

Отчёт

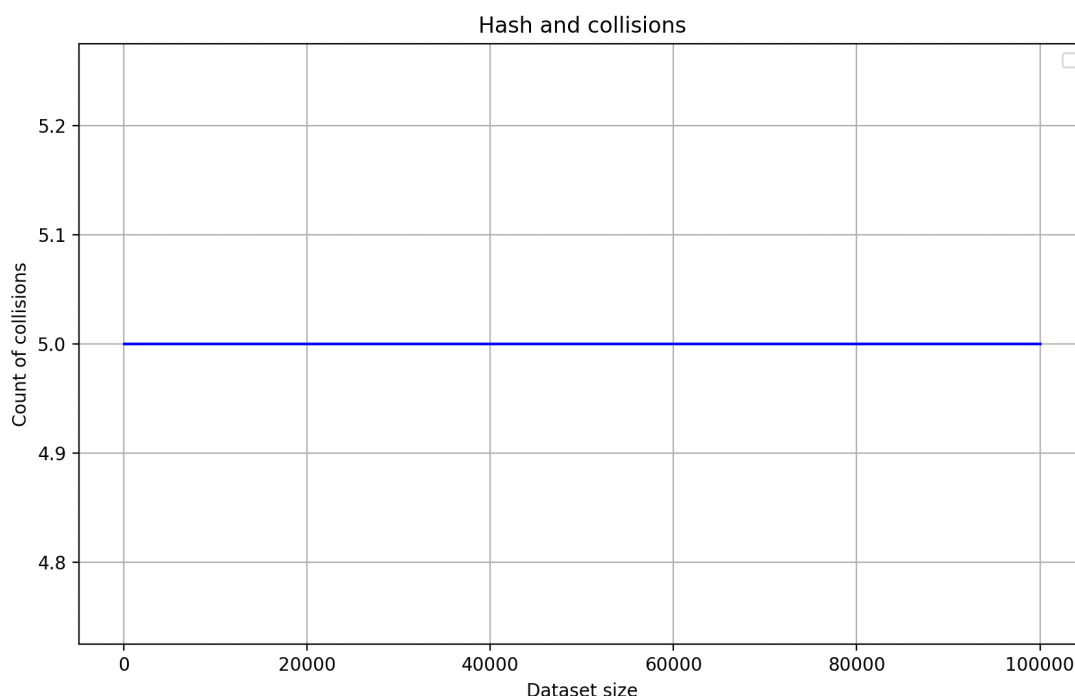
по теме «Алгоритмы поиска данных»

Гриднева Е. В., СКБ222

<https://github.com/emokater/data-search-algorithms/tree/main>

Анализ графиков.

2) График зависимости количества коллизий от размерности массива:



У меня каждый раз одинаковое количество коллизий, от размерности массива оно не зависит.

Если на маленькой выборке некоторые два разных имени всё же сработали в один и тот же индекс (коллизия), то при увеличении размера выборки эти же пары имён будут продолжать падать в тот же слот, потому что хэш каждого имени не меняется. Поэтому график $y = 5$.

4) Время поиска в ассоциативном массиве `multimap<key, object>` в сравнении с остальными способами поиска:

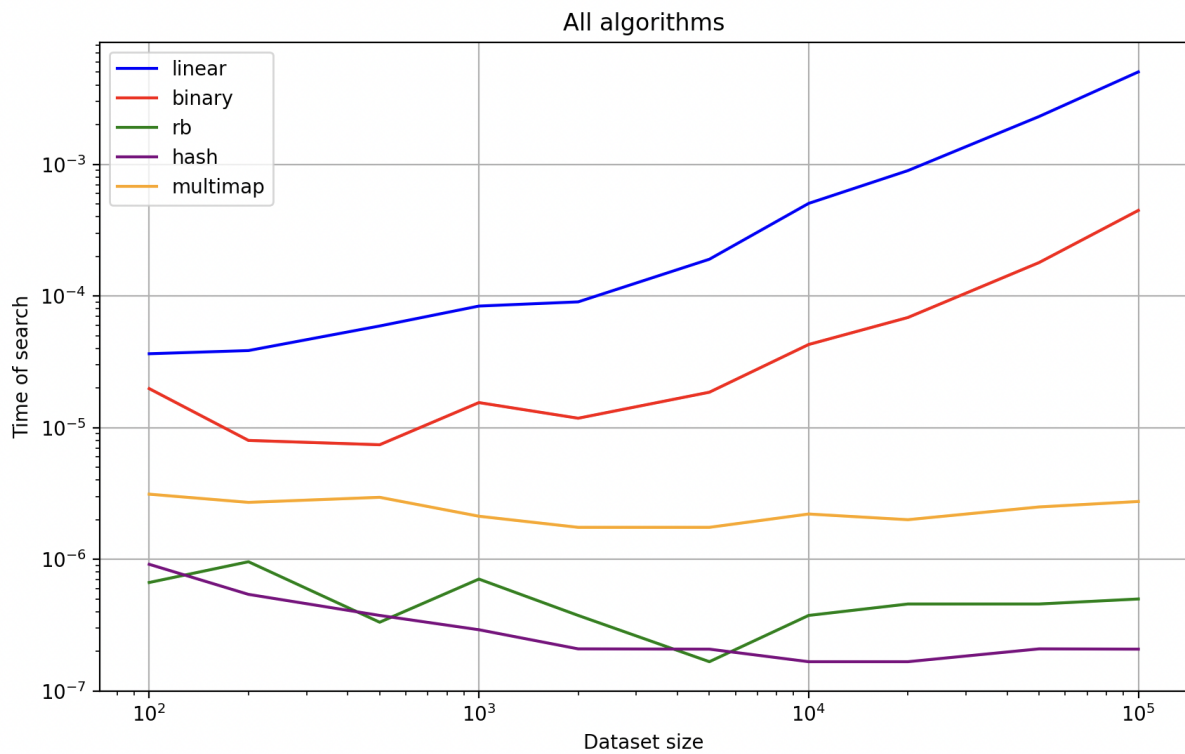
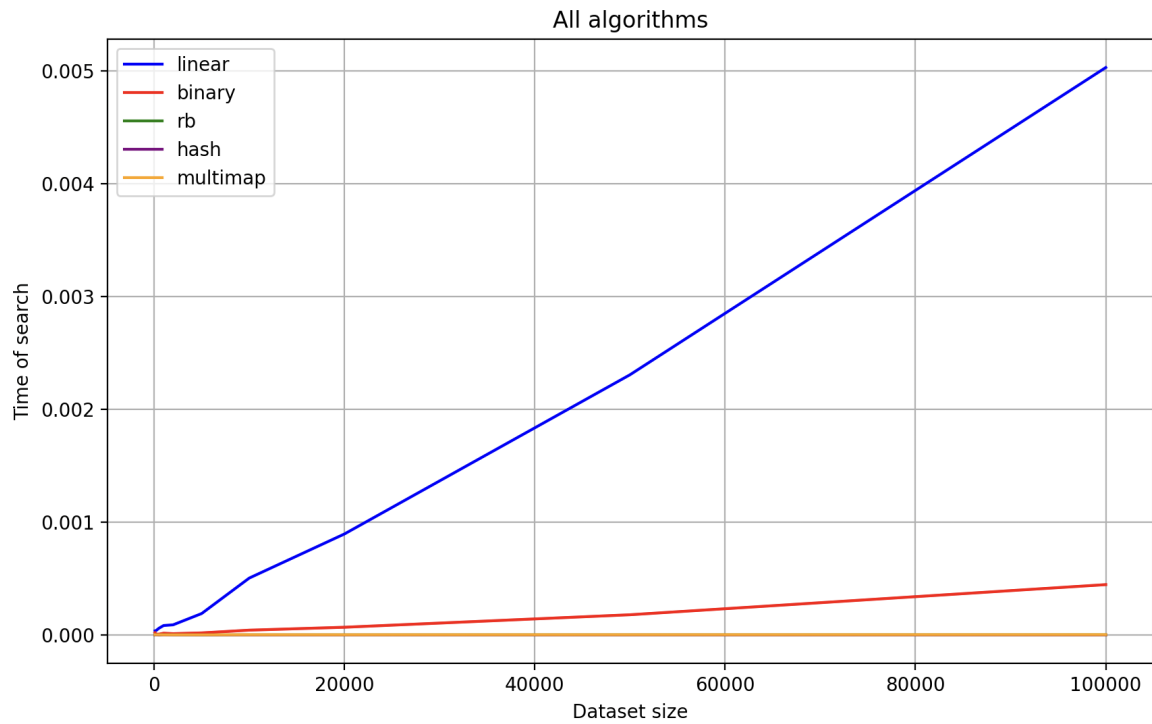
Наиболее эффективен и самый быстрый — поиск по собственной хеш-таблице ($O(1)$).

