MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY

«KHARKIV POLYTECHNIC INSTITUTE»

Department of Software Engineering and Management Information Technologies

Report from lab № 3

discipline «Algorithm and Data Structures»

Kharkiv

2018

Theme :Basic Data Structures:

Objective: explore red-black trees and get programming skills of algorithms

that process them.

Task:

Develop a program that reads numbers N, M (1 <N, M <256), a sequence of N

keys (integers, real numbers or strings (up to 255 characters) depending on the

variant), and a sequence of M keys. The program saves the first sequence to redblack

trees.

Whenever a new element is added to tree, statistics must be display according

to variant.

1 The minimum element and its color;

2 The maximum element and its color.

After building a tree, results of the following operations must be shown for the

tree and for every element x of the second sequence (according to variant).

1 Does item x exist in the tree and what is its color.

2 Successor(x) and its color.

3 Predecessor(x) and its color.

Usage of ready data structures (e.g., STL) is prohibited, but string

implementations can be used (for example, std::string in C++).

**Progress of the lab:**

Theory : The search tree data structure supports many dynamic-set operations, including

SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT,

and DELETE. Thus, we can use a search tree both as a dictionary and as a priority

queue.

Basic operations on a binary search tree take time proportional to the height of

the tree. For a complete binary tree with n nodes, such operations run in Θ(*lg n*)

worst-case time. If the tree is a linear chain of n nodes, however, the same operations

take Θ(*n*) worst-case time.

Thus, the set operations are fast if the height of the search tree is small. If its

height is large, however, the set operations may run no faster than with a linked list.

Red-black trees are one of many search-tree schemes that are “balanced” in order to

guarantee that basic dynamic-set operations take O(lg n) time in the worst case.

A red-black tree is a binary search tree with one extra bit of storage per node:

its color, which can be either RED or BLACK. By constraining the node colors on

any simple path from the root to a leaf, red-black trees ensure that no such path is

more than twice as long as any other, so that the tree is approximately balanced.

Each node of the tree now contains the attributes color, key, left, right, and p.

If a child or the parent of a node does not exist, the corresponding pointer attribute of

the node contains the value NIL. We shall regard these NILs as being pointers to

leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as

being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following red-black

properties.

12

1. Every node is either red or black.

2. The root is black.

3. Every leaf (NIL) is black.

4. If a node is red, then both its children are black.

5. For each node, all simple paths from the node to descendant leaves contain

the same number of black nodes

Tree-Insert and Tree-Delete operations take *O*(*lg n*) on a red-black tree, they

do not guarantee that the modified binary search tree will be a red-black tree. A tree

can require modification and rotations. This process is described in lectures and

recommended sources.

CODE:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace R\_B\_trees

{

// C# program to find minimum value node in Binary Search Tree

// A binary tree node

class Program

{

static void Main(string[] args)

{

RB tree = new RB();

while (true)

{

try

{

Console.WriteLine("\nMenu");

Console.WriteLine("1. Add a record to the Tree");

Console.WriteLine("2. check a record for existance ");

Console.WriteLine("3. Exit ");

string ch = Console.ReadLine();

switch (ch)

{

case "1":

{

//Console.WriteLine("\n Enter element to Insert");

//int record = Convert.ToInt32(Console.ReadLine());

//tree.Insert(record);

tree.Insert(5);

tree.DisplayTree();

tree.getMax();

Console.WriteLine();

tree.Insert(3);

tree.DisplayTree();

tree.getMax();

Console.WriteLine();

tree.Insert(7);

tree.DisplayTree();

tree.getMax();

Console.WriteLine();

tree.Insert(1);

tree.DisplayTree();

tree.getMax();

Console.WriteLine();

tree.Insert(9);

tree.DisplayTree();

tree.getMax();

Console.WriteLine();

tree.Insert(-1);

tree.DisplayTree();

tree.getMax();

Console.WriteLine();

tree.Insert(11);

tree.DisplayTree();

tree.getMax();

Console.WriteLine();

tree.Insert(6);

tree.DisplayTree();

tree.getMax();

Console.WriteLine();

tree.DisplayTree();

tree.getMax();

Console.WriteLine();

}

break;

case "2":

{

Console.WriteLine("\n Enter element to find");

int record = Convert.ToInt32(Console.ReadLine());

tree.Find(record);

}

break;

case "3":

return;

default:

{

Console.WriteLine("\nInvalid option");

}

break;

}

}

catch (Exception e)

{

Console.WriteLine("Check for the values entered.");

}

}

}

}

enum Color

{

Red,

Black

}

class RB

{

/// <summary>

/// Object of type Node contains 4 properties

/// Colour

/// Left

/// Right

/// Parent

/// Data

/// </summary>

public class Node

{

public Color colour;

public Node left;

public Node right;

public Node parent;

public int data;

public Node(int data) { this.data = data; }

public Node(Color colour) { this.colour = colour; }

public Node(int data, Color colour) { this.data = data; this.colour = colour; }

}

/// <summary>

/// Root node of the tree (both reference & pointer)

