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-черное дерево

Студент гр. 9304 _____ Тиняков С.А.

Преподаватель _____ Филатов Ар.Ю.

Санкт-Петербург
2020

Цель работы.

Изучить структуру данных красно-чёрное дерево. Реализовать данную структуру на языке программирования C++.

Задание.

Бинароное дерево поиска: красно-чёрное дерево.

По заданной последовательности элементов *Elem* построить данную структуру данных.

Для построенной структуры проверить, входит ли в неё элемент *e* типа *Elem*, и если входит, то в скольких экземплярах. Добавить элемент *e* в структуру данных. Предусмотреть возможность потвторного выполнения с другим элементом.

Основные теоретические положения.

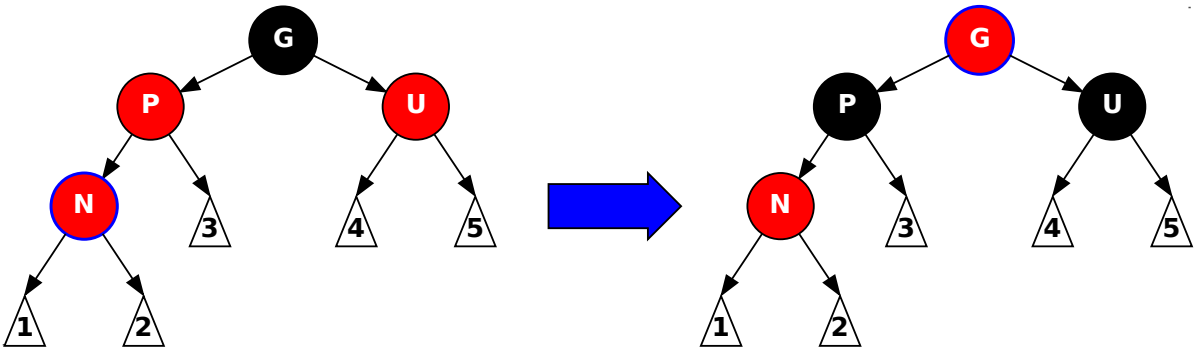
Красно-чёрное дерево — это бинарное самобалансирующееся дерево поиска. Каждый узел имеет дополнительный параметр — цвет: красный или чёрный, который используется для балансировки дерева при вставке и удалении. Бинарное дерево поиска является красно-чёрным деревом если:

1. Каждый узел красный или чёрный
2. Корень дерева всегда чёрный
3. Все пустые узлы (*NIL*) чёрные
4. У красного узла оба сына чёрные
5. Любой путь от заданного узла до пустого (*NIL*) потомка проходит через одинаковое количество черных вершин — чёрная высота.

Выполнение работы.

Введём следующие обозначения: текущий узел(*N*) — узел, для которого выполняется перебалансировка, отец(*P*) — узел, для которого данный является сыном, дед(*G*) — узел, для которого отец является сыном, дядя(*U*) — сын деда, который не является отцом. Алгоритм вставки работает следующим образом: сначала новый элемент вставляется как в обычное бинарное дерево поиска. Новый узел имеет красный цвет. Далее делается перебалансировка дерева. Возможны четыре случая:

1. Узел *N* является корнем. В этом случае узел *N* перекрашивается в чёрный цвет
2. Отец *P* — чёрный. В этом случае ни одно из требования красно-чёрного дерева не нарушается. Никаких дополнительных действий делать не надо.
3. Отец *P* и дядя *U* — красные. В этом случае узлы *P* и *U* перекрашиваются в чёрный цвет, а дед *G* — в красный. Пример приведён на рис. 1. Далее вызывается перебалансировка для узла *G*.