Почти не уступает ему по скорости **RB-дерево** ($O(\log n)$).

Далее идёт `std::multimap` ($O(\log(n))$).

Потом — небалансированное бинарное дерево (BST), где из-за неравномерного роста глубина может расти с размером данных (худший случай - $O(N)$, средний $O(\log^2(n))$).

Замыкает список **линейный поиск**, чья скорость падает обратно пропорционально N ($O(N)$).



Laboratory work №2

Generated by Doxygen 1.13.2

1 Topic Index	1
1.1 Topics	1
2 Class Index	3
2.1 Class List	3
3 File Index	5
3.1 File List	5
4 Topic Documentation	7
4.1 Constructors and destructor	7
4.1.1 Detailed Description	7
4.1.2 Function Documentation	7
4.1.2.1 RBTREE()	7
4.2 Insert, search and visualization of the tree	7
4.2.1 Detailed Description	8
4.2.2 Function Documentation	8
4.2.2.1 Insert() [1/2]	8
4.2.2.2 Insert() [2/2]	8
4.2.2.3 Search()	8
4.2.2.4 SearchAll() [1/2]	8
4.2.2.5 SearchAll() [2/2]	9
4.3 Supporting methods for basic methods	9
4.3.1 Detailed Description	9
5 Class Documentation	11
5.1 Flower Class Reference	11
5.1.1 Member Function Documentation	12
5.1.1.1 EqFlowers()	12
5.2 HashTable Class Reference	12
5.2.1 Detailed Description	13
5.2.2 Constructor & Destructor Documentation	13
5.2.2.1 HashTable()	13
5.2.3 Member Function Documentation	13
5.2.3.1 Search()	13
5.3 Item Class Reference	14
5.3.1 Detailed Description	14
5.4 Node< T > Class Template Reference	14
5.4.1 Detailed Description	14
5.5 RBNODE< T > Class Template Reference	15
5.5.1 Detailed Description	15
5.5.2 Member Function Documentation	15
5.5.2.1 operator=()	15
5.6 RBTREE< T > Class Template Reference	16

5.6.1 Detailed Description	16
5.7 Tree< T > Class Template Reference	17
5.7.1 Detailed Description	17
6 File Documentation	19
6.1 binary_tree.h File Reference	19
6.2 binary_tree.h	19
6.3 flower.h File Reference	21
6.4 flower.h	21
6.5 hash.h File Reference	21
6.5.1 Detailed Description	22
6.5.2 Function Documentation	22
6.5.2.1 hashFunc_rs()	22
6.6 hash.h	23
6.7 io.h File Reference	25
6.7.1 Function Documentation	25
6.7.1.1 parserCSV()	25
6.7.1.2 saveRes()	26
6.8 io.h	26
6.9 linear.h File Reference	26
6.9.1 Detailed Description	27
6.9.2 Function Documentation	27
6.9.2.1 linearSearch()	27
6.9.2.2 searchAll()	27
6.10 linear.h	28
6.11 rb_tree.h File Reference	28
6.11.1 Detailed Description	29
6.12 rb_tree.h	29
Index	33

Chapter 1

Topic Index

1.1 Topics

Here is a list of all topics with brief descriptions:

Constructors and destructor	7
Insert, search and visualization of the tree	7
Supporting methods for basic methods	9

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Flower	Information about a flower (name, color, scent intensity, and habitat regions)	11
HashTable	Implements a hash table using separate chaining to store Flower objects by their string names	12
Item	Represents an item (node) in the linked list for a hash table slot	14
Node< T >	Represents a node in a binary search tree	14
RBNode< T >	Represents a node in a Red-Black Tree	15
RBTREE< T >	Implements a Red-Black Tree	16
Tree< T >	Implements a simple Binary Search Tree (BST)	17

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

binary_tree.h	Defines a templated Binary Search Tree (BST) with insertion, search, and traversal functions . . .	19
flower.h	Declaration of the Flower class	21
hash.h	Defines a hash table for storing Flower objects using separate chaining for collision resolution .	21
io.h	Interface for input/output functions: parsing CSV files and save result to files	25
linear.h	Contains functions for performing linear search on an array	26
rb_tree.h	Defines a templated Red-Black Tree with insertion, search, and traversal functions	28

Chapter 4

Topic Documentation

4.1 Constructors and destructor

Functions

- **Tree< T >::Tree** (T value)
- **Item::Item** (const string &key, const Flower &value)
- **RBTree< T >::RBTree** (T value)

4.1.1 Detailed Description

4.1.2 Function Documentation

4.1.2.1 RBTree()

```
template<typename T>
RBTree< T >::RBTree (
    T value) [inline]
```

< The root node is always initialized with BLACK color.

4.2 Insert, search and visualization of the tree

Functions

- void **Tree< T >::Insert** (const T &value)
Insert a new value into the BST.
- **Node< T > * Tree< T >::Search** (const T &value) const
Search for the first node containing a given value.
- vector< **Node< T > * Tree< T >::SearchAll** (const T &value)
Search for all nodes containing a given value.
- void **Tree< T >::PrintTree** ()
Print all values in the tree using pre-order traversal.
- void **RBTree< T >::Insert** (const T &value)
Insert a new value into the Red-Black Tree.
- **RBNode< T > * RBTree< T >::SearchAll** (const T &value)
Search for all nodes containing a given value.
- void **RBTree< T >::PrintTree** ()
Print all nodes in the tree using pre-order traversal.

4.2.1 Detailed Description

4.2.2 Function Documentation

4.2.2.1 Insert() [1/2]

```
template<typename T>
void RBTREE< T >::Insert (
    const T & value) [inline]
```

If the tree is empty, the new node becomes the root and is colored BLACK. Otherwise, insert similar to a BST and then rebalance via Balance().