/// </summary>

private Node root;

/// <summary>

/// New instance of a Red-Black tree object

/// </summary>

public RB() { }

/// <summary>

/// Left Rotate

/// </summary>

/// <param name="X"></param>

/// <returns>void</returns>

private void LeftRotate(Node X)

{

Node Y = X.right; // set Y

X.right = Y.left;//turn Y's left subtree into X's right subtree

if (Y.left != null)

{

Y.left.parent = X;

}

if (Y != null)

{

Y.parent = X.parent;//link X's parent to Y

}

if (X.parent == null)

{

root = Y;

}

if (X == X.parent.left)

{

X.parent.left = Y;

}

else

{

X.parent.right = Y;

}

Y.left = X; //put X on Y's left

if (X != null)

{

X.parent = Y;

}

}

/// <summary>

/// Rotate Right

/// </summary>

/// <param name="Y"></param>

/// <returns>void</returns>

private void RightRotate(Node Y)

{

// right rotate is simply mirror code from left rotate

Node X = Y.left;

Y.left = X.right;

if (X.right != null)

{

X.right.parent = Y;

}

if (X != null)

{

X.parent = Y.parent;

}

if (Y.parent == null)

{

root = X;

}

if (Y == Y.parent.right)

{

Y.parent.right = X;

}

if (Y == Y.parent.left)

{

Y.parent.left = X;

}

X.right = Y;//put Y on X's right

if (Y != null)

{

Y.parent = X;

}

}

/// <summary>

/// Display Tree

/// </summary>

public void DisplayTree()

{

if (root == null)

{

Console.WriteLine("Nothing in the tree!");

return;

}

if (root != null)

{

InOrderDisplay(root);

}

}

/// <summary>

/// Find item in the tree

/// </summary>

/// <param name="key"></param>

public Node Find(int key)

{

bool isFound = false;

Node temp = root;

Node item = null;

while (!isFound)

{

if (temp == null)

{

break;

}

if (key < temp.data)

{

temp = temp.left;

}

if (key > temp.data)

{

temp = temp.right;

}

if (temp != null)

if (key == temp.data)

{

isFound = true;

item = temp;

}

}

if (isFound)

{

Console.WriteLine("{0} was found and {1} is the color", key ,temp.colour);

return temp;

}

else

{

Console.WriteLine("{0} not found", key);

return null;

}

}

/// <summary>

/// Insert a new object into the RB Tree

/// </summary>

/// <param name="item"></param>

public void Insert(int item)

{

Node newItem = new Node(item);

if (root == null)

{

root = newItem;

root.colour = Color.Black;

return;

}

Node Y = null;

Node X = root;

while (X != null)

{

Y = X;

if (newItem.data < X.data)

{

X = X.left;

}

else

{

X = X.right;

}

}

newItem.parent = Y;

if (Y == null)

{

root = newItem;

}

else if (newItem.data < Y.data)

{

Y.left = newItem;

}

else

{

Y.right = newItem;

}

newItem.left = null;

newItem.right = null;

newItem.colour = Color.Red;//colour the new node red

InsertFixUp(newItem);//call method to check for violations and fix

}

private void InOrderDisplay(Node current)

{

if (current != null)

{

InOrderDisplay(current.left);

Console.Write("({0}) ", current.data);

InOrderDisplay(current.right);

}

}

public void getMax()

{

Node maxi = root;

max(root, ref maxi);

Console.WriteLine("max is " + maxi.data);

}

private void max(Node current, ref Node maxi)

{

if (current != null)

{

max(current.left, ref maxi);

if ( current.data > maxi.data)

maxi = current;

max(current.right, ref maxi);

}

}

private void InsertFixUp(Node item)

{

//Checks Red-Black Tree properties

while (item != root && item.parent.colour == Color.Red)

{

/\*We have a violation\*/

if (item.parent == item.parent.parent.left)

{

Node Y = item.parent.parent.right;

if (Y != null && Y.colour == Color.Red)//Case 1: uncle is red

{

item.parent.colour = Color.Black;

Y.colour = Color.Black;

item.parent.parent.colour = Color.Red;

item = item.parent.parent;

}

else //Case 2: uncle is black

{

if (item == item.parent.right)

{

item = item.parent;

LeftRotate(item);

}

//Case 3: recolour & rotate

item.parent.colour = Color.Black;

item.parent.parent.colour = Color.Red;

RightRotate(item.parent.parent);

}

}

else

{

//mirror image of code above

Node X = null;

