Национальный исследовательский университет
«Высшая Школа Экономики»
Московский институт электроники и математики им. А. Н. Тихонова

Лабораторная работа №2

Выполнил:

Дёма Иван Романович, СКБ212

Проверил:

Драчёв Григорий Александрович

Москва,
2024

1. Задание

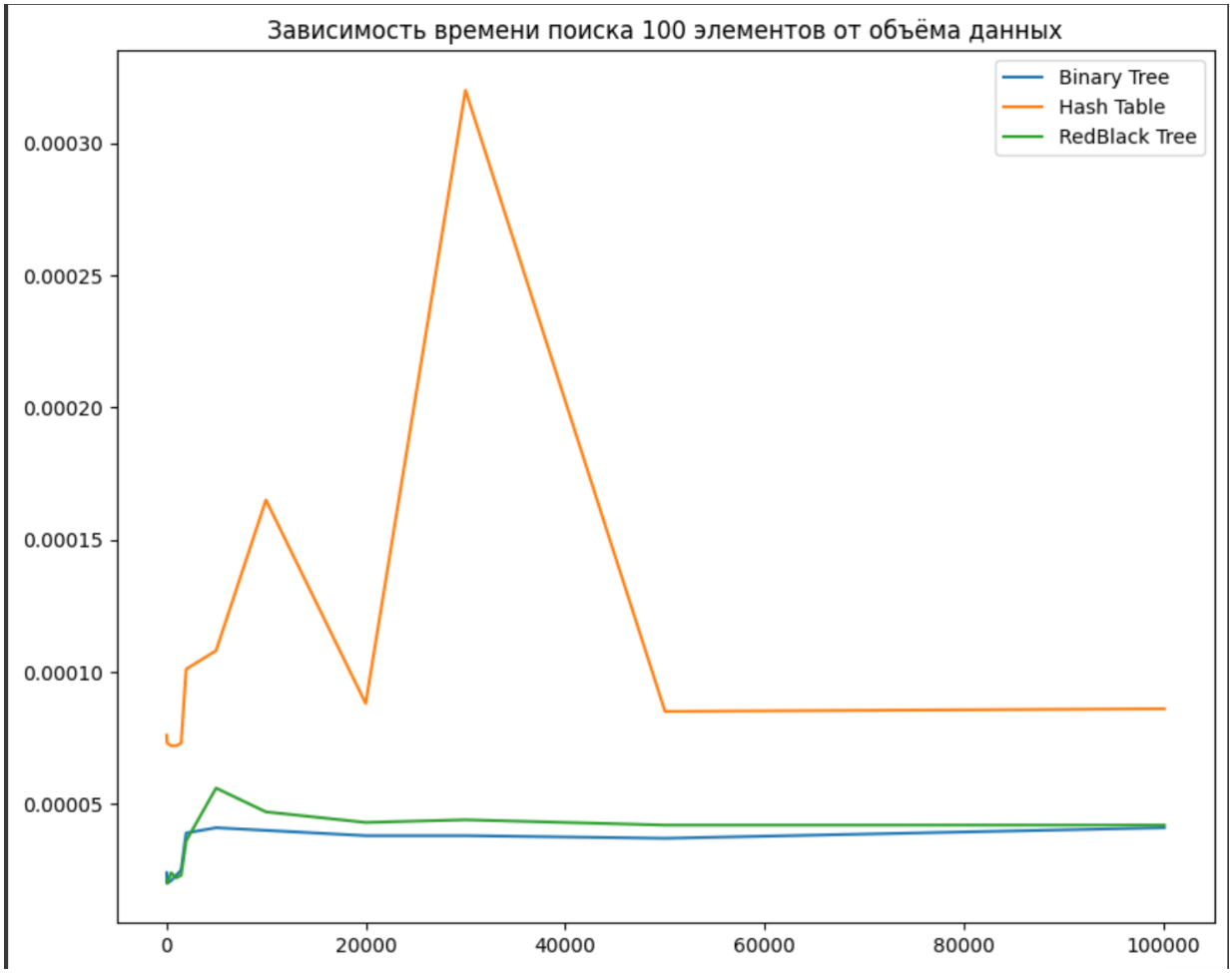
Реализовать поиск заданного элемента в массиве объектов по ключу в соответствии с вариантом (ключом является первое НЕ числовое поле объекта) следующими методами:

- с помощью бинарного дерева поиска
- с помощью красно-черного дерева
- с помощью хэш таблицы

2. Сравнение алгоритмов

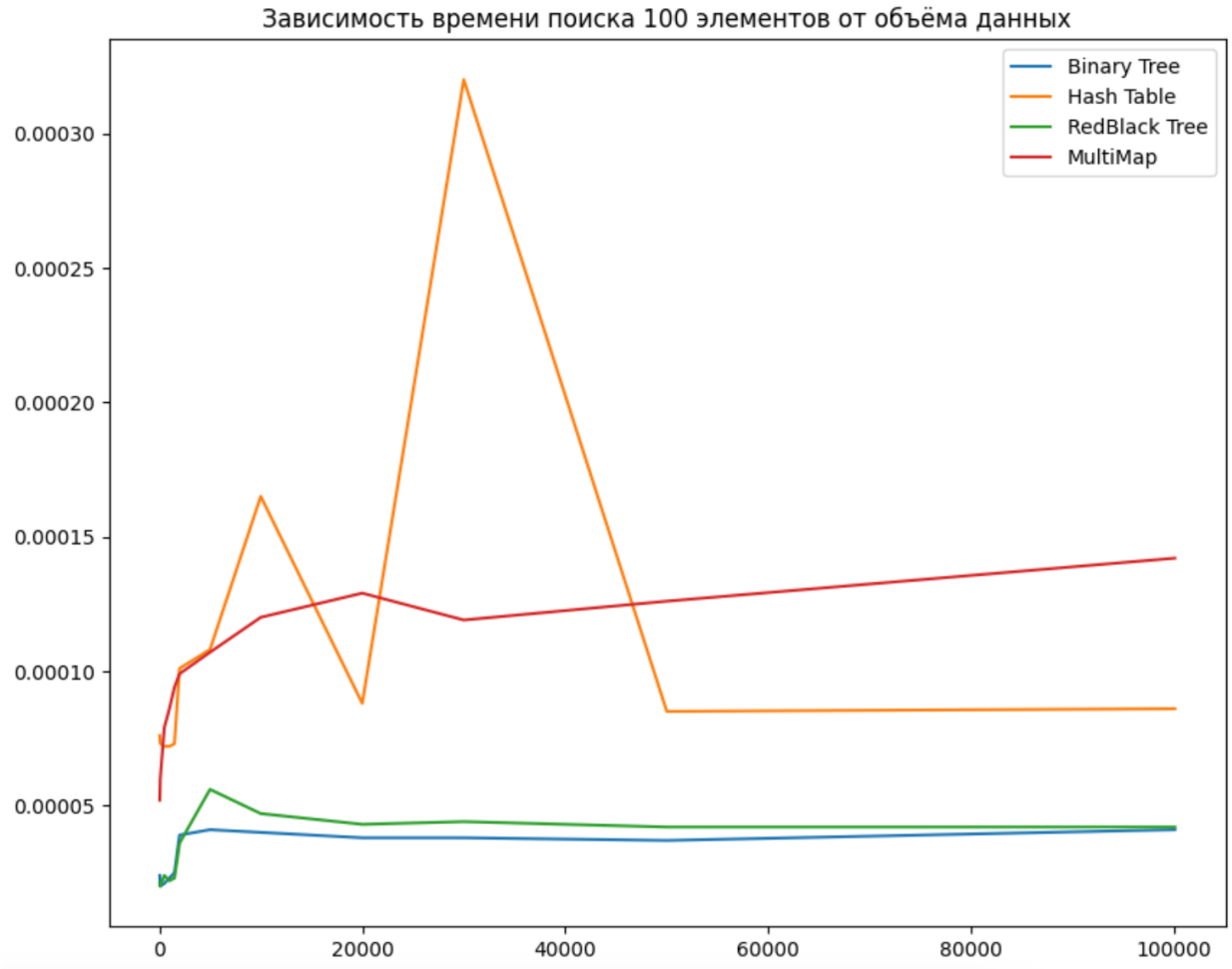
2.1. Сравнение между собой

График 1:



2.2 Сравнение с встроенным multimap

График 2:



3. Документация к коду

Репозиторий с исходными файлами:

Ссылка

Документация:

Алфавитный указатель классов

Классы

Классы с их кратким описанием.

