МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: Красно-чёрное дерево

Студент гр. 9303 _____ Халилов Ш.А.

Преподаватель _____ Филатов Ар.Ю.

Санкт-Петербург

2020

Цель работы.

Реализовать алгоритм красно-чёрное дерево

Задание.

Вариант №27-28. Красно-чёрное дерево.

Основные теоретические положения.

Описание алгоритма

У каждого символа, встречаемого в поданной на вход строке считается частота входа в строку. Потом на основе символов и их частот встреч строится бинарное дерево, путем разбиения строки на две части по суммарной частоте этих двух частей. Слева берется более короткая строка с меньшей суммарной частотой встреч. Потом по этому дереву каждый символ кодируется путем к нему в дереве, т. е. „0“ – левое поддерево, „1“ – провае поддерево. А декодирование происходит по обратному методу: если в кодированной строке встречается „0“, то идем по дереву в левое поддерево, если „1“, то в правое поддерево, и так пока в текущем дереве строка не будет длины 1.

Красно-чёрное дерево - двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" и "чёрный".

Сбалансированное RB-дерево бинарного поиска - это RB-дерево бинарного поиска, в котором все пути от внешних связей к корню содержат одинаковое количество черных узлов.

Узлы красятся в один из двух цветов. Если узел - красный, его сыновья - обязательно чёрные. Вставляемый узел - всегда красный. При появлении цепочки из двух красных узлов, дерево перестраивается. Красно-чёрным называется бинарное поисковое дерево, у которого каждому узлу сопоставлен

дополнительный атрибут - цвет и для которого выполняются следующие свойства:

1. Каждый узел промаркирован красным или чёрным цветом

2. Корень и конечные узлы (листья) дерева - чёрные

3. У красного узла родительский узел - черный

4. Все простые пути из любого узла **x** до листьев содержат одинокавое количества чёрных узлов

5. Чёрный узел может иметь чёрного родителя

Теорема: красно-чёрное дерево с N ключами имеет высоту h = O(log N).

Реализация

Вставка элемента

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет nil (то есть этот сын — лист). Вставляем вместо него новый элемент с nil -потомками и красным цветом. Теперь проверяем балансировку. Если отец нового элемента черный, то никакое из свойств дерева не нарушено. Если же он красный, то нарушается свойство 3, для исправления достаточно рассмотреть два случая:

1. "Дядя" этого узла тоже красный. Тогда, чтобы сохранить свойства 3 и 4, просто перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" — в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин "отцы" черные. Проверяем, не нарушена ли балансировка. Если в результате этих перекрашиваний

мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству 2.

2. "Дядя" чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком. Таким образом, свойство 3 и постоянство черной высоты сохраняются.

Удаление вершины

При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

1. Если у вершины нет детей, то изменяем указатель на неё у родителя на nil.

2. Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.

3. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины и она самая левая в нем, иначе бы мы взяли ее левого ребенка. Иными словами сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Проверим балансировку дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки

потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины:

1. Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас **x** имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.

2. Если брат текущей вершины был чёрным, то получаем три случая:

   - Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество чёрных узлов на путях, проходящих через **b**, но добавит один к числу чёрных узлов на путях, проходящих через **x**, восстанавливая тем самым влиянние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.

   - Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у **x** есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни **x**, ни его отец не влияют на эту трансформацию.

   - Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца - в чёрный, делаем

вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 3 и 4 не нарушаются. Но у **x** теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и **b** был добавлен в качестве чёрного дедушки. Таким образом, проходящие через **x** пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.

Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева. Из рассмотренных случаев ясно, что при удалении выполняется не более трёх вращений.

Выполнение работы.

Для реализации задачи были реализованы следующие класс и функции:

Node – класс, описывающий узел дерева. Каждый узел имеет ключ, цвет

,указатели на левое и правое поддеревья и еще указатель на родительские деревья.

