanf("%d", &ch);

switch (ch)

{

case 1:

printf("Enter element to be inserted.\n");

scanf("%d", &elem);

root = insert(root, elem);

break;

case 2:

printf("Enter element to be deleted.\n");

scanf("%d", &elem);

root = del(root, elem);

break;

case 3:

printf("Enter element to be searched.\n");

scanf("%d", &elem);

tp = search(root, elem);

if (tp == NULL)

printf("Element not found.\n");

else

printf("Element found in tree.\n");

break;

case 4:

printf("Elements of tree in descending order are:\n");

reverse\_inorder(root, 1);

printf("\n");

break;

case 5:

reverse\_inorder(root, 2);

printf("The maximum element in the tree is %d.\n", max);

break;

case 6:

reverse\_inorder(root, 2);

printf("The minimum element in the tree is %d.\n", min);

break;

case 7:

printf("The maximum width of the tree is %d.\n", getMaxWidth(root));

break;

case 8:

exit(1);

default:

printf("Erroneous input.\n");

}

}

return 0;

}

**OUTPUT:-**