Список файлов

Файлы

Полный список файлов.

Классы

Шаблон класса BinaryTree< T, U >

Класс Бинарного дерева
`#include <BinaryTree.h>`

Открытые члены

- **BinaryTree** ()
  Стандартный конструктор класса

- **~BinaryTree** ()
  Стандартный деструктор класса

- void **insert_node** (Node *&node, T key, U data)
  Функция вставки очередного узла в дерево

- U * **find_node** (Node *node, T key)
  Функция поиска элемента по ключу

- void **print_tree** (Node *node, int depth=0)
  Функция для печати дерева на экран

Открытые атрибуты

- Node * **root**

Подробное описание

template<typename T, typename U>

class BinaryTree< T, U >

Класс Бинарного дерева

Класс имеет следуюшие поля:

Параметры шаблона

Node	- узел дерева, содержащий в себе ключ-значение, количество значений, которые находятся под одним ключом и указетли на левого и правого потомка
root	- вершина дерева

Конструктор(ы)

template<typename T , typename U > BinaryTree< T, U >::BinaryTree () `[inline]`

Стандартный конструктор класса

Инициализирует поле root, как указатель на NULL

template<typename T , typename U > BinaryTree< T, U >::~BinaryTree ()`[inline]`

Стандартный деструктор класса

Вызывает приватную функуию clear()

Методы

template<typename T , typename U > U * BinaryTree< T, U >::find_node (Node * *node*, T *key*)

Функция поиска элемента по ключу

Аргументы

node	- узел, от которого нужно искать элемент
key	- ключ элемента

Возвращает

Массив элементов, лежащих под нужным ключом

template<typename T , typename U > void BinaryTree< T, U >::insert_node (Node *& *node*, T *key*, U *data*)

Функция вставки очередного узла в дерево

Аргументы

node	- узел от которого нужно искать место для вставки
key	- ключ элемента
data	- значение элемента

template<typename T , typename U > void BinaryTree< T, U >::print_tree (Node * *node*, int *depth* = `0`)

Функция для печати дерева на экран

Аргументы

node	- узел, с которого нужно печатать
depth	- (необязательный) - глубина печати дерева

Данные класса

template<typename T , typename U > Node* BinaryTree< T, U >::root

---

Объявления и описания членов класса находятся в файле:

BinaryTree.h

Класс Goods

Класс продуктов, поставляемых на экспорт
`#include <exportGoods.h>`

Открытые члены

- **Goods** ()
  Стандартный конструктор без параметров

- **Goods** (const **Goods** &good)=default
  Стандартный конструктор копирования

- **Goods** (std::string name, std::string country, int quant, int price)
  Стандартный конструктор с параметрами

- **~Goods** ()=default
  Стандартный деструктор

- std::string **getName** () const
  Get-тер для значения поля prod_name.

- std::string **getCountry** () const
  Get-тер для значения поля export_country.

- int **getQuant** () const
  Get-тер для значения поля quant.

- int **getPrice** () const
  Get-тер для значения поля price.

- bool **operator<** (const **Goods** &good) const
  Оператор сравнения <.

- bool **operator>** (const **Goods** &good) const
  Оператор сравнения >

- bool **operator>=** (const **Goods** &good) const
  Оператор сравнения >=.

- bool **operator<=** (const **Goods** &good) const
  Оператор сравнения <=.

- **Goods** & **operator=** (const **Goods** &good)=default
  Стандартный оператор =.

Подробное описание

Класс продуктов, поставляемых на экспорт

Класс, объекты которого будут подвергнуты различным сортировкам в соответствии с заданием. Класс имеет следуюшие поля:

Параметры шаблона

prod_name	- Название товара
export_country	- Страна, в которую экспортируются товары
quant	- Количество товаров
price	- Цена за партию товара

Конструктор(ы)

Goods::Goods ()

Стандартный конструктор без параметров

Проставляет всем полям класса "нулевые" значения

Goods::Goods (const Goods & *good*) `[default]`

Стандартный конструктор копирования

Все поля good копируются в новый объект класса Good

Аргументы

good	- объект подлежащий копированию

Goods::Goods (std::string *name*, std::string *country*, int *quant*, int *price*)

Стандартный конструктор с параметрами

Аргументы

name	- имя товара
country	- страна экспорта
quant	- количество товара
price	- цена товара

Goods::~Goods () `[default]`

Стандартный деструктор

Методы

std::string Goods::getCountry () const `[inline]`

Get-тер для значения поля export_country.