Рисунко 1 — Пример для случая 3

4. Отец P — красный, дядя U — чёрный. В этом случае необходимо сделать поворот дерева(возможно два). Сначала проверяется, находятся ли узел N и P в одной стороне. Т.е. если отец — левый сын, то и текущий узел тоже должен быть левым сыном. Аналогично для правой стороны. Далее будет считаться, что отец — левый сын. Для ситуации, когда отец — правый сын, все действия аналогичны, только симметричны. Если отец и текущий узел в разных сторонах, то сначала нужно сделать малый поворот: узлы P и N меняются местами, при этом отец становится левым сыном текущего узла, а левый сын узла N становится правым сыном узла P. Пример приведён на рис. 2.

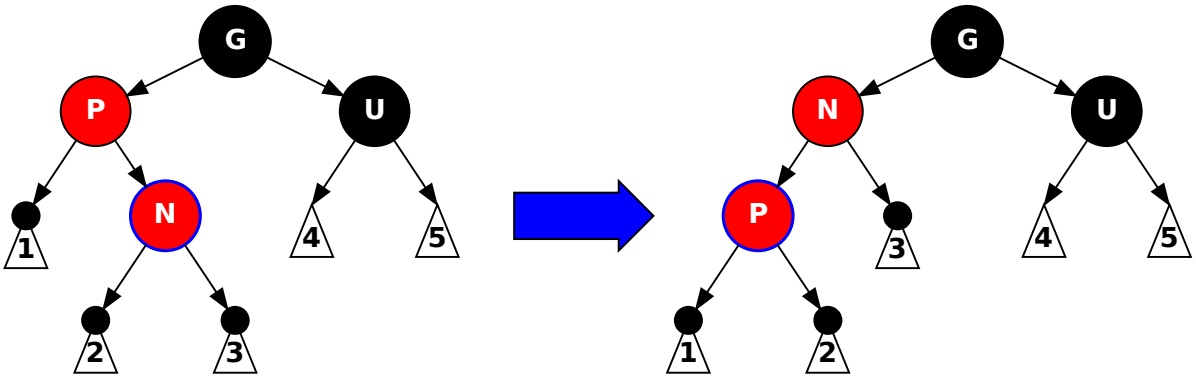


Рисунок 2 — Пример малого поворота

Если был выполнен малый поворот, то в большом повороте узел *P* — это узел *N*, а *N* — это *P*. Если отец и текущий узел находятся в одной стороне, то выполняется большой поворот: отец помещается на место деда, перекрашивается в чёрный цвет, правым сыном становится дед. Дед перекрашивается в красный цвет, левым сыном становится бывший правый сын отца. Пример приведён на рис. 3.

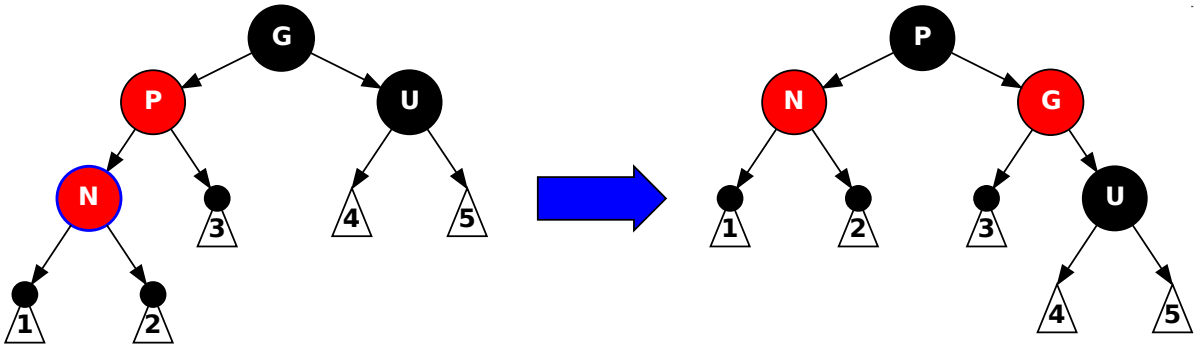


Рисунок 3 — Пример большого поворота

Программа работает со стандартными потоками входа и выхода. На вход программа получает число*(n)* — размер массива, затем считывает *n* чисел, после чего следует число, которое необходимо найти. Программа выводит элементы дерева в ЛКП порядке, затем количество найденного элемента(0 или 1). Также программа может вывести дерево после каждой вставки. Узлы выводятся цветом, который они имеют(красный или чёрный), рёбра выводятся синим. Также у каждого конечного пустоого (*NIL*) узла выводится чёрная высота от корня дерева.

