

# Semester: FALL18

Course Code: CSE225.L

**Section: 05**

# PROJECT REPORT

# Red Black Tree

Submitted on: 27th Dec. 2018

# Prepared By –

## E.M.K. Fahmidur Rahman -1620535042

## Afshaan Md Shaan -1620910042

## K.M.H. Mubin -1621802042

**Data Structure: Red Black Tree**

**Introduction**

Red-black trees are an evolution of binary search trees that aim to keep the tree balanced without affecting the complexity of the primitive operations. This is done by coloring each node in the tree with either red or black and preserving a set of properties that guarantee that the deepest path in the tree is not longer than twice the shortest one.

A red-black tree is a binary search tree with the following properties:

1. Every node is colored with either red or black.

2. All leaf (nil) nodes are colored with black; if a node’s child is missing then we will assume that it has a nil child in that place and this nil child is always colored black.

3. Both children of a red node must be black nodes.

4. Every path from a node n to a descendent leaf has the same number of black nodes (not counting node n). We call this number the black height of n, which is denoted by bh(n).

Red black tree have most of the Binary search tree operation. Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Log n) after every insertion and deletion, then we can guarantee an upper bound of O(Log n) for all these operations. The height of a Red-Black tree is always O(Log n) where n is the number of nodes in the tree.

**Rotation**

red-black trees employ a key operation known as rotation. Rotation is a binary operation, between a parent node and one of its children, that swaps nodes and modify their pointers while preserving the inorder traversal of the tree (so that elements are still sorted).

There are four types of rotations: left-left rotation, right-right rotation, left-right rotation, right-left rotation . Left rotation swaps the parent node with its right child, while right rotation swaps the parent node with its left child.

Red-black tree operations are a modified version of BST operations, with the modifications aiming to preserve the properties of red-black trees while keeping the operations complexity a function of tree height.

**Insertion**

Inserting a node in a red-black tree step process are :

1. A BST insertion, which takes O(log n) as shown before.

2. Fixing any violations to red-black tree properties that may occur after applying step 1. This step is O(log n) also, as we start by fixing the newly inserted node, continuing up along the path to the root node and fixing nodes along that path. Fixing a node is done in constant time and involves re-coloring some nodes and doing rotations.

**Deletion**

The same concept behind red-black tree insertions applies here. Removing a node from a red-black tree makes use of the BST deletion procedure and then restores the red-black tree properties in O(log n). The total running time for the deletion process takes O(log n) time, then, which meets the complexity requirements for the primitive operations.

**Conclusion**

Red black tree has so many real world application. It is use in sets and maps and also use for standard template library. Red black tree are also use for search in dictionaries and searching on the web.

**Division of Responsibilities:**

E.M.K. Fahmidur Rahman – Research and code Writing.

Afshaan Md Shaan – Research, report writing and code writing.

K.M.H. Mubin – Research, Presentation writing and code writing.

**Source Code**

**Red Black Tree**

**RBTREE.H**

#ifndef RBTREE\_H

#define RBTREE\_H

#include <iostream>

using namespace std;

struct node

{

int key;

node \*parent;

char color;

node \*left;

node \*right;

};

class RBtree

{

node \*root;

node \*q;

public :

RBtree();

void insert();

void insertfix(node \*);

void leftrotate(node \*);

void rightrotate(node \*);

void del();

node\* successor(node \*);

void delfix(node \*);

void disp();

void display( node \*);

void search();

};

#endif // RBTREE\_H

**RBTREE.CPP**

#include "rbtree.h"

RBtree::RBtree()

{

q = 0;

root = 0;

}

void RBtree::insert()

{

int z;

cout<<"\nEnter a value: ";

cin>>z;

node \*p,\*q;

node \*t=new node;

t->key=z;

t->left=NULL;

t->right=NULL;

t->color='r';

p=root;

q=NULL;

if(root==NULL)

{

root=t;

t->parent=NULL;

}

else

{

while(p!=NULL)

{

q=p;

if(p->key<t->key)

p=p->right;

else

p=p->left;

}

t->parent=q;

if(q->key<t->key)

q->right=t;

else

q->left=t;

}

insertfix(t);