Основные функции:

InsertRecurse() - функция ищет место для элемента и вставляет элемент в это место

Insert()- функция получает элемент и выдает его для добавления в лес, после выдает элемент для проверки свойства

CreateForest() - функция создает дерево из значений и дает их для добавления

fixProperties() - функция для исправленных свойств красно-черного дерева, после вставки

RotateLeft() - функция поворота дерева влево

RotateRight() - функция поворота дерева вправо

deleteElem() - функция получает значение и предоставляет для поиска элемент со значением, равным полученному. и после нахождения элемента,

который нужен, передать его на удаление и вызвать функцию для исправления свойств

deleteNode() - функция удаления элемента из дерева

deleteFixup() - функция для исправленных свойств красно-черного дерева, после удаления

findElem() - функция поиска элемента в дереве

CountElem() - функция для подсчета количества вхождений элемента в дерево

WriteToFile() - функция записи элементов в файл по возрастанию значения

free() - функция очистки памяти под элементом

ReadFromConsole() - функция чтения элементов из файла

ReadFromFile() - функция чтения элементов из консоли

showForestOnConcole() - функция для отображения дерева на консоли

ShowForestWithoutColor() - функция для отображения дерева на консоли без цвета

Исходный код программы представлен в приложении А. Результаты тестирования включены в приложение Б

Выводы.

В ходе выполнения работы была изучена структура данных типа красно-чёрное дерево . Написана программа, в которой был реализован класс данной структуры данных, и написаны методы для работы с ней — добавление элемента, удаление, проверка на пустоту и печать дерева на экран

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: RedBlackTree.h

```
include <iostream>
#include <fstream>
enum color_t {
     BLACK = 0,
     RED = 1
};

class Node
{
private:
     Node *parent;
     Node *left;
     Node *right;
     int key;
     color_t Color;
public:
     Node(){
          Color = RED;
          parent = left = right = nullptr;
     }
     Node(int val, color_t cl) : key(val), Color(cl) {
          parent = left = right = nullptr;
     }
     Node *&Left() { return left; }
     Node *&Right() { return right; }
     Node *&Parent() { return parent; }
     int getVal() { return key; }
     void setValue(int val) { key = val; }
     color_t& color(){ return Color; }
     ~Node(){}
};
```

```
void createForest(Node *&root, int *arr, int size);
void showForestOnConcole(Node *root, int n);
void ShowForestWithoutColor(Node *root, int);
void fixProperties(Node *elem);
bool isRed(Node *n);
bool isBlack(Node *n);
bool isLeft(Node *n);
bool isRight(Node *n);
Node *GrandParent(Node *n);
Node *Parent(Node *n);
Node *Brother(Node *n);
Node *Uncle(Node *n);
Node * Root(Node* elem);
void InsertRecurse(Node *&root, Node *n);
Node *Insert(Node *root, Node *n);
Node *findElem(Node *root, int key);
void RotateLeft(Node *n);
void RotateRight(Node *n);
void deleteElem(Node *&root, int key);
void deleteNode(Node *elem);
void deleteFixup(Node *elem);
bool isLeaf(Node *elem);
bool isNotLeaf(Node *elem);
void free(Node *elem);
void WriteToFile(Node *elem, std::ofstream &fout);
Node* ReadFromConsole(Node *head);
Node* ReadFromFile( Node* head );
int CountElem(Node *elem, int key);
```

