**LAB CYCLE 3**

**SUBMITTED TO : SUBMITTED BY:**

**FOUSIA MISS NANDANA ANIL**

**ROLL NO : MCA107**

**S1 MCA**

**1.BINOMIAL HEAP**

#include<stdio.h>

#include<stdlib.h>

struct node {

int n;

int degree;

struct node\* parent;

struct node\* child;

struct node\* sibling;

};

struct node\* MAKE\_bin\_HEAP();

int bin\_LINK(struct node\*, struct node\*);

struct node\* CREATE\_NODE(int);

struct node\* bin\_HEAP\_UNION(struct node\*, struct node\*);

struct node\* bin\_HEAP\_INSERT(struct node\*, struct node\*);

struct node\* bin\_HEAP\_MERGE(struct node\*, struct node\*);

struct node\* bin\_HEAP\_EXTRACT\_MIN(struct node\*);

int REVERT\_LIST(struct node\*);

int DISPLAY(struct node\*);

struct node\* FIND\_NODE(struct node\*, int);

int bin\_HEAP\_DECREASE\_KEY(struct node\*, int, int);

int bin\_HEAP\_DELETE(struct node\*, int);

int count = 1;

struct node\* MAKE\_bin\_HEAP() {

struct node\* np;

np = NULL;

return np;

}

struct node \* H = NULL;

struct node \*Hr = NULL;

int bin\_LINK(struct node\* y, struct node\* z) {

y->parent = z;

y->sibling = z->child;

z->child = y;

z->degree = z->degree + 1;

}

struct node\* CREATE\_NODE(int k) {

struct node\* p;//new node;

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

p->n = k;

return p;

}

struct node\* bin\_HEAP\_UNION(struct node\* H1, struct node\* H2) {

struct node\* prev\_x;

struct node\* next\_x;

struct node\* x;

struct node\* H = MAKE\_bin\_HEAP();

H = bin\_HEAP\_MERGE(H1, H2);

if (H == NULL)

return H;

prev\_x = NULL;

x = H;

next\_x = x->sibling;

while (next\_x != NULL) {

if ((x->degree != next\_x->degree) || ((next\_x->sibling != NULL)

&& (next\_x->sibling)->degree == x->degree)) {

prev\_x = x;

x = next\_x;

} else {

if (x->n <= next\_x->n) {

x->sibling = next\_x->sibling;

bin\_LINK(next\_x, x);

} else {

if (prev\_x == NULL)

H = next\_x;

else

prev\_x->sibling = next\_x;

bin\_LINK(x, next\_x);

x = next\_x;

}

}

next\_x = x->sibling;

}

return H;

}

struct node\* bin\_HEAP\_INSERT(struct node\* H, struct node\* x) {

struct node\* H1 = MAKE\_bin\_HEAP();

x->parent = NULL;

x->child = NULL;

x->sibling = NULL;

x->degree = 0;

H1 = x;

H = bin\_HEAP\_UNION(H, H1);

return H;

}

struct node\* bin\_HEAP\_MERGE(struct node\* H1, struct node\* H2) {

struct node\* H = MAKE\_bin\_HEAP();

struct node\* y;

struct node\* z;

struct node\* a;

struct node\* b;

y = H1;

z = H2;

if (y != NULL) {

if (z != NULL && y->degree <= z->degree)

H = y;

DATA STRUCTURES LAB

40

else if (z != NULL && y->degree > z->degree)

H = z;

else

H = y;

} else

H = z;

while (y != NULL && z != NULL) {

if (y->degree < z->degree) {

y = y->sibling;

} else if (y->degree == z->degree) {

a = y->sibling;

y->sibling = z;

y = a;

} else {

b = z->sibling;

z->sibling = y;

z = b;

}

}

return H;

}

int DISPLAY(struct node\* H) {

struct node\* p;

if (H == NULL) {

printf("\nHEAP EMPTY");

return 0;

}

printf("\nTHE ROOT NODES ARE:-");

p = H;

while (p != NULL) {

printf("%d", p->n);

if (p->sibling != NULL)

printf("-->");

p = p->sibling;

}

printf("\n");

}

struct node\* bin\_HEAP\_EXTRACT\_MIN(struct node\* H1) {

int min;

struct node\* t = NULL;

struct node\* x = H1;

struct node \*Hr;

struct node\* p;

Hr = NULL;

if (x == NULL) {

printf("\nNOTHING TO EXTRACT");

return x;