Класс *RedBlackTreeNode* является узлом дерева. В себе хранит указатели на сыновей, на родителя, также хранит цвет и информацию о том, левый или правый это потомок. Также есть конструкторы и операторы копирования и перемещения. Также перегружен оператор вывода. Класс *RedBlackTree* является красно-чёрным деревом. Имеет операторы и конструкторы копирования и перемещения. Метод *Insert* вставляет элемент в дерево и вызывает перебалансировку. Метод *Find* ищет значение в дереве. Метод *Balance* выполняет перебалансировку дерева. Метод *Recolor* выполняет перекраску дерева(случай 3). Метод *SmallRotate* выполняет малый

поворот(случай 4, отец и текущий узел в разных сторонах). Метод *BigRotate* выполняет большой поворот(случай 4, отец и текущий узел в одной стороне). Метод *Rotate* выполняет поворот дерева, если это необхоимо. Для класс *RedBlackTree* перегружен оператор вывода, который вызывает метод *Print*, который выводит дерево. Метод *PrintData* выводит элементы дерева в ЛКП порядке.

Разработанный программный код см. в приложении А.

Тестирование.

Тестирование проводилось при помощи *python*-скрипта. Для заданного размера генерировался набор случайны чисел, которые подавались на вход программе. Также случайно генерировалось число для поиска. Проверка происходила через удаление дупликатов и сортировки средствами языке программирования *Python*. Набор размер для тестирования: *25, 73, 549, 1091*.

Выводы.

Была изучена структура данных красно-чёрное дерево. Данное вид структуры был реализован на языке программирования C++.