Parameters

<i>value</i>	Reference to the value to insert.
--------------	-----------------------------------

4.2.2.2 Insert() [2/2]

```
template<typename T>
void Tree< T >::Insert (
    const T & value) [inline]
```

If the tree is empty, the new value becomes the root. Otherwise, traverse the tree and insert the new node in the correct position to maintain BST property.

Parameters

<i>value</i>	Reference to the value to insert.
--------------	-----------------------------------

4.2.2.3 Search()

```
template<typename T>
Node< T > * Tree< T >::Search (
    const T & value) const [inline]
```

Parameters

<i>value</i>	Reference to the value to search for.
--------------	---------------------------------------

Returns

Pointer to the node containing the value, or nullptr if not found.

4.2.2.4 SearchAll() [1/2]

```
template<typename T>
RBNODE< T > * RBTREE< T >::SearchAll (
    const T & value) [inline]
```

Parameters

<i>value</i>	Reference to the value to search for.
--------------	---------------------------------------

Returns

A vector of pointers to nodes containing the value. If none found, returns an empty vector.

4.2.2.5 SearchAll() [2/2]

```
template<typename T>
vector< Node< T > * > Tree< T >::SearchAll (
    const T & value) [inline]
```

Parameters

<i>value</i>	Reference to the value to search for.
--------------	---------------------------------------

Returns

A vector of pointers to nodes containing the value. If none found, returns an empty vector.

4.3 Supporting methods for basic methods

4.3.1 Detailed Description

Chapter 5

Class Documentation

5.1 Flower Class Reference

The `Flower` class contains information about a flower (name, color, scent intensity, and habitat regions).

```
#include <flower.h>
```

Public Member Functions

- `bool EqFlowers (const Flower &other) const`
Compares the current object with another based on key fields.

Constructors

Default constructor.

- `Flower (string name, string color, string smell, vector< string > regions)`
- `Flower (const Flower &other)=default`
Copy constructor.
- `Flower (Flower &&other)=default`
Move constructor.
- `~Flower ()=default`
Destructor.

Getters

- `string GetName () const`
- `string GetColor () const`
- `string GetSmell () const`
- `vector< string > GetRegions () const`

Setters

- `void SetName (string name)`
- `void SetColor (string color)`
- `void SetSmell (string smell)`
- `void SetRegions (vector< string > regions)`

Operator Overloading

Comparison based on key fields (name, color, smell).

- `bool operator> (const Flower &other) const`
- `bool operator< (const Flower &other) const`
- `bool operator>= (const Flower &other) const`
- `bool operator<= (const Flower &other) const`
- `bool operator== (const Flower &other) const`
- `Flower & operator= (const Flower &other)`

5.1.1 Member Function Documentation

5.1.1.1 EqFlowers()

```
bool Flower::EqFlowers (
    const Flower & other) const
```

Parameters

<i>other</i>	Reference to another Flower object.
--------------	-------------------------------------

Returns

true if name, color, and smell match; otherwise false.

Two objects are considered "equal" if their name, color, and smell fields are the same. The regions field is ignored in this comparison.

The documentation for this class was generated from the following file:

- [flower.h](#)

5.2 HashTable Class Reference

Implements a hash table using separate chaining to store Flower objects by their string names.

```
#include <hash.h>
```

Public Member Functions

- [HashTable](#) (const vector< [Flower](#) > &data)
Construct a hash table and insert a vector of Flower objects into it.
- [~HashTable](#) ()
Destructor. Frees memory for all Items and their vectors.
- vector< [Flower](#) > * [Search](#) (const string &key) const
Search for all Flower objects associated with a given key.
- long long [GetCount](#) ()
- long long [GetCountUnq](#) ()
- long long [GetCollisions](#) ()
- [Item](#) ** [GetItems](#) ()
- void [GetTable](#) ()
Print the contents of the hash table to stdout.

5.2.1 Detailed Description

The table uses:

- hashFunc_rs to map string keys to hash table slot indices.
- An array of pointers to [Item](#) (linked-list heads) of size SIZE.
- A collision counter to track how many chaining operations occurred.

Note

The [Flower](#) class must provide a method GetName() returning a string key.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 HashTable()

```
HashTable::HashTable (
    const vector< Flower > & data) [inline]
```

For each [Flower](#) in data:

1. Compute its key via GetName(), and hash to an index.
2. If the bucket is empty, create a new [Item](#).
3. If the bucket already has items, traverse the chain:
 - If an existing [Item](#) has the same key, append the [Flower](#) to its vector.
 - Otherwise, add a new [Item](#) at the end of the chain and increment collisions.

Parameters

<i>data</i>	A vector of Flower objects to insert into the hash table.
-------------	---

5.2.3 Member Function Documentation

5.2.3.1 Search()

```
vector< Flower > * HashTable::Search (
    const string & key) const [inline]
```

Parameters

<i>key</i>	The string key to search for.
------------	-------------------------------

Returns

Pointer to a vector of [Flower](#) objects if the key is found; nullptr if the key does not exist in the table.

The documentation for this class was generated from the following file:

- [hash.h](#)

5.3 Item Class Reference

Represents an item (node) in the linked list for a hash table slot.

```
#include <hash.h>
```

Public Member Functions

- **Item** (const string &key, const [Flower](#) &value)

Public Attributes

- string **key_**
The string key for this item.
- vector< [Flower](#) > * **values_**
Pointer to a vector of [Flower](#) objects with this key.
- [Item](#) * **next_**
Pointer to the next [Item](#) in the chain (collision list).

5.3.1 Detailed Description

Each item stores:

- **key_**: The string key that hashes to this slot.
- **values_**: A pointer to a vector of [Flower](#) objects associated with this key.
- **next_**: Pointer to the next item in the same slot's linked list (for chaining).

The documentation for this class was generated from the following file:

- [hash.h](#)

5.4 Node< T > Class Template Reference

Represents a node in a binary search tree.

```
#include <binary_tree.h>
```

Public Member Functions

- **Node** (T value)

Public Attributes

- T **value_**
The value stored in the node.
- [Node](#) * **left_** = nullptr
Pointer to the left child node.
- [Node](#) * **right_** = nullptr
Pointer to the right child node.

5.4.1 Detailed Description

```
template<typename T>  
class Node< T >
```

Template Parameters

<i>T</i>	Type of the value stored in the node.
----------	---------------------------------------

The documentation for this class was generated from the following file:

- [binary_tree.h](#)

5.5 RBNode< T > Class Template Reference

Represents a node in a Red-Black [Tree](#).

```
#include <rb_tree.h>
```

Public Member Functions

- **RBNode** (const *T* value)
- [RBNode](#) & [operator=](#) (const [RBNode](#) &other)
Assignment operator for [RBNode](#).

Public Attributes

- vector< *T* > **values_**
- [Color](#) **color_**
- [RBNode](#) * **left_**
- [RBNode](#) * **right_**
- [RBNode](#) * **parent_**

5.5.1 Detailed Description

```
template<typename T>
class RBNode< T >
```

Each node stores a value of type *T*, its color, and pointers to left, right, and parent nodes.