}

// int min=x->n;

p = x;

while (p->sibling != NULL) {

if ((p->sibling)->n < min) {

min = (p->sibling)->n;

t = p;

x = p->sibling;

}

p = p->sibling;

}

if (t == NULL && x->sibling == NULL)

H1 = NULL;

else if (t == NULL)

H1 = x->sibling;

else if (t->sibling == NULL)

t = NULL;

else

t->sibling = x->sibling;

if (x->child != NULL) {

REVERT\_LIST(x->child);

(x->child)->sibling = NULL;

}

H = bin\_HEAP\_UNION(H1, Hr);

return x;

}

int REVERT\_LIST(struct node\* y) {

if (y->sibling != NULL) {

REVERT\_LIST(y->sibling);

(y->sibling)->sibling = y;

} else {

Hr = y;

}

}

struct node\* FIND\_NODE(struct node\* H, int k) {

struct node\* x = H;

struct node\* p = NULL;

if (x->n == k) {

p = x;

return p;

}

if (x->child != NULL && p == NULL) {

p = FIND\_NODE(x->child, k);

}

if (x->sibling != NULL && p == NULL) {

p = FIND\_NODE(x->sibling, k);

}

return p;

}

int bin\_HEAP\_DECREASE\_KEY(struct node\* H, int i, int k) {

int temp;

struct node\* p;

struct node\* y;

struct node\* z;

p = FIND\_NODE(H, i);

if (p == NULL) {

printf("\nINVALID CHOICE OF KEY TO BE REDUCED");

return 0;

}

if (k > p->n) {

printf("\nSORY!THE NEW KEY IS GREATER THAN CURRENT ONE");

return 0;

}

p->n = k;

y = p;

z = p->parent;

while (z != NULL && y->n < z->n) {

temp = y->n;

y->n = z->n;

z->n = temp;

y = z;

z = z->parent;

}

printf("KEY REDUCED SUCCESSFULLY!");

}

int bin\_HEAP\_DELETE(struct node\* H, int k) {

struct node\* np;

if (H == NULL) {

printf("\nHEAP EMPTY");

return 0;

}

bin\_HEAP\_DECREASE\_KEY(H, k, -1000);

np = bin\_HEAP\_EXTRACT\_MIN(H);

if (np != NULL)

printf("NODE DELETED SUCCESSFULLY");

}

int main() {

int i, n, m, l;

struct node\* p;

struct node\* np;

char ch;

printf("\nENTER THE NUMBER OF ELEMENTS:");

scanf("%d", &n);

printf("\nENTER THE ELEMENTS:\n");

for (i = 1; i <= n; i++) {

scanf("%d", &m);

np = CREATE\_NODE(m);

H = bin\_HEAP\_INSERT(H, np);

}

DISPLAY(H);

do {

printf("\nMENU:-\n");

printf("1)INSERT AN ELEMENT......2)EXTRACT THE MINIMUM KEY NODE...3)DECREASE A NODE KEY... 4)DELETE A NODE...5)QUIT\n");

scanf("%d", &l);

switch (l) {

case 1:

do {

printf("ENTER THE ELEMENT TO BE INSERTED:");

scanf("%d", &m);

p = CREATE\_NODE(m);

H = bin\_HEAP\_INSERT(H, p);

printf("NOW THE HEAP IS:");

DISPLAY(H);

printf("INSERT MORE(y/Y)= ");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'Y' || ch == 'y');

break;

case 2:

do {

printf("EXTRACTING THE MINIMUM KEY NODE");

p = bin\_HEAP\_EXTRACT\_MIN(H);

if (p != NULL)

printf("THE EXTRACTED NODE IS %d", p->n);

printf("NOW THE HEAP IS:");

DISPLAY(H);

printf("EXTRACT MORE(y/Y)");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'Y' || ch == 'y');

break;

case 3:

do {

printf("ENTER THE KEY OF THE NODE TO BE DECREASED:");

scanf("%d", &m);

printf("ENTER THE NEW KEY : ");

scanf("%d", &l);

bin\_HEAP\_DECREASE\_KEY(H, m, l);

printf("NOW THE HEAP IS:");

DISPLAY(H);

printf("DECREASE MORE(y/Y)");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'Y' || ch == 'y');

break;

case 4:

do {

printf("ENTER THE KEY TO BE DELETED: ");

scanf("%d", &m);

bin\_HEAP\_DELETE(H, m);

printf("DELETE MORE(y/Y)");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'y' || ch == 'Y');

break;

case 5:

break;

default:

printf("\nINVALID ENTRY...TRY AGAIN....\n");