Название файла: RedBlackTree.cpp

```
#include "RedBlackTree.h"
using namespace std;
```

```
int main()
{
    setlocale( 0, "ru-RU");
    bool flag = true;
    int options;
    bool hesTree = false;
    Node *root;
    int elem;
    string name;
    ofstream output;
    while (flag) {
          cout << "\nвыберите один из вариантов:\n";
          cout << "[1] Добавить элемент:\n";
          cout << "[2] Добавить элементы из файла:\n";
          cout << "[3] Добавить элементы из консоли:\n";
          cout << "[4] Удалить элемент:\n";
          cout << "[5] Показать дерево:\n";
          cout << "[6] записать в файл по возрастанию:\n";
          cout << "[7] подсчитать количество вхождений:\n";
          cout << "[8] Выход:\n";
          cout << ">>> ";
          cin >> options;
          switch (options)
          {
               case 1:
                    cout << "Пожалуйста, напишите элемент (целые
числа): ";
               cin >> elem;
                    root = Insert(root, new Node(elem, RED));
                    break;
               case 2:
                    root = ReadFromFile( root );
                    break;
```

```
                case 3:
                     root = ReadFromConsole( root );
                     break;
                case 4:
                     cout << "Пожалуйста, напишите элемент (целые
числа): ";
                     cin >> elem;
                     deleteElem( root, elem );
                     break;
                case 5:
                     showForestOnConcole(root, 0);
                     // ShowForestWithoutColor(root, 0 );
                     break;
                case 6:
                     cout << "пожалуйста, напишите имя файла: ";
                     cin >> name;
                     output.open( name , ios_base::out );
                     WriteToFile( root, output);
                     output.close();
                     break;
                case 7:
                     cout << "Пожалуйста, напишите элемент (целые
числа): ";
                     cin >> elem;
                     cout    <<    "элемент    существует    "<<
CountElem( root, elem )<<" раз\n";
                     break;
                case 8:
                     cout << "пока!!\n";
                     flag = false;
                     break;
                default:
                break;
     }
```

```
     }
     return 0;
}

int CountElem(Node *elem, int key){
    int l = 0, r = 0;
    if(elem->Left())
        l = CountElem(elem->Left(), key);
    if(elem->Right())
        r = CountElem(elem->Right(), key);
    if( elem->getVal() == key ){
        return 1+l+r;
    }
    return l+r;
}
Node* ReadFromFile( Node* head ){

    string name;
    cin >> name;
    ifstream input( name );
    int len;
    input >> len;
    int *OutArray = new int[len];
    cout << ">>> Arr["<< len <<"]:\n{ ";
    for (int i = 0; i < len; i++)
    {
        input >> OutArray[i];
        cout << OutArray[i] <<" ";
    }
    cout << "}\n";
    createForest(head, OutArray , len );
    delete [] OutArray;
    return head;
```

```
}
Node* ReadFromConsole(Node *head){
    int len;
    int *OutArray = new int[len];
    for (int i = 0; i < len; i++)
    {
        cin >> OutArray[i];
    }
    cout << "\n";
    createForest(head, OutArray , len );
    delete [] OutArray;
    return head;
}
void WriteToFile(Node *elem, ofstream &fout)
{
    if( elem->Left() ){

         WriteToFile(elem->Left(), fout);
    }
     fout << elem->getVal() <<", ";
    if( elem->Right() ){

         WriteToFile(elem->Right(), fout);
    }
}
bool isLeaf(Node *elem){
     return elem == nullptr;
}
bool isNotLeaf(Node *elem){
     return elem != nullptr;
}
Node * Root(Node* elem){
     Node *root = elem;
    if(root){
```

```
        while (root->Parent())
        {
            root = root->Parent();
        }
    }
    return root;
};
Node *findElem(Node *root, int key)
{
    Node *tmp = root;
    while (tmp != nullptr)
    {
        if (tmp->getVal() == key)
        {
            return tmp;
        }
        if (tmp->getVal() > key)
        {
            tmp = tmp->Left();
        }
        else
        {
            tmp = tmp->Right();
        }
    }
    return nullptr;
}
void showForestOnConcole(Node *root, int n)
{
    if( !root || root == nullptr ){