Была разработана программа, которая строит красно-чёрное дерево по заданному набору чисел и находит какое-то число в этом дерево. При реализации классов использовались шаблоны. Для класса *RedBlackTreeNode* и *RedBlackTree* перегружены операторы вывода. Также была реализованна возможность наглядного представления дерева с использованием цветов. В реализации алгоритма использовались такие возможности *17*-ого стандарта C++, как лямбда-функии и умные указатели.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Source/lab5.cpp

```
#include <iostream>
#include <memory>

#define PRINT

#ifdef PRINT

#define RED "\033[1;31m"
#define BLACK "\033[1;30m"
#define BLUE "\033[1;34m"
#define NORMAL "\033[0m"

#endif

template<typename T>
class RedBlackTreeNode{
public:
    bool is_red = true;
    bool is_left = false;
    T data;
            std::shared_ptr<RedBlackTreeNode<T>>   left{nullptr},
right{nullptr};
    std::weak_ptr<RedBlackTreeNode<T>> parent;
    RedBlackTreeNode() = default;

    RedBlackTreeNode(T& data){
        this->data = data;
    }

    RedBlackTreeNode(T&& data){
        this->data = data;
    }

    ~RedBlackTreeNode() = default;

    RedBlackTreeNode(const RedBlackTreeNode<T>& node){
        data = node.data;
        left = node.left;
        right = node.right;
        parent = node.parent;
        is_red = node.is_red;
        is_left = node.is_left;
    }

    RedBlackTreeNode& operator=(const RedBlackTreeNode<T>& node){
        if(&node == this) return *this;
        data = node.data;
        left = node.left;
        right = node.right;
        parent = node.parent;
```

```
            is_red = node.is_red;
            is_left = node.is_left;
            return *this;
        }

        RedBlackTreeNode(const RedBlackTreeNode<T>&& node){
            data = std::move(node.data);
            left = std::move(node.left);
            right = std::move(node.right);
            parent = std::move(node.parent);
            is_red = node.is_red;
            is_left = node.is_left;
        }

        RedBlackTreeNode& operator=(const RedBlackTreeNode<T>&& node)
{
            if(&node == this) return *this;
            data = std::move(node.data);
            left = std::move(node.left);
            right = std::move(node.right);
            parent = std::move(node.parent);
            is_red = node.is_red;
            is_left = node.is_left;
            return *this;
        }
    };

    template<typename T>
    std::ostream&  operator<<(std::ostream&  os,  RedBlackTreeNode<T>&
node){
        os << "{data: " << node.data << "; is_red: " << node.is_red
           << "; is_left: " << node.is_left << "; left: ";
        if(node.left) os << node.left->data;
        else os << "nullptr";
        os << "; right: ";
        if(node.right) os << node.right->data;
        else os << "nullptr";
        os << "; parent: ";
        if(node.parent.lock()) os << node.parent.lock()->data;
        else os << "nullptr";
        os << ";}";
        return os;
    }

    template<typename T>
    class RedBlackTree{
    using NodePtr = std::shared_ptr<RedBlackTreeNode<T>>;
    protected:
        NodePtr head{nullptr};

        bool IsBlackNode(NodePtr node){
            return (node == nullptr || !node->is_red);
        }

        bool IsRedNode(NodePtr node){
            return !IsBlackNode(node);
```

```
        }

        void Recolor(NodePtr node){
            if(!node->parent.lock()){
                node->is_red = false;
                return;
            }
            if(!node->parent.lock()->parent.lock()){
                auto parent = node->parent.lock();
                parent->is_red = false;
                return;
            }
            auto parent = node->parent.lock();
            auto grandparent = parent->parent.lock();
               auto uncle = (parent->is_left ? grandparent->right :
grandparent->left);
            if(IsRedNode(parent) && IsRedNode(uncle)){
                parent->is_red = uncle->is_red = false;
                grandparent->is_red = true;
                Balance(grandparent);
            }else if(IsRedNode(parent)){
                parent->is_red = false;
            }
        }

        void SmallRotate(NodePtr node){
            if(!node) return;
            if(!node->parent.lock()) return;
            if(!node->parent.lock()->parent.lock()) return;
            auto parent = node->parent.lock();
            auto grandparent = parent->parent.lock();
            node->parent = grandparent;
            if(parent->is_left){
                grandparent->left = node;
                node->is_left = true;
                parent->right = node->left;
                if(node->left){
                    node->left->is_left = false;
                    node->left->parent = parent;
                }
                parent->parent = node;
                node->left = parent;
            }else{
                grandparent->right = node;
                node->is_left = false;
                parent->left = node->right;
                if(node->right){
                    node->right->is_left = true;
                    node->right->parent = parent;
                }
                parent->parent = node;
                node->right = parent;
            }
        }

        void BigRotate(NodePtr node){
```

```
            if(!node) return;
            if(!node->parent.lock()) return;
            if(!node->parent.lock()->parent.lock()) return;
            auto parent = node->parent.lock();
            auto grandparent = parent->parent.lock();
               auto uncle = (parent->is_left ? grandparent->right :
grandparent->left);
            std::swap(parent->data, grandparent->data);
            if(parent->is_left){
                parent->left = parent->right;
                if(parent->left) parent->left->is_left = true;
                parent->right = uncle;
                if(uncle) uncle->parent = parent;
                grandparent->right = parent;
                parent->is_left = false;
                grandparent->left = node;
                node->parent = grandparent;
            }else{
                parent->right = parent->left;
                if(parent->right) parent->right->is_left = false;
                parent->left = uncle;