}

} while (l != 5);

}

**2.B TREES**

#include<stdio.h>

#include<stdlib.h>

#define M 5

struct node{

int n; /\* n < M No. of keys in node will always less than order of B

tree \*/

int keys[M-1]; /\*array of keys\*/

struct node \*p[M]; /\* (n+1 pointers will be in use) \*/

}\*root=NULL;

enum KeyStatus { Duplicate,SearchFailure,Success,InsertIt,LessKeys };

void insert(int key);

void display(struct node \*root,int);

void DelNode(int x);

void search(int x);

enum KeyStatus ins(struct node \*r, int x, int\* y, struct node\*\* u);

int searchPos(int x,int \*key\_arr, int n);

enum KeyStatus del(struct node \*r, int x);

int main()

{

int key;

int choice;

printf("Creation of B tree for node %d\n",M);

while(1)

{

printf("1.Insert\n");

printf("2.Delete\n");

printf("3.Search\n");

printf("4.Display\n");

printf("5.Quit\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("Enter the key : ");

scanf("%d",&key);

insert(key);

break;

case 2:

printf("Enter the key : ");

scanf("%d",&key);

DelNode(key);

break;

case 3:

printf("Enter the key : ");

scanf("%d",&key);

search(key);

break;

case 4:

printf("Btree is :\n");

display(root,0);

break;

case 5:

exit(1);

default:

printf("Wrong choice\n");

break;

}/\*End of switch\*/

}/\*End of while\*/

return 0;

}/\*End of main()\*/

void insert(int key)

{

struct node \*newnode;

int upKey;

enum KeyStatus value;

value = ins(root, key, &upKey, &newnode);

if (value == Duplicate)

printf("Key already available\n");

if (value == InsertIt)

{

struct node \*uproot = root;

root=malloc(sizeof(struct node));

root->n = 1;

root->keys[0] = upKey;

root->p[0] = uproot;

root->p[1] = newnode;

}/\*End of if \*/

}/\*End of insert()\*/

enum KeyStatus ins(struct node \*ptr, int key, int \*upKey,struct node\*\*newnode)

{

struct node \*newPtr, \*lastPtr;

int pos, i, n,splitPos;

int newKey, lastKey;

enum KeyStatus value;

if (ptr == NULL)

{

\*newnode = NULL;

\*upKey = key;

return InsertIt;

}

n = ptr->n;

pos = searchPos(key, ptr->keys, n);

if (pos < n && key == ptr->keys[pos])

return Duplicate;

value = ins(ptr->p[pos], key, &newKey, &newPtr);

if (value != InsertIt)

return value;

/\*If keys in node is less than M-1 where M is order of B tree\*/

if (n < M - 1)

{

pos = searchPos(newKey, ptr->keys, n);

/\*Shifting the key and pointer right for inserting the new key\*/

for (i=n; i>pos; i--)

{

ptr->keys[i] = ptr->keys[i-1];

ptr->p[i+1] = ptr->p[i];

}

/\*Key is inserted at exact location\*/

ptr->keys[pos] = newKey;

ptr->p[pos+1] = newPtr;

++ptr->n; /\*incrementing the number of keys in node\*/

return Success;

}

if (pos == M - 1)

{

lastKey = newKey;

lastPtr = newPtr;

}

else /\*If keys in node are maximum and position of node to be inserted

is not last\*/

{

lastKey = ptr->keys[M-2];

lastPtr = ptr->p[M-1];

for (i=M-2; i>pos; i--)

{

ptr->keys[i] = ptr->keys[i-1];

ptr->p[i+1] = ptr->p[i];

}

ptr->keys[pos] = newKey;

ptr->p[pos+1] = newPtr;

}

splitPos = (M - 1)/2;

(\*upKey) = ptr->keys[splitPos];

(\*newnode)=malloc(sizeof(struct node));/\*Right node after split\*/

ptr->n = splitPos; /\*No. of keys for left splitted node\*/

(\*newnode)->n = M-1-splitPos;/\*No. of keys for right splitted node\*/

for (i=0; i < (\*newnode)->n; i++)

{

(\*newnode)->p[i] = ptr->p[i + splitPos + 1];

if(i < (\*newnode)->n - 1)