Template Parameters

<i>T</i>	Type of the value stored in the node.
----------	---------------------------------------

5.5.2 Member Function Documentation

5.5.2.1 operator=()

```
template<typename T>
RBNode & RBNode< T >::operator= (
    const RBNode< T > & other) [inline]
```

Parameters

<i>other</i>	The node to copy from.
--------------	------------------------

Returns

Reference to this node after copying.

Copies value, color, and pointers from another node.

The documentation for this class was generated from the following file:

- [rb_tree.h](#)

5.6 RBTREE< T > Class Template Reference

Implements a Red-Black [Tree](#).

```
#include <rb_tree.h>
```

Public Member Functions

- [RBTREE](#) (T value)
- void [Insert](#) (const T &value)
Insert a new value into the Red-Black [Tree](#).
- [RBNODE](#)< T > * [SearchAll](#) (const T &value)
Search for all nodes containing a given value.
- void [PrintTree](#) ()
Print all nodes in the tree using pre-order traversal.

5.6.1 Detailed Description

```
template<typename T>
class RBTREE< T >
```

Provides operations to insert values, search for a single value or all occurrences, and print the tree. Maintains Red-Black properties for balancing after insertions.

Template Parameters

<i>T</i>	Type of the values stored in the tree.
----------	--

The documentation for this class was generated from the following file:

- [rb_tree.h](#)

5.7 Tree< T > Class Template Reference

Implements a simple Binary Search [Tree](#) (BST).

```
#include <binary_tree.h>
```

Public Member Functions

- **Tree** (T value)
- void [Insert](#) (const T &value)
Insert a new value into the BST.
- [Node](#)< T > * [Search](#) (const T &value) const
Search for the first node containing a given value.
- vector< [Node](#)< T > * > [SearchAll](#) (const T &value)
Search for all nodes containing a given value.
- void **PrintTree** ()
Print all values in the tree using pre-order traversal.

5.7.1 Detailed Description

```
template<typename T>  
class Tree< T >
```

Provides operations to insert values, search for a single value or all occurrences, and print the tree in a pre-order traversal.

Template Parameters

<i>T</i>	Type of the values stored in the tree.
----------	--

The documentation for this class was generated from the following file:

- [binary_tree.h](#)

Chapter 6

File Documentation

6.1 binary_tree.h File Reference

Defines a templated Binary Search [Tree](#) (BST) with insertion, search, and traversal functions.

```
#include <vector>
#include <iostream>
```

Classes

- class [Node](#)< T >
Represents a node in a binary search tree.
- class [Tree](#)< T >
Implements a simple Binary Search [Tree](#) (BST).

6.2 binary_tree.h

[Go to the documentation of this file.](#)

```
00001
00003
00004 #ifndef BINARY_TREE_H
00005 #define BINARY_TREE_H
00006
00007 #include <vector>
00008 #include <iostream>
00009 using namespace std;
00010
00016 template <typename T>
00017 class Node {
00018 public:
00019     Node() = default;
00020     Node(T value) { value_ = value; }
00021
00022 public:
00023     T value_;
00024     Node *left_ = nullptr;
00025     Node *right_ = nullptr;
00026 };
00027
00036 template <typename T>
00037 class Tree {
00038 public:
00041
```

```

00042     Tree() { root_ = nullptr; }
00043     Tree(T value) { root_ = new Node<T>(value); }
00044     ~Tree() { DeleteTree(root_); }
00045
00046
00047
00057     void Insert(const T& value) {
00058         if (!root_) {
00059             root_ = new Node<T>(value);
00060             return;
00061         }
00062
00063         Node<T> *cur = root_;
00064         while (true) {
00065             if (value < cur->value_) {
00066                 if (cur->left_ == nullptr) {
00067                     cur->left_ = new Node<T>(value);
00068                     break;
00069                 }
00070                 cur = cur->left_;
00071             } else {
00072                 if (cur->right_ == nullptr) {
00073                     cur->right_ = new Node<T>(value);
00074                     break;
00075                 }
00076                 cur = cur->right_;
00077             }
00078         }
00079     }
00080
00087     Node<T>* Search(const T& value) const { return SupportSearch(root_, value); }
00088
00094     vector<Node<T>*> SearchAll(const T& value) {
00095         vector<Node<T>*> res;
00096         Node<T> *tmp = SupportSearch(root_, value);
00097
00098         while (tmp) {
00099             res.push_back(tmp);
00100             tmp = SupportSearch(tmp->right_, value);
00101         }
00102
00103         return res;
00104     }
00105
00107     void PrintTree() { SupportPrint(root_); }
00108
00109 private:
00110     Node<T> *root_ = nullptr;
00111
00112 private:
00113
00121     void SupportPrint(Node<T> *root) {
00122         if (root) {
00123             cout << root->value_ << endl;
00124             SupportPrint(root->left_);
00125             SupportPrint(root->right_);
00126         }
00127     }
00128
00139     Node<T>* SupportSearch(Node<T>* root, const T& value) {
00140         Node<T> *cur = root;
00141
00142         while (cur) {
00143             if (cur->value_ == value) { return cur; }
00144
00145             if (value < cur->value_) {
00146                 cur = cur->left_;
00147             } else {
00148                 cur = cur->right_;
00149             }
00150         }
00151
00152         return nullptr;
00153     }
00154
00156     void DeleteTree(Node<T> *root) {
00157         if (root) {
00158             DeleteTree(root->left_);
00159             DeleteTree(root->right_);
00160             delete root;
00161         }
00162     }
00163 };
00164
00165
00166 #endif

```

6.3 flower.h File Reference

Declaration of the [Flower](#) class.

```
#include <string>
#include <vector>
```

Classes

- class [Flower](#)

The [Flower](#) class contains information about a flower (name, color, scent intensity, and habitat regions).

6.4 flower.h

[Go to the documentation of this file.](#)

```
00001
00003
00004 #ifndef FLOWER_H
00005 #define FLOWER_H
00006
00007 #include <string>
00008 #include <vector>
00009
00010 using namespace std;
00011
00013 class Flower {
00014 public:
00018     Flower() = default;
00019     Flower(string name, string color, string smell, vector<string> regions);
00021     Flower(const Flower& other) = default;
00023     Flower(Flower&& other) = default;
00025     ~Flower() = default;
00027
00030     string GetName() const { return name_; }
00031     string GetColor() const { return color_; }
00032     string GetSmell() const { return smell_; }
00033     vector<string> GetRegions() const { return regions_; }
00035
00038     void SetName(string name) { name_ = name; }
00039     void SetColor(string color) { color_ = color; }
00040     void SetSmell(string smell) { smell_ = smell; }
00041     void SetRegions(vector<string> regions) { regions_ = regions; }
00043
00047     bool operator>(const Flower& other) const;
00048     bool operator<(const Flower& other) const;
00049     bool operator>=(const Flower& other) const;
00050     bool operator<=(const Flower& other) const;
00051     bool operator==(const Flower& other) const;
00052     Flower& operator=(const Flower& other);
00054
00061     bool EqFlowers(const Flower& other) const;
00062
00063 private:
00064     string name_;
00065     string color_;
00066     string smell_;
00067     vector<string> regions_;
00068 };
00069
00070 #endif
```

6.5 hash.h File Reference

Defines a hash table for storing [Flower](#) objects using separate chaining for collision resolution.

```
#include <string>
#include <vector>
#include <iostream>
#include "flower.h"
```

Classes

- class [Item](#)
Represents an item (node) in the linked list for a hash table slot.
- class [HashTable](#)
Implements a hash table using separate chaining to store [Flower](#) objects by their string names.