                if(uncle) uncle->parent = parent;
                grandparent->left = parent;
                parent->is_left = true;
                grandparent->right = node;
                node->parent = grandparent;
            }
        }

        void Rotate(NodePtr node){
            if(!node->parent.lock()) return;
            if(!node->parent.lock()->parent.lock()) return;
            auto parent = node->parent.lock();
              auto uncle = (parent->is_left ? parent->parent.lock()-
>right :
                            parent->parent.lock()->left);
            if(IsRedNode(parent) && IsBlackNode(uncle)){
                if(node->is_left != parent->is_left){
                    SmallRotate(node);
                    BigRotate(parent);
                    return;
                }else{
                    BigRotate(node);
                }
            }
            return;
        }

        void Balance(NodePtr node){
            Rotate(node);
            Recolor(node);
        }

    public:
        RedBlackTree() = default;
        ~RedBlackTree() = default;
```

```
        RedBlackTree(const RedBlackTree<T>& tree){
               auto Copy = [](NodePtr parent, NodePtr& dest, const
NodePtr& src,  auto&& Copy)->void{
                if(src == nullptr) return;
                    dest = std::make_shared<RedBlackTreeNode<T>>(src-
>data);
                dest->parent = parent;
                Copy(dest, dest->left, src->left, Copy);
                Copy(dest, dest->right, src->right, Copy);
            };
            Copy(nullptr, head, tree.head, Copy);
        }

        RedBlackTree& operator=(const RedBlackTree<T>& tree){
            if(&tree == this) return *this;
               auto Copy = [](NodePtr parent, NodePtr& dest, const
NodePtr& src,  auto&& Copy)->void{
                if(src == nullptr) return;
                    dest = std::make_shared<RedBlackTreeNode<T>>(src-
>data);
                dest->parent = parent;
                Copy(dest, dest->left, src->left, Copy);
                Copy(dest, dest->right, src->right, Copy);
            };
            Copy(nullptr, head, tree.head, Copy);
            return *this;
        }

        RedBlackTree(RedBlackTree<T>&& tree){
            head = std::move(tree.head);
        }

        RedBlackTree& operator=(RedBlackTree<T>&& tree){
            if(&tree == this) return *this;
            head = std::move(tree.head);
            return *this;
        }

        void Insert(T new_data){
                                              auto     new_node     =
std::make_shared<RedBlackTreeNode<T>>(new_data);
            if(head == nullptr){
                head = new_node;
                Balance(new_node);
                return;
            }
            NodePtr cur = head;
            while(true){
                if(new_data == cur->data){
                    cur->data = new_data;
                    break;
                }
                if(new_data < cur->data){
                    if(cur->left == nullptr){
                        cur->left = new_node;
```

```
                        new_node->parent = cur;
                        new_node->is_left = true;
                        break;
                    }else{
                        cur = cur->left;
                    }
                }
                if(new_data > cur->data){
                    if(cur->right == nullptr){
                        cur->right = new_node;
                        new_node->parent = cur;
                        new_node->is_left = false;
                        break;
                    }else{
                        cur = cur->right;
                    }
                }
            }
            Balance(new_node);
        }

        void PrintData(std::ostream& os = std::cout){
            auto print = [&os](NodePtr node, auto&& print)->void{
                if(node == nullptr) return;
                print(node->left, print);
                os << node->data << " ";
                print(node->right, print);
            };
            print(head, print);
            os << "\n";
        }

        #ifdef PRINT
        void Print(std::ostream& os = std::cout){
            int deep = 0;
            std::string tab = "";
               auto print = [&tab, &deep, &os](NodePtr node, auto&&
print)->void{
                if(node == nullptr){
                      os << BLACK << "nullptr ("<< deep+1 << ")\n" <<
NORMAL;
                    return;
                }
                  if(node->is_red) os << RED <<node->data << "\n" <<
NORMAL;
                else os << BLACK << node->data << "\n" << NORMAL;
                if(!node->is_red) deep++;
                os << BLUE << tab << "|    " << "\n" << tab << "|---"
;
                tab += "|    ";
                print(node->right, print);
                tab = tab.substr(0, tab.size() - 5);
                os << BLUE << tab << "|    " << "\n" << tab << "|---"
;
                tab += "     ";
                print(node->left, print);
```

```
                tab = tab.substr(0, tab.size() - 5);
                if(!node->is_red) deep--;
            };
            print(head, print);
            os << "\n";
        }
        #endif

        int Find(T find_data){
            int count = 0;
              auto find = [&find_data, &count](NodePtr node, auto&&
find)->void{
                if(node == nullptr) return;
                if(find_data == node->data) count++;
                if(find_data < node->data) find(node->left, find);
                else find(node->right, find);
            };
            find(head, find);
            return count;
        }

    };

    template<typename T>
    std::ostream& operator<<(std::ostream& os, RedBlackTree<T>& rbt){
        rbt.Print(os);
        return os;
    }


    int main(){
        int count, find;
        RedBlackTree<int> rbt;
        std::cin >> count;
        for(int i = 0; i < count; i++){
            int temp;
            std::cin >> temp;
            rbt.Insert(temp);
            #ifdef PRINT
            std::cout << rbt;
            #endif
        }
        std::cin >> find;
        rbt.PrintData();
        std::cout << "Count of element: " << rbt.Find(find) << "\n";

        return 0;
    }
```