(\*newnode)->keys[i] = ptr->keys[i + splitPos + 1];

else

(\*newnode)->keys[i] = lastKey;

}

(\*newnode)->p[(\*newnode)->n] = lastPtr;

return InsertIt;

}/\*End of ins()\*/

void display(struct node \*ptr, int blanks)

{

if (ptr)

{

int i;

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

printf(" ");

for (i=0; i < ptr->n; i++)

printf("%d ",ptr->keys[i]);

printf("\n");

for (i=0; i <= ptr->n; i++)

display(ptr->p[i], blanks+10);

}/\*End of if\*/

}/\*End of display()\*/

void search(int key)

{

int pos, i, n;

struct node \*ptr = root;

printf("Search path:\n");

while (ptr)

{

n = ptr->n;

for (i=0; i < ptr->n; i++)

printf(" %d",ptr->keys[i]);

printf("\n");

pos = searchPos(key, ptr->keys, n);

if (pos < n && key == ptr->keys[pos])

{

printf("Key %d found in position %d of last dispalyed node\n",key,i);

return;

}

ptr = ptr->p[pos];

}

printf("Key %d is not available\n",key);

}/\*End of search()\*/

int searchPos(int key, int \*key\_arr, int n)

{

int pos=0;

while (pos < n && key > key\_arr[pos])

pos++;

return pos;

}/\*End of searchPos()\*/

void DelNode(int key)

{

struct node \*uproot;

enum KeyStatus value;

value = del(root,key);

switch (value)

{

case SearchFailure:

printf("Key %d is not available\n",key);

break;

case LessKeys:

uproot = root;

root = root->p[0];

free(uproot);

break;

}/\*End of switch\*/

}/\*End of delnode()\*/

enum KeyStatus del(struct node \*ptr, int key)

{

int pos, i, pivot, n ,min;

int \*key\_arr;

enum KeyStatus value;

struct node \*\*p,\*lptr,\*rptr;

if (ptr == NULL)

return SearchFailure;

/\*Assigns values of node\*/

n=ptr->n;

key\_arr = ptr->keys;

p = ptr->p;

min = (M - 1)/2;/\*Minimum number of keys\*/

pos = searchPos(key, key\_arr, n);

if (p[0] == NULL)

{

if (pos == n || key < key\_arr[pos])

return SearchFailure;

/\*Shift keys and pointers left\*/

for (i=pos+1; i < n; i++)

{

key\_arr[i-1] = key\_arr[i];

p[i] = p[i+1];

}

return --ptr->n >= (ptr==root ? 1 : min) ? Success : LessKeys;

}/\*End of if \*/

if (pos < n && key == key\_arr[pos])

{

struct node \*qp = p[pos], \*qp1;

int nkey;

while(1)

{

nkey = qp->n;

qp1 = qp->p[nkey];

if (qp1 == NULL)

break;

qp = qp1;

}/\*End of while\*/

key\_arr[pos] = qp->keys[nkey-1];

qp->keys[nkey - 1] = key;

}/\*End of if \*/

value = del(p[pos], key);

if (value != LessKeys)

return value;

if (pos > 0 && p[pos-1]->n > min)

{

pivot = pos - 1; /\*pivot for left and right node\*/

lptr = p[pivot];

rptr = p[pos];

rptr->p[rptr->n + 1] = rptr->p[rptr->n];

for (i=rptr->n; i>0; i--)

{

rptr->keys[i] = rptr->keys[i-1];

rptr->p[i] = rptr->p[i-1];

}

rptr->n++;

rptr->keys[0] = key\_arr[pivot];

rptr->p[0] = lptr->p[lptr->n];

key\_arr[pivot] = lptr->keys[--lptr->n];

return Success;

}

if (pos > min)

{

pivot = pos; /\*pivot for left and right node\*/

lptr = p[pivot];

rptr = p[pivot+1];

lptr->keys[lptr->n] = key\_arr[pivot];

lptr->p[lptr->n + 1] = rptr->p[0];

key\_arr[pivot] = rptr->keys[0];

lptr->n++;

rptr->n--;

for (i=0; i < rptr->n; i++)

{

rptr->keys[i] = rptr->keys[i+1];

rptr->p[i] = rptr->p[i+1];

}

rptr->p[rptr->n] = rptr->p[rptr->n + 1];

return Success;