Macros

- `#define SIZE 14`
Size of the hash table.

Functions

- unsigned int [hashFunc_rs](#) (string key)
Compute a hash value for a string using the RS (Robert Sedgwicks) algorithm.

6.5.1 Detailed Description

Provides:

- [hashFunc_rs](#): A string-based hash function (RS algorithm).
- [Item](#): A node in the linked list used for collision resolution.
- [HashTable](#): A hash table that stores vectors of [Flower](#) objects under string keys.

6.5.2 Function Documentation

6.5.2.1 [hashFunc_rs\(\)](#)

```
unsigned int hashFunc_rs (  
    string key)
```

The function iterates over each character in the key, updating the hash with a multiplier and accumulating the result. Finally, the value is taken modulo SIZE to obtain a hash-table index.

Parameters

<i>key</i>	The input string to hash.
------------	---------------------------

Returns

An unsigned int in the range [0, SIZE-1], representing the hash-table index.

6.6 hash.h

[Go to the documentation of this file.](#)

```

00001
00002
00003
00004 #ifndef HASH_H
00005 #define HASH_H
00006
00007 #include <string>
00008 #include <vector>
00009 #include <iostream>
00010 #include "flower.h"
00011
00012 #define SIZE 14
00013 using namespace std;
00014
00015 unsigned int hashFunc_rs(string key) {
00016     unsigned int a = 63689;
00017     unsigned int b = 378551;
00018     unsigned int hash = 0;
00019     unsigned int i = 0;
00020
00021     int len = key.length();
00022
00023     for (i = 0; i < len; ++i) {
00024         hash = hash * a + (unsigned char)key[i];
00025         a = a * b;
00026     }
00027
00028     return (hash % SIZE);
00029 }
00030
00031 class Item {
00032 public:
00033     string key_;
00034     vector<Flower> *values_;
00035     Item *next_;
00036
00037 public:
00038     Item() {
00039         key_ = "";
00040         values_ = new vector<Flower>();
00041         next_ = nullptr;
00042     }
00043
00044     Item(const string& key, const Flower& value) {
00045         key_ = key;
00046         values_ = new vector<Flower>();
00047         values_>push_back(value);
00048         next_ = nullptr;
00049     }
00050 };
00051
00052 class HashTable {
00053 public:
00054     HashTable(const vector<Flower>& data) {
00055         NullTable();
00056
00057         count = data.size();
00058
00059         for (size_t i = 0; i < count; ++i) {
00060             string key = data[i].GetName();
00061             Flower value = data[i];
00062             unsigned int hash = hashFunc_rs(key);
00063
00064             if (!items_[hash]) {
00065                 items_[hash] = new Item(key, value);
00066                 unq_count += 1;
00067             } else {
00068                 Item *where = items_[hash];
00069                 int is_collis = 1;
00070
00071                 while (true) {
00072                     if (key == where->key_) {
00073                         where->values_>push_back(value);
00074                         is_collis = 0;
00075                         break;
00076                     } else {
00077                         if (where->next_) {
00078                             where = where->next_;
00079                         } else {
00080                             break;
00081                         }
00082                     }
00083                 }
00084             }
00085         }
00086     }
00087
00088     ~HashTable() {
00089         for (int i = 0; i < items_.size(); ++i) {
00090             delete items_[i];
00091         }
00092     }
00093
00094     void NullTable() {
00095         items_.clear();
00096         unq_count = 0;
00097     }
00098
00099     int GetCount() const {
00100         return unq_count;
00101     }
00102
00103     Item* GetItem(const string& key) const {
00104         unsigned int hash = hashFunc_rs(key);
00105         if (items_[hash]) {
00106             return items_[hash];
00107         }
00108         return nullptr;
00109     }
00110
00111     void Print() const {
00112         for (int i = 0; i < items_.size(); ++i) {
00113             if (items_[i]) {
00114                 cout << "key: " << items_[i]>key_ << " ";
00115                 for (int j = 0; j < items_[i]>values_.size(); ++j) {
00116                     cout << items_[i]>values_[j]>GetName() << " ";
00117                 }
00118                 cout << endl;
00119             }
00120         }
00121     }
00122
00123     void PrintUnqCount() const {
00124         cout << "unq_count: " << unq_count << endl;
00125     }
00126
00127     void PrintCollisCount() const {
00128         cout << "collis_count: " << collis_count << endl;
00129     }
00130
00131     void PrintItems() const {
00132         for (int i = 0; i < items_.size(); ++i) {
00133             if (items_[i]) {
00134                 cout << "Item " << i << ": " << items_[i]>key_ << " ";
00135                 for (int j = 0; j < items_[i]>values_.size(); ++j) {
00136                     cout << items_[i]>values_[j]>GetName() << " ";
00137                 }
00138                 cout << endl;
00139             }
00140         }
00141     }
00142
00143     void PrintUnqItems() const {
00144         for (int i = 0; i < items_.size(); ++i) {
00145             if (items_[i]) {
00146                 cout << "Unq Item " << i << ": " << items_[i]>key_ << " ";
00147                 for (int j = 0; j < items_[i]>values_.size(); ++j) {
00148                     cout << items_[i]>values_[j]>GetName() << " ";
00149                 }
00150                 cout << endl;
00151             }
00152         }
00153     }
00154
00155     void PrintCollisItems() const {
00156         for (int i = 0; i < items_.size(); ++i) {
00157             if (items_[i]) {
00158                 cout << "Collis Item " << i << ": " << items_[i]>key_ << " ";
00159                 for (int j = 0; j < items_[i]>values_.size(); ++j) {
00160                     cout << items_[i]>values_[j]>GetName() << " ";
00161                 }
00162                 cout << endl;
00163             }
00164         }
00165     }
00166
00167     void PrintItemsAndUnqCount() const {
00168         PrintItems();
00169         PrintUnqCount();
00170     }
00171
00172     void PrintItemsAndCollisCount() const {
00173         PrintItems();
00174         PrintCollisCount();
00175     }
00176
00177     void PrintItemsAndUnqCollisCount() const {
00178         PrintItems();
00179         PrintUnqCount();
00180         PrintCollisCount();
00181     }
00182
00183     void PrintUnqItemsAndCollisCount() const {
00184         PrintUnqItems();
00185         PrintCollisCount();
00186     }
00187
00188     void PrintCollisItemsAndUnqCount() const {
00189         PrintCollisItems();
00190         PrintUnqCount();
00191     }
00192
00193     void PrintCollisItemsAndUnqCollisCount() const {
00194         PrintCollisItems();
00195         PrintUnqCount();
00196         PrintCollisCount();
00197     }
00198
00199     void PrintUnqItemsAndUnqCollisCount() const {