Возвращает

Значение поля export_country - страну экспорту

std::string Goods::getName () const [inline]

Get-тер для значения поля prod_name.

Возвращает

Значения поля prod_name - название товара

int Goods::getPrice () const [inline]

Get-тер для значения поля price.

Возвращает

Значения поля price - цену товара

int Goods::getQuant () const [inline]

Get-тер для значения поля quant.

Возвращает

Значения поля quant - количество товара

bool Goods::operator< (const Goods & *good*) const

Оператор сравнения <.

Сравнивает по полям: название товара, его количество, страна экспорта Сравнение происходит по убыванию приоритета

Аргументы

good	- правый операнд

Возвращает

true либо false в зависимости от результата сравнения

bool Goods::operator<= (const Goods & *good*) const

Оператор сравнения <=.

Сравнивает по полям: название товара, его количество, страна экспорта Сравнение происходит по убыванию приоритета

Аргументы

good	- правый операнд

Возвращает

true либо false в зависимости от результата сравнения

Goods & Goods::operator= (const Goods & *good*) `[default]`

Стандартный оператор =.

Позволяет присваивать объекта класса друг к другу (копировать)

bool Goods::operator> (const Goods & *good*) const

Оператор сравнения >

Сравнивает по полям: название товара, его количество, страна экспорта Сравнение происходит по убыванию приоритета

Аргументы

good	- правый операнд

Возвращает

true либо false в зависимости от результата сравнения

bool Goods::operator>= (const Goods & *good*) const

Оператор сравнения >=.

Сравнивает по полям: название товара, его количество, страна экспорта Сравнение происходит по убыванию приоритета

Аргументы

good	- правый операнд

Возвращает

true либо false в зависимости от результата сравнения

Объявления и описания членов классов находятся в файлах:

exportGoods.hexportGoods.cpp

Шаблон класса HashTable< T, U >

Класс хеш-таблицы
`#include <HashTable.h>`

Открытые члены

- **HashTable** (int size)
  Стандартный конструктор класса

- **~HashTable** ()
  Стандартный деструктор класса

- void **insert** (T key, U data)
  Функция вставки элемента по паре ключ-значение

- U * **find** (T key)
  Функция поиска элемента по ключу

Подробное описание

template<typename T, typename U>

class HashTable< T, U >

Класс хеш-таблицы

Класс имеет следуюшие поля:

Параметры шаблона

table	- таблица: массив связных списков, несущих в себе пару ключ-значение
count	- количество занятых ячеек
size	- общий объем таблицы

Конструктор(ы)

template<typename T , typename U > HashTable< T, U >::HashTable (int *size*) `[explicit]`

Стандартный конструктор класса

Конструктор динамически выделяет память необходимого размера

Аргументы

size	- указывает на необходимый размер таблицы

template<typename T , typename U > HashTable< T, U >::~HashTable ()

Стандартный деструктор класса

Очищает всю память, выделяемую для таблицы

Методы

template<typename T , typename U > U * HashTable< T, U >::find (T *key*)

Функция поиска элемента по ключу

Аргументы

key	- ключ, который необходимо найти

Возвращает

Массив найденных значений, принадлежащих одному ключу

template<typename T , typename U > void HashTable< T, U >::insert (T *key*, U *data*)

Функция вставки элемента по паре ключ-значение

Аргументы

key	- ключ элемента
data	- значение элемента

Объявления и описания членов класса находятся в файле:

HashTable.h

Шаблон класса RedBlackTree< T, U >

Класс Красно-черного дерева
`#include <RedBlackTree.h>`

Открытые члены

- **RedBlackTree** ()
  Стандартный конструктор класса

- **~RedBlackTree** ()
  Стандартный деструктор класса

- void **insert_node** (T key, U data)
  Функция вставки очередного узла в дерево