}/\*End of if \*/

if(pos == n)

pivot = pos-1;

else

pivot = pos;

lptr = p[pivot];

rptr = p[pivot+1];

lptr->keys[lptr->n] = key\_arr[pivot];

lptr->p[lptr->n + 1] = rptr->p[0];

for (i=0; i < rptr->n; i++)

{

lptr->keys[lptr->n + 1 + i] = rptr->keys[i];

lptr->p[lptr->n + 2 + i] = rptr->p[i+1];

}

lptr->n = lptr->n + rptr->n +1;

free(rptr); /\*Remove right node\*/

for (i=pos+1; i < n; i++)

{

key\_arr[i-1] = key\_arr[i];

p[i] = p[i+1];

}

return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;

}

**3.RED BLACK TREE**

**INSERTION**

#include <stdio.h>

#include <stdlib.h>

enum COLOR {Red, Black};

typedef struct tree\_node {

int data;

struct tree\_node \*right;

struct tree\_node \*left;

struct tree\_node \*parent;

enum COLOR color;

}tree\_node;

typedef struct red\_black\_tree {

tree\_node \*root;

tree\_node \*NIL;

}red\_black\_tree;

tree\_node\* new\_tree\_node(int data) {

tree\_node\* n = malloc(sizeof(tree\_node));

n->left = NULL;

n->right = NULL;

n->parent = NULL;

n->data = data;

n->color = Red;

return n;

}

red\_black\_tree\* new\_red\_black\_tree() {

red\_black\_tree \*t = malloc(sizeof(red\_black\_tree));

tree\_node \*nil\_node = malloc(sizeof(tree\_node));

nil\_node->left = NULL;

nil\_node->right = NULL;

nil\_node->parent = NULL;

nil\_node->color = Black;

nil\_node->data = 0;

t->NIL = nil\_node;

t->root = t->NIL;

return t;

}

void left\_rotate(red\_black\_tree \*t, tree\_node \*x) {

tree\_node \*y = x->right;

x->right = y->left;

if(y->left != t->NIL) {

y->left->parent = x;

}

y->parent = x->parent;

if(x->parent == t->NIL) { //x is root

t->root = y;

}

else if(x == x->parent->left) { //x is left child

x->parent->left = y;

}

else { //x is right child

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

void right\_rotate(red\_black\_tree \*t, tree\_node \*x) {

tree\_node \*y = x->left;

x->left = y->right;

if(y->right != t->NIL) {

y->right->parent = x;

}

y->parent = x->parent;

if(x->parent == t->NIL) { //x is root

t->root = y;

}

else if(x == x->parent->right) { //x is left child

x->parent->right = y;

}

else { //x is right child

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

void insertion\_fixup(red\_black\_tree \*t, tree\_node \*z) {

while(z->parent->color == Red) {

if(z->parent == z->parent->parent->left) { //z.parent is the left child

tree\_node \*y = z->parent->parent->right; //uncle of z

if(y->color == Red) { //case 1

z->parent->color = Black;

y->color = Black;

z->parent->parent->color = Red;

z = z->parent->parent;

}

else { //case2 or case3

if(z == z->parent->right) { //case2

z = z->parent; //marked z.parent as new z

left\_rotate(t, z);

}

//case3

z->parent->color = Black; //made parent black

z->parent->parent->color = Red; //made parent red

right\_rotate(t, z->parent->parent);

}

}

else { //z.parent is the right child

tree\_node \*y = z->parent->parent->left; //uncle of z

if(y->color == Red) {

z->parent->color = Black;

y->color = Black;

z->parent->parent->color = Red;

z = z->parent->parent;

}

else {

if(z == z->parent->left) {

z = z->parent; //marked z.parent as new z

right\_rotate(t, z);

}

z->parent->color = Black; //made parent black

z->parent->parent->color = Red; //made parent red

left\_rotate(t, z->parent->parent);

}

}

}

t->root->color = Black;

}

void insert(red\_black\_tree \*t, tree\_node \*z) {

tree\_node\* y = t->NIL; //variable for the parent of the added node

tree\_node\* temp = t->root;

while(temp != t->NIL) {

y = temp;

if(z->data < temp->data)

temp = temp->left;

else

temp = temp->right;

}

z->parent = y;

if(y == t->NIL) { //newly added node is root

t->root = z;

}

else if(z->data < y->data) //data of child is less than its parent, left child

y->left = z;

else

y->right = z;

z->right = t->NIL;

z->left = t->NIL;

insertion\_fixup(t, z);