Название файла: Makefile

```
LAB = lab5

.PHONY: all clean
```

```
all: run_tests

$(LAB): Source/$(LAB).cpp
	g++ $< -g -std=c++17 -o $@

run_tests: $(LAB)
	python3 test.py

clean:
	rm -rf $(LAB)
```

Название файла: test.py

```
import unittest
import subprocess
import os
import filecmp
import random

class TestParamAnalyzer(unittest.TestCase):

    cwd = os.getcwd()
    test_dir = './Tests/'
    tests = []

    @classmethod
    def setUpClass(self):
        print('Start Testing...')

    def start_test(self):
        out = 'output.test'
        check_out = 'check_output.test'
        in_file = 'input.test'
        input_str = ''
        arr = []
        for i in range(self.size):
            arr.append(random.randint(0, 1024))
            input_str += str(arr[-1]) + ' '
        print('List size:', self.size)
        print("List for search: [ ", input_str, ']', sep='')
        output_str = ''
        arr = list(dict.fromkeys(arr))
        for i in sorted(arr):
            output_str += str(i) + ' ';
        input_str = str(self.size) + '\n' + input_str
        finded = []
        for i in range(0, self.search_count):
            search_element = random.randint(0, 1024)
            while search_element in finded:
                search_element += 1
            finded.append(search_element)
            print("Search element:", search_element)
            in_str = input_str + '\n' + str(search_element)
              check_str = output_str + '\nCount of element: ' +
str(arr.count(search_element)) + '\n'
```

```
                with open(out, 'w') as f_out:
                     p = subprocess.run(['./lab5', ], cwd = self.cwd,
stdout = f_out, text = True, input = in_str)
                with open(out, 'r') as f_out:
                    str_out = f_out.read()
                          str_out  =  str_out[str_out.rfind('\n',  0,
(str_out.rfind('\nCount') - 1)) + 1:]
                print('Output:', str_out, sep='\n')
                print('Correct output:', check_str, sep='\n')
                self.assertTrue(str_out == check_str)

        def test_1(self):
            self.size = 25
            self.search_count = 25
            self.start_test()

        def test_2(self):
            self.size = 73
            self.search_count = 25
            self.start_test()

        def test_3(self):
            self.size = 549
            self.search_count = 25
            self.start_test()

        def test_4(self):
            self.size = 1091
            self.search_count = 25
            self.start_test()

        def tearDown(self):
            files = []
            files.extend([f for f in os.listdir('./')
                          if f.endswith('.test')])
            for f in files:
                if os.path.isfile(f):
                    os.remove(f)


    if __name__ == "__main__":
        unittest.main()
```