- U * **find_node** (Node *node, T key)
  Функция поиска элемента по ключу

- void **print_tree** (Node *node, int depth=0)
  Функция для печати дерева на экран

Открытые атрибуты

- Node * **root**
- Node * **nil**

Подробное описание

template<typename T, typename U>

class RedBlackTree< T, U >

Класс Красно-черного дерева

Правила, по которым работает дерево: 1) У листа всегда есть потомок 'NULL' узел 2) Корень и null листья всегда черные 3) У каждого красного узла оба потомка черные 4) Одинаковая черная высота - количество черных узлов от корня до любого 'NULL' 5) Каждый новый узел красный, после вставки - балансируем

Правила, по которым балансируется дерево после каждой вставки: Крайние случаи 0 случай: родитель черный => балансировка не нужна 1 случай: родителя нет (т.е. узел – корень) => просто перекрашиваем в черный

Если дядя красный 2 случай: дед не корень => перекрашиваем родителя и дядю в черный, а деда - в красный 3 случай: дед корень => перекрашиваем родителя и дядю в черный (корень не считается в высоте, все осталось ок, поэтому выполняем 2 случай)

Если дядя черный 4 случай: "зиг-заг" (дед и отец не на одной линии) => левый поворот относительно отца 5 случай: дед и отец на одной линии => отца красим в черный, деда - в красный, правый поворот относительно деда

Класс имеет следуюшие поля:

Параметры шаблона

Node	- узел дерева, содержащий в себе цвет, ключ-значение, количество значений, которые находятся под одним ключом и указетли на родителя, левого и правого потомка
root	- вершина дерева
nil	- нуль-элемент, который вешается на крайние листья

Конструктор(ы)

template<typename T , typename U > RedBlackTree< T, U >::RedBlackTree () `[inline]`

Стандартный конструктор класса

Инициализирует специальную NULL-переменную (NULL-лист) Переменная root инициализируется также

template<typename T , typename U > RedBlackTree< T, U >::~RedBlackTree () `[inline]`

Стандартный деструктор класса

Вызывает приватный методы clear() и очищает память, выделенную под nil

---

Методы

template<typename T , typename U > U * RedBlackTree< T, U >::find_node (Node * *node*, T *key*)

Функция поиска элемента по ключу

Аргументы

node	- узел, от которого необходимо искать элемент
key	- ключ элемента

Возвращает

Массив элементов, лежащих под определённым ключом

template<typename T , typename U > void RedBlackTree< T, U >::insert_node (T *key*, U *data*)

Функция вставки очередного узла в дерево

Аргументы

key	- ключ элемента
data	- значение элемента

template<typename T , typename U > void RedBlackTree< T, U >::print_tree (Node * *node*, int *depth* = `0`)

Функция для печати дерева на экран

Аргументы

node	- узел, с которого нужно печатать
depth	- (необязательный) - глубина печати дерева