}

void inorder(red\_black\_tree \*t, tree\_node \*n) {

if(n != t->NIL) {

inorder(t, n->left);

printf("%d\n", n->data);

inorder(t, n->right);

}

}

int main() {

red\_black\_tree \*t = new\_red\_black\_tree();

tree\_node \*a, \*b, \*c, \*d, \*e, \*f, \*g, \*h, \*i, \*j, \*k, \*l, \*m;

a = new\_tree\_node(10);

b = new\_tree\_node(20);

c = new\_tree\_node(30);

d = new\_tree\_node(100);

e = new\_tree\_node(90);

f = new\_tree\_node(40);

g = new\_tree\_node(50);

h = new\_tree\_node(60);

i = new\_tree\_node(70);

j = new\_tree\_node(80);

k = new\_tree\_node(150);

l = new\_tree\_node(110);

m = new\_tree\_node(120);

insert(t, a);

insert(t, b);

insert(t, c);

insert(t, d);

insert(t, e);

insert(t, f);

insert(t, g);

insert(t, h);

insert(t, i);

insert(t, j);

insert(t, k);

insert(t, l);

insert(t, m);

inorder(t, t->root);

return 0;

}

**DELETION:**

#include <stdio.h>

#include <stdlib.h>

enum COLOR {Red, Black};

typedef struct tree\_node {

int data;

struct tree\_node \*right;

struct tree\_node \*left;

struct tree\_node \*parent;

enum COLOR color;

}tree\_node;

typedef struct red\_black\_tree {

tree\_node \*root;

tree\_node \*NIL;

}red\_black\_tree;

tree\_node\* new\_tree\_node(int data) {

tree\_node\* n = malloc(sizeof(tree\_node));

n->left = NULL;

n->right = NULL;

n->parent = NULL;

n->data = data;

n->color = Red;

return n;

}

red\_black\_tree\* new\_red\_black\_tree() {

red\_black\_tree \*t = malloc(sizeof(red\_black\_tree));

tree\_node \*nil\_node = malloc(sizeof(tree\_node));

nil\_node->left = NULL;

nil\_node->right = NULL;

nil\_node->parent = NULL;

nil\_node->color = Black;

nil\_node->data = 0;

t->NIL = nil\_node;

t->root = t->NIL;

return t;

}

void left\_rotate(red\_black\_tree \*t, tree\_node \*x) {

tree\_node \*y = x->right;

x->right = y->left;

if(y->left != t->NIL) {

y->left->parent = x;

}

y->parent = x->parent;

if(x->parent == t->NIL) { //x is root

t->root = y;

}

else if(x == x->parent->left) { //x is left child

x->parent->left = y;

}

else { //x is right child

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

void right\_rotate(red\_black\_tree \*t, tree\_node \*x) {

tree\_node \*y = x->left;

x->left = y->right;

if(y->right != t->NIL) {

y->right->parent = x;

}

y->parent = x->parent;

if(x->parent == t->NIL) { //x is root

t->root = y;

}

else if(x == x->parent->right) { //x is left child

x->parent->right = y;

}

else { //x is right child

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

void insertion\_fixup(red\_black\_tree \*t, tree\_node \*z) {

while(z->parent->color == Red) {

if(z->parent == z->parent->parent->left) { //z.parent is the left child

tree\_node \*y = z->parent->parent->right; //uncle of z

if(y->color == Red) { //case 1

z->parent->color = Black;

y->color = Black;

z->parent->parent->color = Red;

z = z->parent->parent;

}

else { //case2 or case3

if(z == z->parent->right) { //case2

z = z->parent; //marked z.parent as new z

left\_rotate(t, z);

}

//case3

z->parent->color = Black; //made parent black

z->parent->parent->color = Red; //made parent red

right\_rotate(t, z->parent->parent);

}}

else { //z.parent is the right child

tree\_node \*y = z->parent->parent->left; //uncle of z

if(y->color == Red) {

z->parent->color = Black;

y->color = Black;

z->parent->parent->color = Red;

z = z->parent->parent;

}

else {

if(z == z->parent->left) {

z = z->parent; //marked z.parent as new z

right\_rotate(t, z);

}

z->parent->color = Black; //made parent black

z->parent->parent->color = Red; //made parent red

left\_rotate(t, z->parent->parent);

}}}

t->root->color = Black;

