МИНОБРНАУКИ РФ

ФГБОУ ВО Тверской государственный технический университет

Кафедра “Программное обеспечение”

Тема курсовой работы:

«Красно-черные деревья»

Выполнил:

Соколов Никита Дмитриевич, 2 курс, факультет

информационных технологий, группа: Б ПИН РИС-20.05

Проверил: Мальков Александр Анатольевич

Оглавление

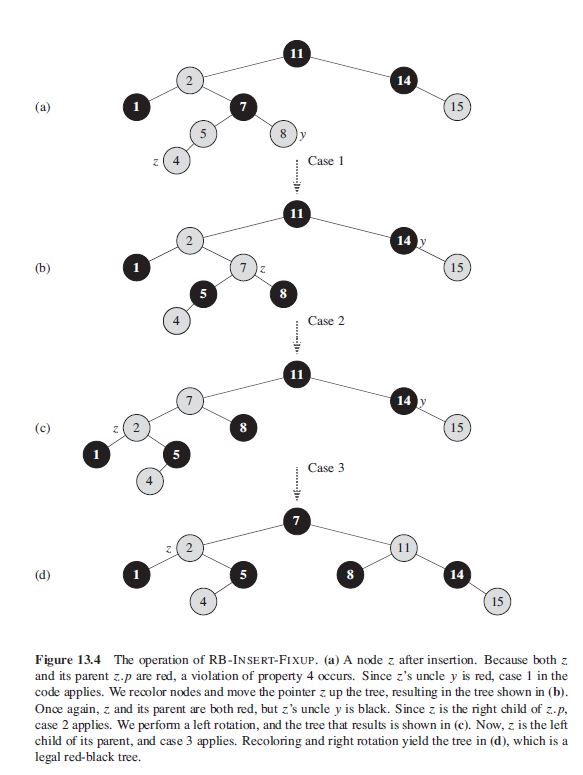
1. [Красно-черное дерево в C# 3](#_Toc103087300)
2. [Используемые методы: 5](#_Toc103087301)
3. [Объекты 6](#_Toc103087302)
4. [Код программы: 6](#_Toc103087303)

# Красно-черное дерево в C#

Красно-черные деревья — еще одна самобалансирующаяся структура данных двоичного дерева поиска. Очень похоже на дерево AVL , которое также является самобалансирующимся и имеет одинаковую временную сложность O (log n) для лучшего, среднего и худшего случая. Красно-черные деревья немного более эффективны при вставке и удалении, поскольку для них требуется меньше работы, поскольку нам не нужно просматривать дерево на предмет дисбаланса, мы просто смотрим на цвет родительского узла и выполняем исправление.

Красно-черное дерево определяется и должно соответствовать следующим требованиям:

* Узел либо красный, либо черный. Это означает, что у каждого узла есть свойство, определяющее цвет, например, с помощью перечисления.
* Корень всегда черный. (Это правило иногда опускается. Поскольку корень всегда можно изменить с красного на черный, но не обязательно наоборот, это правило мало влияет на анализ.)
* Все листья (NIL) черные. (Все листья того же цвета, что и корень.)
* Каждый красный узел должен иметь два черных дочерних узла.
* Каждый путь от данного узла к любому из его дочерних листьев содержит одинаковое количество черных узлов. (Любой листовой узел должен иметь одинаковое количество черных родительских узлов)
* В дереве RB баланс поддерживается за счет «окрашивания» узлов в красный или черный цвет и выполнения необходимых перестановок узлов. В отличие от дерева AVL, где баланс поддерживается проверкой коэффициента баланса. Однако оба требуют вращения узлов.



# Используемые методы

* **Insert():** публичная и приватная версия этого метода, private выполнит всю рекурсивную работу, а также вызовет FixTree() для исправления любых возникающих нарушений. Private принимает ссылку/указатель на узел, который действует как текущий узел при обходе, и, наконец, принимает другую ссылку на узел, которая указывает на новый объект узла для сохранения в дереве.
* **InsertFixUp():** закрытый метод, принимает ссылку/указатель узла и возвращает void. Этот метод вызывается в Insert(), и именно здесь мы определяем, есть ли какие-либо нарушения в дереве; и выполняет необходимые повороты/перекрашивание. Этот метод работает, сначала проверяя, есть ли у нас нарушение, используя цикл while, затем мы идем и проверяем все случаи 1, 2 и 3 и выполняем необходимые исправления для восстановления красно-черного свойства дерева.
* Как показано на рисунках выше:
* а. Узел z красный.  
  б. Если zp — корень, то z.parent — черный.  
  в. Если дерево нарушает какое-либо из красно-черных свойств, то оно нарушает не более  
  одного из них, и нарушение относится либо к свойству 2, либо к свойству 4. Если  
  дерево нарушает свойство 2, это происходит потому, что z является корнем и является красным. . Если дерево  
  нарушает свойство 4, то это потому, что и z, и z.parent красные.
* **Find ():** реализация, используемая в обычном BST, поскольку дерево RB самобалансируется, поиск уже оптимизирован, принимает параметр int и возвращает ссылку на найденный узел, если ничего не найдено, возвращается NULL
* **DisplayTree():** просто вызывает метод RecursiveInOrder() для отображения узлов в окне консоли.
* **Delete():** этот метод работает и удаляет как обычно, затем мы вызываем DeleteFixUp(), чтобы исправить дерево, если будут какие-либо нарушения, и восстановить красно-черные свойства дерева.
* **DeleteFixUp():** этот закрытый метод принимает ссылку на узел и возвращает void. Этот метод проверяет любые нарушения в дереве и выполняет исправление, если после удаления произошли какие-либо нарушения.
* **Повороты :** реализованы RightRotate() и LeftRotate().
* **TreeSuccessor():** принимает ссылку на узел и возвращает ссылку на узел, находит преемника для данного узла.
* **Минимум():** находит минимальный узел в/поддереве данного узла.

# Объекты

**Узел:** объект узла в этой реализации имеет 4 свойства: поле данных, левый указатель, правый указатель и указатель на родительский узел. Родительская ссылка важна в этой реализации, поскольку она будет использоваться для проверки нарушений.

# Код программы:

class Program

{

static void Main(string[] args)

{

RB tree = new RB();

tree.Insert(5);

tree.Insert(3);

tree.Insert(7);

tree.Insert(1);

tree.Insert(9);

tree.Insert(-1);

tree.Insert(11);

tree.Insert(6);

tree.DisplayTree();

tree.Delete(-1);

tree.DisplayTree();

tree.Delete(9);

tree.DisplayTree();

tree.Delete(5);

tree.DisplayTree();

Console.ReadLine();

}

}

//БИБЛИИОТЕКА

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 (key == temp.data)

{

isFound = true;

item = temp;

}

}

if (isFound)

{

Console.WriteLine("{0} was found", key);

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);

}

}

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;

}

}

}