00200         PrintUnqItems();
00201         PrintUnqCount();
00202         PrintCollisCount();
00203     }
00204
00205     void PrintCollisItemsAndCollisCount() const {
00206         PrintCollisItems();
00207         PrintCollisCount();
00208     }
00209
00210     void PrintUnqItemsAndUnqItems() const {
00211         PrintUnqItems();
00212         PrintUnqCount();
00213     }
00214
00215     void PrintCollisItemsAndCollisItems() const {
00216         PrintCollisItems();
00217         PrintCollisCount();
00218     }
00219
00220     void PrintUnqItemsAndCollisItems() const {
00221         PrintUnqItems();
00222         PrintUnqCount();
00223         PrintCollisCount();
00224     }
00225
00226     void PrintCollisItemsAndUnqItems() const {
00227         PrintCollisItems();
00228         PrintCollisCount();
00229         PrintUnqCount();
00230     }
00231
00232     void PrintCollisItemsAndCollisItemsAndUnqCount() const {
00233         PrintCollisItems();
00234         PrintCollisCount();
00235         PrintUnqCount();
00236     }
00237
00238     void PrintUnqItemsAndCollisItemsAndUnqCount() const {
00239         PrintUnqItems();
00240         PrintUnqCount();
00241         PrintCollisCount();
00242     }
00243
00244     void PrintCollisItemsAndUnqItemsAndUnqCount() const {
00245         PrintCollisItems();
00246         PrintCollisCount();
00247         PrintUnqCount();
00248     }
00249
00250     void PrintUnqItemsAndCollisItemsAndCollisCount() const {
00251         PrintUnqItems();
00252         PrintUnqCount();
00253         PrintCollisCount();
00254     }
00255
00256     void PrintCollisItemsAndUnqItemsAndCollisCount() const {
00257         PrintCollisItems();
00258         PrintCollisCount();
00259         PrintUnqCount();
00260     }
00261
00262     void PrintUnqItemsAndCollisItemsAndCollisCount() const {
00263         PrintUnqItems();
00264         PrintUnqCount();
00265         PrintCollisCount();
00266     }
00267
00268     void PrintCollisItemsAndUnqItemsAndCollisCount() const {
00269         PrintCollisItems();
00270         PrintCollisCount();
00271         PrintUnqCount();
00272     }
00273
00274     void PrintUnqItemsAndCollisItemsAndUnqCollisCount() const {
00275         PrintUnqItems();
00276         PrintUnqCount();
00277         PrintCollisCount();
00278     }
00279
00280     void PrintCollisItemsAndUnqItemsAndUnqCollisCount() const {
00281         PrintCollisItems();
00282         PrintCollisCount();
00283         PrintUnqCount();
00284     }
00285
00286     void PrintUnqItemsAndCollisItemsAndUnqCollisCount() const {
00287         PrintUnqItems();
00288         PrintUnqCount();
00289         PrintCollisCount();
00290     }
00291
00292     void PrintCollisItemsAndUnqItemsAndUnqCollisCount() const {
00293         PrintCollisItems();
00294         PrintCollisCount();
00295         PrintUnqCount();
00296     }
00297
00298     void PrintUnqItemsAndCollisItemsAndCollisCountAndUnqCount() const {
00299         PrintUnqItems();
00300         PrintUnqCount();
00301         PrintCollisCount();
00302     }
00303
00304     void PrintCollisItemsAndUnqItemsAndCollisCountAndUnqCount() const {
00305         PrintCollisItems();
00306         PrintCollisCount();
00307         PrintUnqCount();
00308     }
00309
00310     void PrintUnqItemsAndCollisItemsAndCollisCountAndUnqCount() const {
00311         PrintUnqItems();
00312         PrintUnqCount();
00313         PrintCollisCount();
00314     }
00315
00316     void PrintCollisItemsAndUnqItemsAndCollisCountAndUnqCount() const {
00317         PrintCollisItems();
00318         Print
```

```

00134
00135         if (is_collis) {
00136             collisions += 1;
00137             unq_count += 1;
00138             where->next_ = new Item(key, value);
00139         }
00140     }
00141 }
00142 }
00143
00144 ~HashTable() {
00145     for (int i = 0; i < SIZE; ++i) {
00146         SupportDelete(items_[i]);
00147     }
00148 }
00149
00150 vector<Flower>* Search(const string& key) const {
00151     vector<Flower> *res = nullptr;
00152
00153     unsigned int hash = hashFunc_rs(key);
00154     Item *cur = items_[hash];
00155     while (cur) {
00156         if (cur->key_ == key) {
00157             res = cur->values_;
00158             break;
00159         }
00160         cur = cur->next_;
00161     }
00162
00163     return res;
00164 }
00165
00166 long long GetCount() { return count; }
00167 long long GetCountUnq() { return unq_count; }
00168 long long GetCollisions() { return collisions; }
00169 Item** GetItems() { return items_; }
00170
00171 void GetTable() {
00172     for (int i = 0; i < SIZE; ++i) {
00173         Item *cur = items_[i];
00174
00175         cout << i << " \t";
00176         if (!cur) {
00177             cout << "-" << endl;
00178         } else {
00179             while (cur) {
00180                 cout << cur->key_ << "(" << cur->values_->size() << ")" << " \t";
00181                 cur = cur->next_;
00182             }
00183             cout << endl;
00184         }
00185     }
00186
00187     cout << endl << "Count: " << count << endl << "Collisions: " << collisions;
00188 }
00189
00190 private:
00191     Item *items_[SIZE];
00192     long long count;
00193     long long unq_count = 0;
00194     long long collisions = 0;
00195
00196 private:
00197     void NullTable() {
00198         for (int i = 0; i < SIZE; ++i) {
00199             items_[i] = nullptr;
00200         }
00201     }
00202
00203     void SupportDelete(Item *cur) {
00204         while (cur) {
00205             Item *next_node = cur->next_;
00206             delete cur->values_;
00207             delete cur;
00208             cur = next_node;
00209         }
00210     }
00211 }
00212 };
00213
00214 #endif

```

6.7 io.h File Reference

Interface for input/output functions: parsing CSV files and save result to files.

```
#include "flower.h"
#include <string>
#include <vector>
```

Functions

- vector< [Flower](#) > [parserCSV](#) (string filename)
Reads a CSV file and returns a vector of [Flower](#) objects.
- void [saveRes](#) (vector< [Flower](#) > &source, long size, [Flower](#) target)
Run multiple search algorithms on a dataset of [Flower](#) objects and save results to files.

6.7.1 Function Documentation

6.7.1.1 [parserCSV\(\)](#)

```
vector< Flower > parserCSV (
    string filename)
```

Parameters

<i>filename</i>	Path to the input CSV file.
-----------------	-----------------------------

Exceptions

<i>runtime_error</i>	if the file cannot be opened.
----------------------	-------------------------------

Returns

Vector of [Flower](#) objects loaded from the file.

The function opens the given CSV file and discards the first line (assumed to be a header). Each subsequent line must contain exactly four comma-separated fields:

1. name
2. color
3. smell
4. regions — a list of one or more region names enclosed in square brackets, e.g. [[Region1](#),[Region2](#), ...]