}

void RBtree::insertfix(node \*t)

{

node \*u;

if(root==t)

{

t->color='b';

return;

}

while(t->parent!=NULL&&t->parent->color=='r')

{

node \*g=t->parent->parent;

if(g->left==t->parent)

{

if(g->right!=NULL)

{

u=g->right;

if(u->color=='r')

{

t->parent->color='b';

u->color='b';

g->color='r';

t=g;

}

}

else

{

if(t->parent->right==t)

{

t=t->parent;

leftrotate(t);

}

t->parent->color='b';

g->color='r';

rightrotate(g);

}

}

else

{

if(g->left!=NULL)

{

u=g->left;

if(u->color=='r')

{

t->parent->color='b';

u->color='b';

g->color='r';

t=g;

}

}

else

{

if(t->parent->left==t)

{

t=t->parent;

rightrotate(t);

}

t->parent->color='b';

g->color='r';

leftrotate(g);

}

}

root->color='b';

}

}

void RBtree::del()

{

if(root==NULL)

{

cout<<"\nEmpty Tree." ;

return ;

}

int x;

cout<<"\nEnter the value to be deleted: ";

cin>>x;

node \*p;

p=root;

node \*y=NULL;

node \*q=NULL;

int found=0;

while(p!=NULL&&found==0)

{

if(p->key==x)

found=1;

if(found==0)

{

if(p->key<x)

p=p->right;

else

p=p->left;

}

}

if(found==0)

{

cout<<"\nElement Not Found.";

return ;

}

else

{

cout<<"\nDeleted Element: "<<p->key;

cout<<"\nColor: ";

if(p->color=='b')

cout<<"Black\n";

else

cout<<"Red\n";

if(p->parent!=NULL)

cout<<"\nParent: "<<p->parent->key;

else

cout<<"\nThere is no parent of the node. ";

if(p->right!=NULL)

cout<<"\nRight Child: "<<p->right->key;

else

cout<<"\nThere is no right child of the node. ";

if(p->left!=NULL)

cout<<"\nLeft Child: "<<p->left->key;

else

cout<<"\nThere is no left child of the node. ";

cout<<"\nNode Deleted.";

if(p->left==NULL||p->right==NULL)

y=p;

else

y=successor(p);

if(y->left!=NULL)

q=y->left;

else

{

if(y->right!=NULL)

q=y->right;

else

q=NULL;

}

if(q!=NULL)

q->parent=y->parent;

if(y->parent==NULL)

root=q;

else

{

if(y==y->parent->left)

y->parent->left=q;

else

y->parent->right=q;

}

if(y!=p)

{

p->color=y->color;

p->key=y->key;

}

if(y->color=='b')

delfix(q);

}

}

void RBtree::delfix(node \*p)

{

node \*s;

while(p!=root&&p->color=='b')

{

if(p->parent->left==p)

{

s=p->parent->right;

if(s->color=='r')

{

s->color='b';

p->parent->color='r';

leftrotate(p->parent);

s=p->parent->right;

}

if(s->right->color=='b'&&s->left->color=='b')

{

s->color='r';

p=p->parent;

}

else

{

if(s->right->color=='b')

{

s->left->color='b';

s->color='r';

rightrotate(s);

s=p->parent->right;

}

s->color=p->parent->color;

p->parent->color='b';

s->right->color='b';

leftrotate(p->parent);

p=root;

}

}

else

{

s=p->parent->left;

if(s->color=='r')

{

s->color='b';

p->parent->color='r';

rightrotate(p->parent);

s=p->parent->left;

}

if(s->left->color=='b'&&s->right->color=='b')

{

s->color='r';

p=p->parent;

}

else

{

if(s->left->color=='b')

{

s->right->color='b';

s->color='r';

leftrotate(s);

s=p->parent->left;

}

s->color=p->parent->color;

p->parent->color='b';

s->left->color='b';

rightrotate(p->parent);

p=root;

}

}

p->color='b';

root->color='b';

}

}

void RBtree::leftrotate(node \*p)

{

if(p->right==NULL)

return ;

else

{

node \*y=p->right;

if(y->left!=NULL)

{

p->right=y->left;

y->left->parent=p;