Данные класса

template<typename T , typename U > Node* RedBlackTree< T, U >::nil

template<typename T , typename U > Node* RedBlackTree< T, U >::root

---

Объявления и описания членов класса находятся в файле:
RedBlackTree.h

Файлы

Файл BinaryTree.h

```
#include <iostream>
```

Классы

class **BinaryTree< T, U >***Класс Бинарного дерева*

BinaryTree.h

См. документацию.

```
1 #ifndef BINARYTREE_H
2 #define BINARYTREE_H
3 #include <iostream>
4
13 template <typename T, typename U>
14 class BinaryTree {
15 private:
16     struct Node {
17         T key;
18         U data[10000]; //10 for nodes with equal keys
19         int count;
20         Node* left;
21         Node* right;
22
23         Node(T key, U data) : key(key), count(0), left(nullptr), right(nullptr) {
24             this->data[count++] = data;
25         }
26     };
27
32     void clear() { clear_tree(root); }
33
38     void clear_tree(Node* node);
39
40 public:
41     Node* root;
42
47     BinaryTree() : root(nullptr) {
48     }
49
54     ~BinaryTree() { clear(); }
55
62     void insert_node(Node*& node, T key, U data);
63
70     U* find_node(Node* node, T key);
71
77     void print_tree(Node* node, int depth = 0);
78 };
79
80 template <typename T, typename U>
81 void BinaryTree<T, U>::insert_node(Node*& node, T key, U data) {
82     if (node == nullptr) {
83         node = new Node(key, data);
84         if (root == nullptr) root = node;
85     } else if (node->key == key)
86         node->data[node->count++] = data;
87     else if (key < node->key)
88         insert_node(node->left, key, data);
89     else
90         insert_node(node->right, key, data);
91 }
92
93 template <typename T, typename U>
94 U* BinaryTree<T, U>::find_node(Node* node, T key) {
95     U* result = nullptr;
96     while (node != nullptr) {
97         if (node->key == key) {
98             result = node->data;
99             break;
100         } else if (key < node->key)
101             node = node->left;
102         else
103             node = node->right;
104     }
105     return result;
106 }
107
108 template <typename T, typename U>
109 void BinaryTree<T, U>::print_tree(Node* node, int depth) {
110     if (node != nullptr) {
111         print_tree(node->right, depth + 1);
112         for (int i = 0; i < depth; ++i)
113             std::cout << "    "; // Indent based on depth
```

```
114         std::cout << node->key << std::endl;
115         print_tree(node->left, depth + 1);
116     }
117 }
118
119 template <typename T, typename U>
120 void BinaryTree<T, U>::clear_tree(Node* node) {
121     if (node != nullptr) {
122         clear_tree(node->left);
123         clear_tree(node->right);
124         delete node;
125     }
126     root = nullptr;
127 }
128
129
130 #endif //BINARYTREE_H
```

Файл exportGoods.cpp

```
#include "exportGoods.h"
```

Функции

- std::ostream & **operator<<** (std::ostream &out, const **Goods** &good)
  Перегруженный оператор вставки потока <<.

Функции

std::ostream & operator<< (std::ostream & *out*, const Goods & *good*)

Перегруженный оператор вставки потока <<.

Позволяет эффективно выводить все поля класса ввиде форматированной строки

Файл exportGoods.h

```
#include <iomanip>
#include <iostream>
#include <string>
```

Классы

class **Goods***Класс продуктов, поставляемых на экспорт*

Функции

- std::ostream & **operator<<** (std::ostream &out, const **Goods** &good)
  Перегруженный оператор вставки потока <<.

Функции

std::ostream & operator<< (std::ostream & *out*, const Goods & *good*)

Перегруженный оператор вставки потока <<.

Позволяет эффективно выводить все поля класса ввиде форматированной строки

exportGoods.h

См. документацию.

```
1 #ifndef EXPORTGOODS_H
2 #define EXPORTGOODS_H
3 #include <iomanip>
4 #include <iostream>
5 #include <string>
6
18 class Goods {
19    private:
20     std::string prod_name;
21     std::string export_country;
22     int quant, price;
23
24    public:
29     Goods();
30
36     Goods(const Goods& good) = default;
37
44     Goods(std::string name, std::string country, int quant, int price);
45
47     ~Goods() = default;
48
53     std::string getName() const { return prod_name; }
54
58     std::string getCountry() const { return export_country; }
59
64     int getQuant() const { return quant; }
65
70     int getPrice() const { return price; }
71
79     bool operator<(const Goods& good) const;
80
88     bool operator>(const Goods& good) const;
89
97     bool operator>=(const Goods& good) const;
98
106     bool operator<=(const Goods& good) const;
107
112     Goods& operator=(const Goods& good) = default;
113 };
114
119 std::ostream& operator<<(std::ostream& out, const Goods& good);
120
121 #endif  // EXPORTGOODS_H
```