Internally, the parser locates the first three commas to extract the name, color and smell fields. It then strips the surrounding brackets from the remaining substring and splits it on commas to obtain each region. A [Flower](#) is constructed with these values and appended to the result vector. If the file contains no data lines (only a header or is empty), an empty vector is returned.

6.7.1.2 saveRes()

```
void saveRes (
    vector< Flower > & source,
    long size,
    Flower target)
```

This function performs the following steps:

1. Measures and records execution time for linear search, binary search tree search, red-black tree search, hash table search, and multimap search.
2. Writes matching records for each algorithm into separate output files named: "<size>_linear.txt", "<size>_binary.txt", "<size>_rb.txt", "<size>_hash.txt", "<size>_multimap.txt".
3. Appends timing information (and collision count for hash) into "info_time.txt".

Parameters

<i>source</i>	Reference to a vector of Flower objects to be searched.
<i>size</i>	Number of elements in the source vector (expected to match <code>source.size()</code>).
<i>target</i>	The Flower object to search for.

Exceptions

<code>std::runtime_error</code>	If any output file cannot be opened for writing.
---------------------------------	--

6.8 io.h

[Go to the documentation of this file.](#)

```
00001
00006
00007 #ifndef IO_H
00008 #define IO_H
00009
00010 #include "flower.h"
00011 #include <string>
00012 #include <vector>
00013
00014 using namespace std;
00015
00033 vector<Flower> parserCSV(string filename);
00034
00051 void saveRes(vector<Flower>& source, long size, Flower target);
00052
00053 #endif
```

6.9 linear.h File Reference

Contains functions for performing linear search on an array.

```
#include <vector>
```

Functions

- `template<class T>`
`int linearSearch (T a[], long start, long size, T b)`
Perform a linear search to find the first occurrence of an element in an array.
- `template<class T>`
`vector< int > searchAll (T a[], long size, T b)`
Find all occurrences of a given element in an array.

6.9.1 Detailed Description

This file provides two templated functions:

- `linearSearch`: Finds the first occurrence of a given element in an array.
- `searchAll`: Finds all occurrences of a given element and returns their indices.

6.9.2 Function Documentation

6.9.2.1 `linearSearch()`

```
template<class T>
int linearSearch (
    T a[],
    long start,
    long size,
    T b)
```

Parameters

<i>a</i>	Pointer to the array of elements of type T.
<i>start</i>	Index in the array from which to begin the search.
<i>size</i>	Total number of elements in the array (one past the last valid index).
<i>b</i>	The value to search for in the array.

Returns

The index of the first matching element, or -1 if the element is not found.

6.9.2.2 `searchAll()`

```
template<class T>
vector< int > searchAll (
    T a[],
    long size,
    T b)
```

This function uses `linearSearch` to locate each instance of the target value in the array. It stores each found index in a `std::vector` and returns the vector.

Parameters

<i>a</i>	Pointer to the array of elements of type T.
<i>size</i>	Total number of elements in the array.
<i>b</i>	The value to search for in the array.

Returns

A `std::vector<int>` containing the indices where the element `b` was found. If the element is not found, the vector will be empty.

6.10 linear.h

[Go to the documentation of this file.](#)

```

00001
00007
00008 #ifndef LINEAR_H
00009 #define LINEAR_H
00010
00011 #include <vector>
00012 using namespace std;
00013
00023 template<class T>
00024 int linearSearch(T a[], long start, long size, T b) {
00025     for (long i = start; i < size; ++i) {
00026         if (a[i] == b) {
00027             return i;
00028         }
00029     }
00030     return -1;
00031 }
00032
00045 template<class T>
00046 vector<int> searchAll(T a[], long size, T b) {
00047     vector<int> res;
00048
00049     int i = linearSearch(a, 0, size, b);
00050     while (i != -1) {
00051         res.push_back(i);
00052         i = linearSearch(a, i+1, size, b);
00053     }
00054
00055     return res;
00056 }
00057
00058 #endif

```

6.11 rb_tree.h File Reference

Defines a templated Red-Black [Tree](#) with insertion, search, and traversal functions.

```

#include <string>
#include <stdexcept>
#include <iostream>
#include <vector>

```

Classes

- class [RBNode< T >](#)
Represents a node in a Red-Black [Tree](#).
- class [RBTree< T >](#)
Implements a Red-Black [Tree](#).

Enumerations

- enum `Color` { `RED` , `BLACK` }

Enumeration for node color in a Red-Black `Tree`.

6.11.1 Detailed Description

This file provides:

- `RBNode`: A node structure storing a value, color, and pointers to parent and children.
- `RBTree`: A Red-Black `Tree` implementation supporting insertion, search (single and all occurrences), and printing the tree.

6.12 rb_tree.h

[Go to the documentation of this file.](#)

```

00001
00002
00003 #ifndef RB_TREE_H
00004 #define RB_TREE_H
00005
00006 #include <string>
00007 #include <stdexcept>
00008 #include <iostream>
00009 #include <vector>
00010
00011 using namespace std;
00012
00013 typedef enum { RED, BLACK } Color;
00014
00015 template <typename T>
00016 class RBNode {
00017 public:
00018     vector<T> values_;
00019     Color color_;
00020
00021     RBNode *left_, *right_, *parent_;
00022
00023 public:
00024     RBNode() {
00025         color_ = RED;
00026         left_ = nullptr;
00027         right_ = nullptr;
00028         parent_ = nullptr;
00029         values_ = T();
00030     }
00031
00032     RBNode(const T value) {
00033         values_.push_back(value);
00034         color_ = RED;
00035         left_ = nullptr;
00036         right_ = nullptr;
00037         parent_ = nullptr;
00038     }
00039
00040     RBNode& operator=(const RBNode& other) {
00041         values_ = other.values_;
00042         color_ = other.color_;
00043         left_ = other.left_;
00044         right_ = other.right_;
00045         parent_ = other.parent_;
00046
00047         return *this;
00048     }
00049 };
00050
00051 template <typename T>
00052 class RBTree {
00053 public:

```

```

00086
00087     RBTTree() { root_ = nullptr; }
00088     RBTTree(T value) {
00089         root_ = new RBNode<T>(value);
00090         root_>color_ = BLACK;
00091     }
00092     ~RBTTree() { DelTree(root_); }
00093
00094
00095
00096
00097
00106     void Insert(const T& value) {
00107         if (!root_) {
00108             root_ = new RBNode<T>(value);
00109             root_>color_ = BLACK;
00110             return;
00111         }
00112
00113         RBNode<T> *target = root_;
00114         RBNode<T> *parent = nullptr;
00115
00116         while(target) {
00117             parent = target;
00118             if (value < target->values_[0]) {
00119                 target = target->left_;
00120             } else if (value > target->values_[0]){
00121                 target = target->right_;
00122             } else {
00123                 target->values_.push_back(value);
00124                 return;
00125             }
00126         }
00127
00128         target = new RBNode<T>(value);
00129         target->parent_ = parent;
00130
00131         if (value < parent->values_[0]) {
00132             parent->left_ = target;
00133         } else {
00134             parent->right_ = target;
00135         }
00136
00137         Balance(target);
00138     }
00139
00140     // /// @brief Search for all nodes containing a given value.
00141     // /// @param value Reference to the value to search for.
00142     // /// @return Pointer to the node containing the value, or nullptr if not found.
00143     // RBNode<T>* Search(const T& value) {
00144     //     return SupSearch(root_, value);
00145     // }
00146
00147
00148
00149
00150     RBNode<T>* SearchAll(const T& value) {
00151         if (root_) {
00152             RBNode<T> *cur = root_;
00153
00154             while (cur) {
00155                 if (cur->values_[0] == value) {
00156                     return cur;
00157                 }
00158
00159                 if (value < cur->values_[0]) {
00160                     cur = cur->left_;
00161                 } else {
00162                     cur = cur->right_;
00163                 }
00164             }
00165         }
00166
00167         return nullptr;
00168     }
00169
00170
00171     void PrintTree() {
00172         SupportPrint(root_);
00173     }
00174
00175
00176 private:
00177     RBNode<T> *root_;
00178
00179 private:
00180
00181
00182
00183     // /**
00184     //  * \brief Helper function to search for a value starting at a given node.
00185     //  *
00186     //  * Traverses the subtree like a standard BST search: if the current node's value
00187     //  * matches, returns it; if the value is less, goes left; otherwise, goes right.
00188     //  *
00189     //  * \param node Pointer to the current node from which to start searching.
00190     //  * \param value Reference to the value to search for.