        for (int i = 0; i < n; i++)   cout << "\t";
        cout << "\x1b[40;40m    \x1b[0m \n";
        return;
    }
```

```
     showForestOnConcole(root->Left(), n + 1);
     for (int i = 0; i < n; i++)
     {
          cout << "\t";
     }
     cout << "\x1b[40;4" + std::to_string(root->color()) + "m "
+ to_string(root->getVal()) + " \x1b[0m \n";
     showForestOnConcole(root->Right(), n + 1);
     cout << "\n";
}
void ShowForestWithoutColor(Node *root, int n)
{
     if( !root ){
          for (int i = 0; i < n; i++)
            cout << "\t";
          cout << "[ ]\n";
          return;
     }
     ShowForestWithoutColor(root->Left(), n + 1);
     for (int i = 0; i < n; i++)
        cout << "\t";
    cout << root->getVal();
    if(root->color() == BLACK)
        cout << "[x]\n" ;
    else
        cout << "[-]\n" ;
     ShowForestWithoutColor(root->Right(), n + 1);
}
void free(Node *elem){
     if( isLeaf( elem )){
        return;
    }
    if( isLeft ( elem ) ) {
        Parent(elem )->Left() = nullptr;
```

```
    }
    else if( isRight ( elem )  )
    {
        Parent(elem)->Right() = nullptr;
    }
     delete elem;
     elem = nullptr;
}
//! Helper functions:

bool isRed(Node *elem)
{
     if (elem != nullptr)
          return elem->color() == RED;
     return false;
}
bool isBlack(Node *elem)
{
     if (elem != nullptr){
          return elem->color() == BLACK;
     }
     return true;
}
bool isLeft(Node *elem)
{
     if (Parent(elem))
          return Parent(elem)->Left() == elem;
     return false;
}
bool isRight(Node *elem)
{
     if (Parent(elem))
          return Parent(elem)->Right() == elem;
     return false;
```

```
}

Node *Parent(Node *elem)
{
     return elem == nullptr ? nullptr : elem->Parent();
}

Node *GrandParent(Node *elem)
{
     return Parent(Parent(elem));
}

Node *Brother(Node *elem)
{
     if (Parent(elem))
     {
          if (isLeft(elem))
          {
               return Parent(elem)->Right();
          }
          else
          {
               return Parent(elem)->Left();
          }
     }
     return nullptr;
}

Node *Uncle(Node *elem)
{
     return Brother(Parent(elem));
}

//! start algorthim
```

```
void RotateLeft(Node *elem)
{
     Node *Son = elem->Right();
    Node *parent = Parent(elem );
     Son->Parent() = parent;
     if ( parent )
     {
          if (isLeft(elem))
          {
               parent->Left() = Son;
          }
          else
          {
               parent->Right() = Son;
          }
     }

     elem->Right() = Son->Left();
     if (Son->Left())
     {
          Son->Left()->Parent() = elem;
     }
     elem->Parent() = Son;
     Son->Left() = elem;
}
void RotateRight(Node *elem)
{
     Node *Son = elem->Left();
    Node *parent = Parent(elem );
     Son->Parent() = parent;
     if (parent)
     {
          if (isLeft(elem))
          {
```

```
                parent->Left() = Son;
            }
            else
            {
                parent->Right() = Son;
            }
        }
        elem->Left() = Son->Right();

        if ( Son->Right() )
            Son->Right()->Parent() = elem;
        elem->Parent() = Son;
        Son->Right() = elem;
}
void InsertRecurse(Node *&current, Node *elem)
{
        if (current != nullptr)
        {
            if (current->getVal() > elem->getVal())
            {
                if (current->Left() != nullptr)
                {
                    InsertRecurse(current->Left(), elem);
                    return;
                }
                current->Left() = elem;
            }
            else{
                if (current->Right() != nullptr)
                {
                    InsertRecurse(current->Right(), elem);
                    return;
                }
                current->Right() = elem;
```

```
            }
        }
        elem->Parent() = current;
}
void fixProperties(Node *elem)
{
        if (!Parent(elem))
        {
            elem->color() = BLACK;
            return;
        }
        else if (isBlack(Parent(elem)))
        {
            return;
        }
        else if (Uncle(elem) != nullptr && isRed(Uncle(elem)))
        {
            Parent(elem)->color() = BLACK;
            Uncle(elem)->color() = BLACK;
            GrandParent(elem)->color() = RED;
            fixProperties(GrandParent(elem));
            return;
        }