Файл HashTable.h

```
#include <list>
```

Классы

class **HashTable< T, U >***Класс хеш-таблицы*

HashTable.h

См. документацию.

```
1 #ifndef HASHTABLE_H
2 #define HASHTABLE_H
3 #include <list>
4
14 template <typename T, typename U>
15 class HashTable {
16 private:
17     std::list<std::pair<T, U>>* table;
18     int count;
19     int size;
20
25     size_t hash(T key);
26
27 public:
34     explicit HashTable(int size);
35
40     ~HashTable();
41
47     void insert(T key, U data);
48
54     U* find(T key);
55 };
56
57 template <typename T, typename U>
58 size_t HashTable<T, U>::hash(T key) {
59     size_t hashsum = 0;
60     for (int i = 0; i < key.size(); i++) {
61         hashsum += (key[i] * static_cast<int>(pow(31, i))) % size;
62     }
63     return hashsum % size;
64 }
65
66 template <typename T, typename U>
67 HashTable<T, U>::HashTable(int size): count(0), size(size) {
68     table = new std::list<std::pair<T, U>>[size];
69 }
70
71 template <typename T, typename U>
72 HashTable<T, U>::~HashTable() {
73     delete[] table;
74 }
75
76 template <typename T, typename U>
77 void HashTable<T, U>::insert(T key, U data) {
78     if (count == size)
79         std::cerr << "Table is full" << std::endl;
80     else {
81         size_t i = hash(key);
82         bool ifempty = table[i].size() == 0;
83         if (ifempty || table[i].front().first != key) {
84             table[i].push_back(make_pair(key, data));
85             if (ifempty)
86                 count++;
87         }
88     }
89 }
90
91 template <typename T, typename U>
92 U* HashTable<T, U>::find(T key) {
93     U* result = nullptr;
94     size_t i = hash(key);
95     for (auto item : table[i])
96         if (item.first == key)
97             result = &item.second;
98     return result;
99 }
100
101 #endif //HASHTABLE_H
```

Файл main.cpp

```
#include "BinaryTree.h"
#include "HashTable.h"
#include "RedBlackTree.h"
#include "exportGoods.h"
#include <ctime>
#include <fstream>
#include <map>
```

Макросы

- #define **N** 100055
  Устанавливает количество строк, которые будут прочитаны из файла

- #define **cycle** 12
  Устанавливает, сколько измерений необходимо прозвести (число от 0 до 9)

- #define **FIND** 100
  Устанавливает, сколько поисков необходимо прозвести

Функции

- int **main** ()
  Основная функция в программе

Макросы

#define cycle 12

Устанавливает, сколько измерений необходимо прозвести (число от 0 до 9)

#define FIND 100

Устанавливает, сколько поисков необходимо прозвести

#define N 100055

Устанавливает количество строк, которые будут прочитаны из файла

Функции

int main ()

Основная функция в программе

Возвращает

ноль, если программа завершилась успешно

В начале программы создаётся вектор data, в который считается весь файл.

```
std::vector<Goods> data;
```

Стоит обратить внимание, что данные помещаются в вектор с помощью emplace_back, дабы избежать излишнего копирования

```
data.emplace_back(name, country, quant, price);
```

Для проведения измерений задаётся статически массив, в котором записано количество элементов, которые будут учавствовать в сортировке на каждом конкретном замере

```
int sizes[9] = {100, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100050};
```

До начала замеров, открывается файл datagraph.txt на запись, в который запишутся данные для дальнейшего построения графиков некоторыми средствами визуализации данных

```
std::ofstream graph("datagraph.txt");
```

