LAB CYCLE 3

SUBMITTED TO :

FOUSIA MISS

SUBMITTED BY:

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;
```

```
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