        else if (isRight(elem) && isLeft(Parent(elem)))
        {
            RotateLeft(Parent(elem));
            elem = elem->Left();
        }
        else if (isLeft(elem) && isRight(Parent(elem)))
        {
            RotateRight(Parent(elem));
            elem = elem->Right();
        }
```

```
      Parent(elem)->color() = BLACK;
      GrandParent(elem)->color() = RED;
      if (isLeft(elem) && isLeft(Parent(elem)))
      {
           RotateRight(GrandParent(elem));
      }
      else
      {
           RotateLeft(GrandParent(elem));
      }
}
Node *Insert(Node *root, Node *elem)
{
      InsertRecurse(root, elem);
      fixProperties(elem);
      root = elem;
      while (root->Parent())
      {
           root = root->Parent();
      }
      return root;
}
void createForest(Node *&root, int *arr, int size)
{
      for (int i = 0; i < size; i++)
      {
           root = Insert(root, new Node(arr[i], RED));
      }
}
void deleteElem(Node*& root, int key)
{
    if(!root) {
        cout << "Ð¿ÑƒÑ‚Ð¾Ð¹ Ð´ÐµÑ€ÐµÐ²Ð¾!!!\n";
        return;
```

```
    }

    Node *elem = findElem( root, key);
    if(!elem) {
        cout << "ÑÑ,Ð¾Ñ, "<< key <<" ÑÐ»ÐµÐ¼ÐµÐ½Ñ, Ð½Ðµ ÑÑƒÑ
%ÐµÑÑ,Ð²ÑƒÐµÑ, Ð² ÑÑ,Ñ€ÑƒÐ°Ñ,ÑƒÑ€Ðµ!!!\n";
        return;
    }
    if( !Parent( elem )){
        if( elem->Left() && !elem->Right()){
            if(!elem->Left()->Left() || !elem->Left()->Right()){
                root = elem->Left();
                root->Parent() = nullptr;
                root->color() = BLACK;
                free( elem );
                return;
            }
        }
        else if( elem->Right() && !elem->Left() )
        {
            if(!elem->Right()->Left() && !elem->Right()-
>Right()){
                root = elem->Right();
                root->Parent() = nullptr;
                root->color() = BLACK;
                free( elem );
                return;
            }
        }
        else if( !elem->Left() && !elem->Right() )
        {
            free( elem );
            root = nullptr;
            return;
```

```
        }
    }
    deleteNode(elem);
    root = Root(root);
}
void deleteFixup(Node *elem) {

     Node *root = Root( elem );

    while (elem != root && isBlack( elem )) {
        if ( isLeft( elem ) ) {
            Node *w = Parent( elem )->Right();
            if (w->color() == RED) {
                w->color() = BLACK;
                Parent( elem )->color() = RED;
                RotateLeft( Parent( elem ) );
                w = Parent( elem )->Right();
            }
            if ((isLeaf( w->Left()) || isBlack( w-
>Left() ))&&( isLeaf( w->Right()) || isBlack( w->Right() ))) {
                w->color() = RED;
                elem = Parent(elem);
            }
            else {
                if (isBlack( w->Right() )) {
                    w->Left()->color() = BLACK;
                    w->color() = RED;
                    RotateRight ( w );
                    w = Parent( elem )->Right();
                }
                w->color() = Parent( elem )->color();
                Parent( elem )->color() = BLACK;
                w->Right()->color() = BLACK;
                RotateLeft ( Parent( elem ) );
```

```
                elem = root;
            }
        } else {
            Node *w = Parent( elem )->Left();
            if (w->color() == RED) {
                w->color() = BLACK;
                Parent( elem )->color() = RED;
                RotateRight ( Parent( elem ) );
                w = Parent( elem )->Left();
            }
            if ( (isLeaf(w->Right()) || isBlack( w->Right())) &&
                    (isLeaf( w->Left() ) || isBlack( w-
>Left()))){
                w->color() = RED;
                elem = Parent( elem );
            } else {
                if ( isBlack( w->Left() )) {
                    w->Right()->color() = BLACK;
                    w->color() = RED;
                    RotateLeft( w );
                    w = Parent( elem )->Left();
                }
                w->color() = Parent( elem )->color();
                Parent( elem )->color() = BLACK;
                w->Left()->color() = BLACK;
                RotateRight ( Parent( elem ) );
                elem = root;
            }
        }
    }
    elem->color() = BLACK;
}
void deleteNode(Node *elem) {
       Node *x, *descendant;
```