}

else

p->right=NULL;

if(p->parent!=NULL)

y->parent=p->parent;

if(p->parent==NULL)

root=y;

else

{

if(p==p->parent->left)

p->parent->left=y;

else

p->parent->right=y;

}

y->left=p;

p->parent=y;

}

}

void RBtree::rightrotate(node \*p)

{

if(p->left==NULL)

return ;

else

{

node \*y=p->left;

if(y->right!=NULL)

{

p->left=y->right;

y->right->parent=p;

}

else

p->left=NULL;

if(p->parent!=NULL)

y->parent=p->parent;

if(p->parent==NULL)

root=y;

else

{

if(p==p->parent->left)

p->parent->left=y;

else

p->parent->right=y;

}

y->right=p;

p->parent=y;

}

}

node\* RBtree::successor(node \*p)

{

node \*y=NULL;

if(p->left!=NULL)

{

y=p->left;

while(y->right!=NULL)

y=y->right;

}

else

{

y=p->right;

while(y->left!=NULL)

y=y->left;

}

return y;

}

void RBtree::disp()

{

display(root);

}

void RBtree::display(node \*p)

{

if(root==NULL)

{

cout<<"\nEmpty Tree.";

return ;

}

if(p!=NULL)

{

cout<<"\n\t NODE: ";

cout<<"\n Key: "<<p->key;

cout<<"\n Color: ";

if(p->color=='b')

cout<<"Black";

else

cout<<"Red";

if(p->parent!=NULL)

cout<<"\n Parent: "<<p->parent->key;

else

cout<<"\n There is no parent of the node. ";

if(p->right!=NULL)

cout<<"\n Right Child: "<<p->right->key;

else

cout<<"\n There is no right child of the node. ";

if(p->left!=NULL)

cout<<"\n Left Child: "<<p->left->key;

else

cout<<"\n There is no left child of the node. ";

cout<<endl;

if(p->left)

{

cout<<"\n\nLeft:\n";

display(p->left);

}

if(p->right)

{

cout<<"\n\nRight:\n";

display(p->right);

}

}

}

void RBtree::search()

{

if(root==NULL)

{

cout<<"\nEmpty Tree\n" ;

return ;

}

int x;

cout<<"\n Enter a value to be searched: ";

cin>>x;

node \*p=root;

int found=0;

while(p!=NULL&& found==0)

{

if(p->key==x)

found=1;

if(found==0)

{

if(p->key<x)

p=p->right;

else

p=p->left;

}

}

if(found==0)

cout<<"\nElement Not Found.";

else

{

cout<<"\n\t FOUND NODE: ";

cout<<"\n Key: "<<p->key;

cout<<"\n Color: ";

if(p->color=='b')

cout<<"Black";

else

cout<<"Red";

if(p->parent!=NULL)

cout<<"\n Parent: "<<p->parent->key;

else

cout<<"\n There is no parent of the node. ";

if(p->right!=NULL)

cout<<"\n Right Child: "<<p->right->key;

else

cout<<"\n There is no right child of the node. ";

if(p->left!=NULL)

cout<<"\n Left Child: "<<p->left->key;

else

cout<<"\n There is no left child of the node. ";

cout<<endl;

}

}

**MANI.CPP**

#include <iostream>

#include "rbtree.h"

using namespace std;

int main()

{

int ch,y=0;

RBtree obj;

do

{

cout<<"\n\t RED BLACK TREE "<<endl;

cout<<"\t-----------------"<<endl;

cout<<"\n 1. Insert a node in the tree ";

cout<<"\n 2. Delete a node from the tree";

cout<<"\n 3. Search for an element in the tree";

cout<<"\n 4. Display the tree ";

cout<<"\n 5. Exit " ;

cout<<"\nEnter Your Choice: ";

cin>>ch;

switch(ch)

{

case 1 :

obj.insert();

cout<<"\nNode Inserted.\n";

break;

case 2 :

obj.del();

break;

case 3 :

obj.search();

break;

case 4 :

obj.disp();

break;

case 5 :

y=1;

break;

default :

cout<<"\nEnter a Valid Choice.";

}

cout<<endl;

}

while(y!=1);

return 1;

}