Файл RedBlackTree.h

```
#include <iostream>
```

Классы

class **RedBlackTree< T, U >***Класс Красно-черного дерева*

RedBlackTree.h

См. документацию.

```
1 #ifndef REDBLACKTREE_H
2 #define REDBLACKTREE_H
3 #include <iostream>
4
34 template <typename T, typename U>
35 class RedBlackTree {
36 private:
37     struct Node {
38         T key;
39         U data[10000]; //10 for nodes with equal keys
40         int count;
41         char color;
42         Node* parent;
43         Node* left;
44         Node* right;
45
46         explicit Node(char color): count(0), parent(nullptr), left(nullptr),
right(nullptr) {
47             this->color = 'b';
48         }
49
50         Node(char color, T key, U data, Node* p, Node* left,
51              Node* right) : key(key), count(0),
52                             parent(p), left(left),
53                             right(right) {
54             this->color = color;
55             this->data[count++] = data;
56         }
57     };
58
63     void fix_insert(Node* node);
64
69     void left_rotate(Node* node);
70
75     void right_rotate(Node* node);
76
81     void clear() { clear_tree(root); }
82
87     void clear_tree(Node* node);
88
89 public:
90     Node* root;
91     Node* nil;
92
98     RedBlackTree() {
99         nil = new Node('b');
100         root = nil;
101     }
102
107     ~RedBlackTree() {
108         clear();
109         delete nil;
110     }
111
117     void insert_node(T key, U data);
118
125     U* find_node(Node* node, T key);
126
132     void print_tree(Node* node, int depth = 0);
133 };
134
135 template <typename T, typename U>
136 void RedBlackTree<T, U>::insert_node(T key, U data) {
137     Node* current = root;
138     Node* parent = nullptr;
139     while (current != nil && current->key != key) {
140         parent = current;
141         if (key < current->key)
142             current = current->left;
143         else
144             current = current->right;
145     }
```

```
146
147     if (current->key == key)
148         current->data[current->count++] = data;
149     else {
150         Node* new_node = new Node('r', key, data, parent, nil, nil);
151         if (parent == nullptr)
152             root = new_node;
153         else if (key < parent->key)
154             parent->left = new_node;
155         else
156             parent->right = new_node;
157         fix_insert(new_node);
158     }
159 }
160
161 template <typename T, typename U>
162 void RedBlackTree<T, U>::left_rotate(Node* node) {
163     Node* right = node->right;
164     node->right = right->left;
165     if (right->left != nil)
166         right->left->parent = node;
167     right->parent = node->parent;
168     if (node->parent == nullptr)
169         root = right;
170     else if (node == node->parent->left)
171         node->parent->left = right;
172     else
173         node->parent->right = right;
174     right->left = node;
175     node->parent = right;
176 }
177
178 template <typename T, typename U>
179 void RedBlackTree<T, U>::right_rotate(Node* node) {
180     Node* left = node->left;
181     node->left = left->right;
182     if (left->right != nil)
183         left->right->parent = node;
184     left->parent = node->parent;
185     if (node->parent == nullptr)
186         root = left;
187     else if (node == node->parent->right)
188         node->parent->right = left;
189     else
190         node->parent->left = left;
191     left->right = node;
192     node->parent = left;
193 }
194
195 template <typename T, typename U>
196 void RedBlackTree<T, U>::fix_insert(Node* node) {
197     while (node->parent != nullptr && node->parent->color == 'r') {
198         if (node->parent == node->parent->parent->left) {
199             Node* uncle = node->parent->parent->right;
200             if (uncle->color == 'r') {
201                 uncle->color = 'b';
202                 node->parent->color = 'b';
203                 node->parent->parent->color = 'r';
204                 node = node->parent->parent;
205             } else {
206                 if (node == node->parent->right) {
207                     node = node->parent;
208                     left_rotate(node);
209                 }
210                 node->parent->color = 'b';
211                 node->parent->parent->color = 'r';
212                 right_rotate(node->parent->parent);
213             }
214         } else {
215             Node* uncle = node->parent->parent->left;
216             if (uncle->color == 'r') {
217                 uncle->color = 'b';
218                 node->parent->color = 'b';
219                 node->parent->parent->color = 'r';
220                 node = node->parent->parent;
221             } else {
222                 if (node == node->parent->left) {
```

```
223                     node = node->parent;
224                     right_rotate(node);
225                 }
226                 node->parent->color = 'b';
227                 node->parent->parent->color = 'r';
228                 left_rotate(node->parent->parent);
229             }
230         }
231     }
232     root->color = 'b';
233 }
234
235 template <typename T, typename U>
236 U* RedBlackTree<T, U>::find_node(Node* node, T key) {
237     U* result = nullptr;
238     while (node != nil) {
239         if (node->key == key) {
240             result = node->data;
241             break;
242         } else if (key < node->key)
243             node = node->left;
244         else
245             node = node->right;
246     }
247     return result;
248 }
249
250 template <typename T, typename U>
251 void RedBlackTree<T, U>::print_tree(Node* node, int depth) {
252     if (node != nullptr) {
253         print_tree(node->right, depth + 1);
254         for (int i = 0; i < depth; ++i)
255             std::cout << "    "; // Indent based on depth
256         std::cout << node->key << std::endl;
257         print_tree(node->left, depth + 1);
258     }
259 }
260
261 template <typename T, typename U>
262 void RedBlackTree<T, U>::clear_tree(Node* node) {
263     if (node != nil) {
264         clear_tree(node->left);
265         clear_tree(node->right);
266         delete node;
267     }
268     root = nullptr;
269 }
270
271
272 #endif //REDBLACKTRE
```