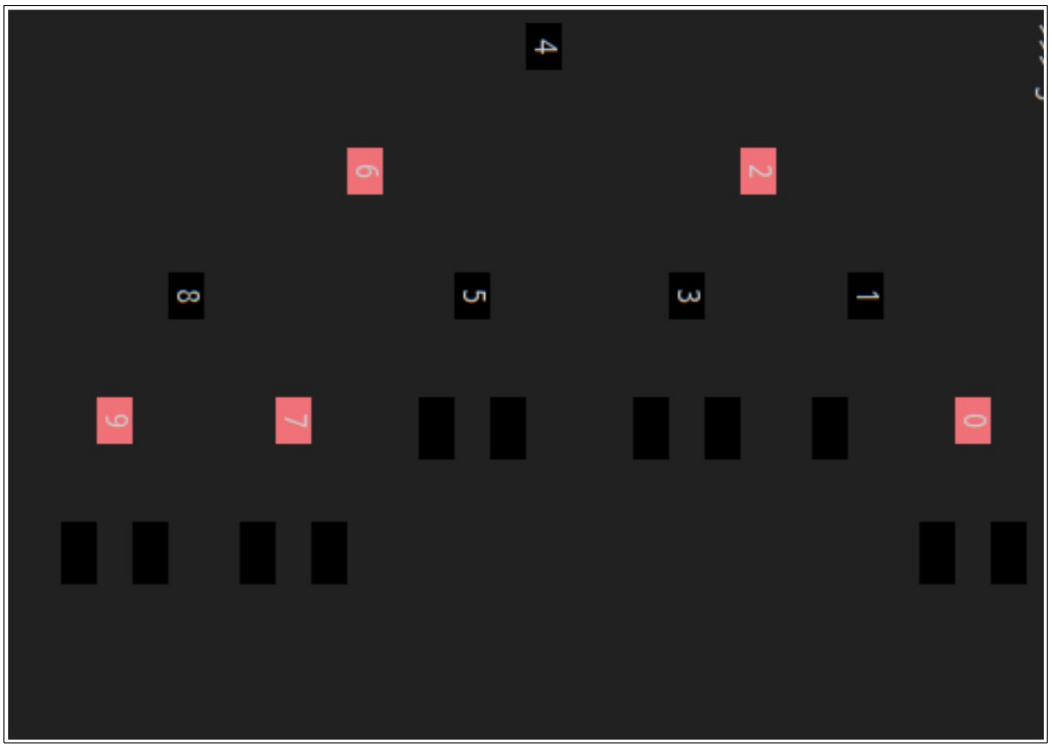
```
if (!elem) return;
 Node *root = Root(elem);
if (isLeaf(elem->Left()) || isLeaf( elem->Right() )) {
    descendant = elem;
} else {
    descendant = elem->Right();
    while (descendant->Left()){
        descendant = descendant->Left();
    }
}
 if ( descendant->Left() || descendant->Right() ) {
      if (descendant->Right() )
           x = descendant->Right();
      else
      x = descendant->Left();

 x->Parent() = descendant->Parent();
      if (Parent( descendant ))
    {
        if ( isLeft( descendant ) )
            Parent( descendant )->Left() = x;
        else
            Parent(descendant)->Right() = x;
    }
      if (descendant != elem) {
      elem->setValue( descendant->getVal() );
    }
    if (isBlack( descendant ))
    {
        deleteFixup (x);
    }
 }
 else
 {
```

```
            if (descendant != elem)
                  elem->setValue( descendant->getVal() );
            if (isBlack( descendant ))
                  deleteFixup (descendant);
      }
      free(descendant);
}
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Тест с значениями [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]



Тест с значениями: [3, 69, 71, 78, 9, 54, 9, 11, 42, 40, 45, 82, 91, 96, 83]