```

```

00191 // * \return Pointer to the node containing the value, or nullptr if not found.
00192 // */
00193 // RBNODE<T>* SupSearch(RBNODE<T> *node, const T& value) {
00194 //     if (node) {
00195 //         RBNODE<T> *cur = node;
00196 //
00197 //         while (cur) {
00198 //             if (cur->values_[0] == value) {
00199 //                 return cur;
00200 //             }
00201 //
00202 //             if (value < cur->value_) {
00203 //                 cur = cur->left_;
00204 //             } else {
00205 //                 cur = cur->right_;
00206 //             }
00207 //         }
00208 //     }
00209 //
00210 //     return nullptr;
00211 // }
00212
00214 void SupportPrint(RBNODE<T> *root) {
00215     if (root) {
00216         cout << root->values_ << " " << root->color_ << endl;
00217
00218         SupportPrint(root->left_);
00219         SupportPrint(root->right_);
00220     }
00221 }
00222
00224 void DelTree(RBNODE<T> *root) {
00225     if (root) {
00226         DelTree(root->left_);
00227         DelTree(root->right_);
00228         delete root;
00229     }
00230 }
00231
00249 void Balance(RBNODE<T> *node) {
00250     RBNODE<T> *dad = node->parent_;
00251
00252     while (dad && dad->color_ == RED) {
00253         RBNODE<T> *grand = dad->parent_;
00254         RBNODE<T> *uncle;
00255
00256         if (!grand) {
00257             return;
00258         } else if (dad == grand->left_) {
00259             uncle = grand->right_;
00260
00261             // 1
00262             if (uncle && uncle->color_ == RED) {
00263                 dad->color_ = BLACK;
00264                 uncle->color_ = BLACK;
00265                 grand->color_ = RED;
00266
00267                 node = grand;
00268                 dad = node->parent_;
00269                 if (dad) { grand = dad->parent_; }
00270
00271                 continue;
00272             } else { // 2
00273                 if (dad->right_ == node) {
00274                     LeftRotate(node, dad, grand);
00275                     node = dad;
00276                     dad = node->parent_;
00277                 }
00278                 dad->color_ = BLACK;
00279                 grand->color_ = RED;
00280                 RightRotate(dad, grand, grand->parent_);
00281                 break;
00282             }
00283         } else {
00284             uncle = grand->left_;
00285
00286             if (uncle && uncle->color_ == RED) {
00287                 dad->color_ = BLACK;
00288                 uncle->color_ = BLACK;
00289                 grand->color_ = RED;
00290
00291                 node = grand;
00292                 dad = node->parent_;
00293                 if (dad) { grand = dad->parent_; }
00294
00295                 continue;
00296             } else {

```

```

00297         if (dad->left_ == node) {
00298             RightRotate(node, dad, grand);
00299             node = dad;
00300             dad = node->parent_;
00301         }
00302         dad->color_ = BLACK;
00303         grand->color_ = RED;
00304         LeftRotate(dad, grand, grand->parent_);
00305         break;
00306     }
00307 }
00308 }
00309
00310     root_->color_ = BLACK;
00311 }
00312
00313 void LeftRotate(RBNode<T> *child, RBNode<T> *dad, RBNode<T> *grand) {
00314     RBNode<T> *grandson = child->left_;
00315
00316     dad->right_ = grandson;
00317     if (grandson) { grandson->parent_ = dad; }
00318
00319     child->left_ = dad;
00320     dad->parent_ = child;
00321
00322     child->parent_ = grand;
00323     if (!grand) {
00324         root_ = child;
00325     } else if (grand->left_ == dad) {
00326         grand->left_ = child;
00327     } else {
00328         grand->right_ = child;
00329     }
00330 }
00331
00332 void RightRotate(RBNode<T> *child, RBNode<T> *dad, RBNode<T> *grand) {
00333     RBNode<T> *grandson = child->right_;
00334
00335     dad->left_ = grandson;
00336     if (grandson) { grandson->parent_ = dad; }
00337
00338     dad->parent_ = child;
00339     child->right_ = dad;
00340
00341     child->parent_ = grand;
00342     if (!grand) {
00343         root_ = child;
00344     } else if (grand->right_ == dad) {
00345         grand->right_ = child;
00346     } else {
00347         grand->left_ = child;
00348     }
00349 }
00350 };
00351 #endif

```

Index

- binary_tree.h, [19](#)
- Constructors and destructor, [7](#)
 - RBTree, [7](#)
- EqFlowers
 - Flower, [12](#)
- Flower, [11](#)
 - EqFlowers, [12](#)
- flower.h, [21](#)
- hash.h, [21](#), [23](#)
 - hashFunc_rs, [22](#)
- hashFunc_rs
 - hash.h, [22](#)
- HashTable, [12](#)
 - HashTable, [13](#)
 - Search, [13](#)
- Insert
 - Insert, search and visualization of the tree, [8](#)
- Insert, search and visualization of the tree, [7](#)
 - Insert, [8](#)
 - Search, [8](#)
 - SearchAll, [8](#), [9](#)
- io.h, [25](#), [26](#)
 - parserCSV, [25](#)
 - saveRes, [25](#)
- Item, [14](#)
- linear.h, [26](#), [28](#)
 - linearSearch, [27](#)
 - searchAll, [27](#)
- linearSearch
 - linear.h, [27](#)
- Node< T >, [14](#)
- operator=
 - RBNode< T >, [15](#)
- parserCSV
 - io.h, [25](#)
- rb_tree.h, [28](#), [29](#)
- RBNode< T >, [15](#)
 - operator=, [15](#)
- RBTree
 - Constructors and destructor, [7](#)
- RBTree< T >, [16](#)
- saveRes
 - io.h, [25](#)
- Search
 - HashTable, [13](#)
 - Insert, search and visualization of the tree, [8](#)
- SearchAll
 - Insert, search and visualization of the tree, [8](#), [9](#)
- searchAll
 - linear.h, [27](#)
- Supporting methods for basic methods, [9](#)
- Tree< T >, [17](#)