X = item.parent.parent.left;

if (X != null && X.colour == Color.Black)//Case 1

{

item.parent.colour = Color.Red;

X.colour = Color.Red;

item.parent.parent.colour = Color.Black;

item = item.parent.parent;

}

else //Case 2

{

if (item == item.parent.left)

{

item = item.parent;

RightRotate(item);

}

//Case 3: recolour & rotate

item.parent.colour = Color.Black;

item.parent.parent.colour = Color.Red;

LeftRotate(item.parent.parent);

}

}

root.colour = Color.Black;//re-colour the root black as necessary

}

}

/// <summary>

/// Deletes a specified value from the tree

/// </summary>

/// <param name="item"></param>

public void Delete(int key)

{

//first find the node in the tree to delete and assign to item pointer/reference

Node item = Find(key);

Node X = null;

Node Y = null;

if (item == null)

{

Console.WriteLine("Nothing to delete!");

return;

}

if (item.left == null || item.right == null)

{

Y = item;

}

else

{

Y = TreeSuccessor(item);

}

if (Y.left != null)

{

X = Y.left;

}

else

{

X = Y.right;

}

if (X != null)

{

X.parent = Y;

}

if (Y.parent == null)

{

root = X;

}

else if (Y == Y.parent.left)

{

Y.parent.left = X;

}

else

{

Y.parent.left = X;

}

if (Y != item)

{

item.data = Y.data;

}

if (Y.colour == Color.Black)

{

DeleteFixUp(X);

}

}

/// <summary>

/// Checks the tree for any violations after deletion and performs a fix

/// </summary>

/// <param name="X"></param>

private void DeleteFixUp(Node X)

{

while (X != null && X != root && X.colour == Color.Black)

{

if (X == X.parent.left)

{

Node W = X.parent.right;

if (W.colour == Color.Red)

{

W.colour = Color.Black; //case 1

X.parent.colour = Color.Red; //case 1

LeftRotate(X.parent); //case 1

W = X.parent.right; //case 1

}

if (W.left.colour == Color.Black && W.right.colour == Color.Black)

{

W.colour = Color.Red; //case 2

X = X.parent; //case 2

}

else if (W.right.colour == Color.Black)

{

W.left.colour = Color.Black; //case 3

W.colour = Color.Red; //case 3

RightRotate(W); //case 3

W = X.parent.right; //case 3

}

W.colour = X.parent.colour; //case 4

X.parent.colour = Color.Black; //case 4

W.right.colour = Color.Black; //case 4

LeftRotate(X.parent); //case 4

X = root; //case 4

}

else //mirror code from above with "right" & "left" exchanged

{

Node W = X.parent.left;

if (W.colour == Color.Red)

{

W.colour = Color.Black;

X.parent.colour = Color.Red;

RightRotate(X.parent);

W = X.parent.left;

}

if (W.right.colour == Color.Black && W.left.colour == Color.Black)

{

W.colour = Color.Black;

X = X.parent;

}

else if (W.left.colour == Color.Black)

{

W.right.colour = Color.Black;

W.colour = Color.Red;

LeftRotate(W);

W = X.parent.left;

}

W.colour = X.parent.colour;

X.parent.colour = Color.Black;

W.left.colour = Color.Black;

RightRotate(X.parent);

X = root;

}

}

if (X != null)

X.colour = Color.Black;

}

private Node Minimum(Node X)

{

while (X.left.left != null)

{

X = X.left;

}

if (X.left.right != null)

{

X = X.left.right;

}

return X;

}

private Node TreeSuccessor(Node X)

{

if (X.left != null)

{

return Minimum(X);

}

else

{

Node Y = X.parent;

while (Y != null && X == Y.right)

{

X = Y;

Y = Y.parent;

}

return Y;

}

}

}

}

:

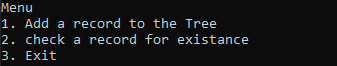


Figure.1- Menu of program

\

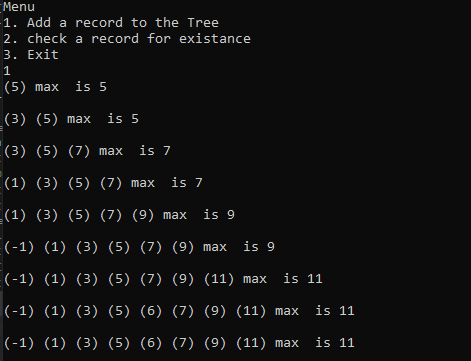


Figure.2- Add records to tree

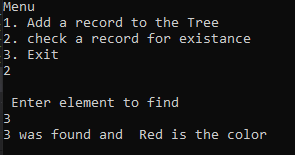


Figure.3- check for existence

Conclusion:

In this laboratory the study of Red black trees was considered

* Build a red black tree.
* Perform data structure operation.
* Search red black tree.