}

void insert(red\_black\_tree \*t, tree\_node \*z) {

tree\_node\* y = t->NIL; //variable for the parent of the added node

tree\_node\* temp = t->root;

while(temp != t->NIL) {

y = temp;

if(z->data < temp->data)

temp = temp->left;

else

temp = temp->right;

}

z->parent = y;

if(y == t->NIL) { //newly added node is root

t->root = z;

}

else if(z->data < y->data) //data of child is less than its parent, left child

y->left = z;

else

y->right = z;

z->right = t->NIL;

z->left = t->NIL;

insertion\_fixup(t, z);

}

void rb\_transplant(red\_black\_tree \*t, tree\_node \*u, tree\_node \*v) {

if(u->parent == t->NIL)

t->root = v;

else if(u == u->parent->left)

u->parent->left = v;

else

u->parent->right = v;

v->parent = u->parent;

}

tree\_node\* minimum(red\_black\_tree \*t, tree\_node \*x) {

while(x->left != t->NIL)

x = x->left;

return x;

}

void rb\_delete\_fixup(red\_black\_tree \*t, tree\_node \*x) {

while(x != t->root && x->color == Black) {

if(x == x->parent->left) {

tree\_node \*w = x->parent->right;

if(w->color == Red) {

w->color = Black;

x->parent->color = Red;

left\_rotate(t, x->parent);

w = x->parent->right;

}

if(w->left->color == Black && w->right->color == Black) {

w->color = Red;

x = x->parent;

}

else {

if(w->right->color == Black) {

w->left->color = Black;

w->color = Red;

right\_rotate(t, w);

w = x->parent->right;

}

w->color = x->parent->color;

x->parent->color = Black;

w->right->color = Black;

left\_rotate(t, x->parent);

x = t->root;

}}

else {

tree\_node \*w = x->parent->left;

if(w->color == Red) {

w->color = Black;

x->parent->color = Red;

right\_rotate(t, x->parent);

w = x->parent->left;

}

if(w->right->color == Black && w->left->color == Black) {

w->color = Red;

x = x->parent;

}

else {

if(w->left->color == Black) {

w->right->color = Black;

w->color = Red;

left\_rotate(t, w);

w = x->parent->left;

}

w->color = x->parent->color;

x->parent->color = Black;

w->left->color = Black;

right\_rotate(t, x->parent);

x = t->root;

}}}

x->color = Black;

}

void rb\_delete(red\_black\_tree \*t, tree\_node \*z) {

tree\_node \*y = z;

tree\_node \*x;

enum COLOR y\_orignal\_color = y->color;

if(z->left == t->NIL) {

x = z->right;

rb\_transplant(t, z, z->right);

}

else if(z->right == t->NIL) {

x = z->left;

rb\_transplant(t, z, z->left);

}

else {

y = minimum(t, z->right);

y\_orignal\_color = y->color;

x = y->right;

if(y->parent == z) {

x->parent = z;

}

else {

rb\_transplant(t, y, y->right);

y->right = z->right;

y->right->parent = y;

}

rb\_transplant(t, z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

if(y\_orignal\_color == Black)

rb\_delete\_fixup(t, x);

}

void inorder(red\_black\_tree \*t, tree\_node \*n) {

if(n != t->NIL) {

inorder(t, n->left);

printf("%d\n", n->data);

inorder(t, n->right);

}}

int main() {

red\_black\_tree \*t = new\_red\_black\_tree();

tree\_node \*a, \*b, \*c, \*d, \*e, \*f, \*g, \*h, \*i, \*j, \*k, \*l, \*m;

a = new\_tree\_node(10);

b = new\_tree\_node(20);

c = new\_tree\_node(30);

d = new\_tree\_node(100);

e = new\_tree\_node(90);

f = new\_tree\_node(40);

g = new\_tree\_node(50);

h = new\_tree\_node(60);

i = new\_tree\_node(70);

j = new\_tree\_node(80);

k = new\_tree\_node(150);

l = new\_tree\_node(110);

m = new\_tree\_node(120);

insert(t, a);

insert(t, b);

insert(t, c);

insert(t, d);

insert(t, e);

insert(t, f);

insert(t, g);

insert(t, h);

insert(t, i);

insert(t, j);

insert(t, k);

insert(t, l);

insert(t, m);

rb\_delete(t, a);

rb\_delete(t, m);

inorder(t, t->root);

return 0;}

**LINK TO GITHUB REPOSITORY:**

<https://github.com/NandanaAnil/Data-Structures.git>