

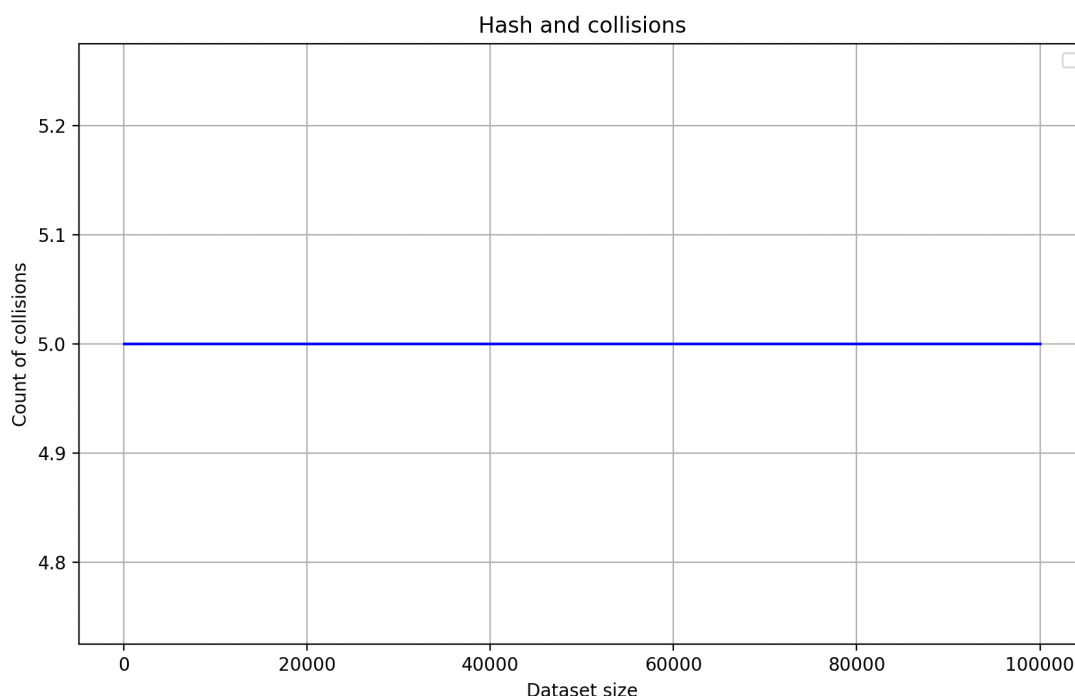
Отчёт

по теме «Алгоритмы поиска данных»

Гриднева Е. В., СКБ222

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 числа коллизий остаётся плоским.

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

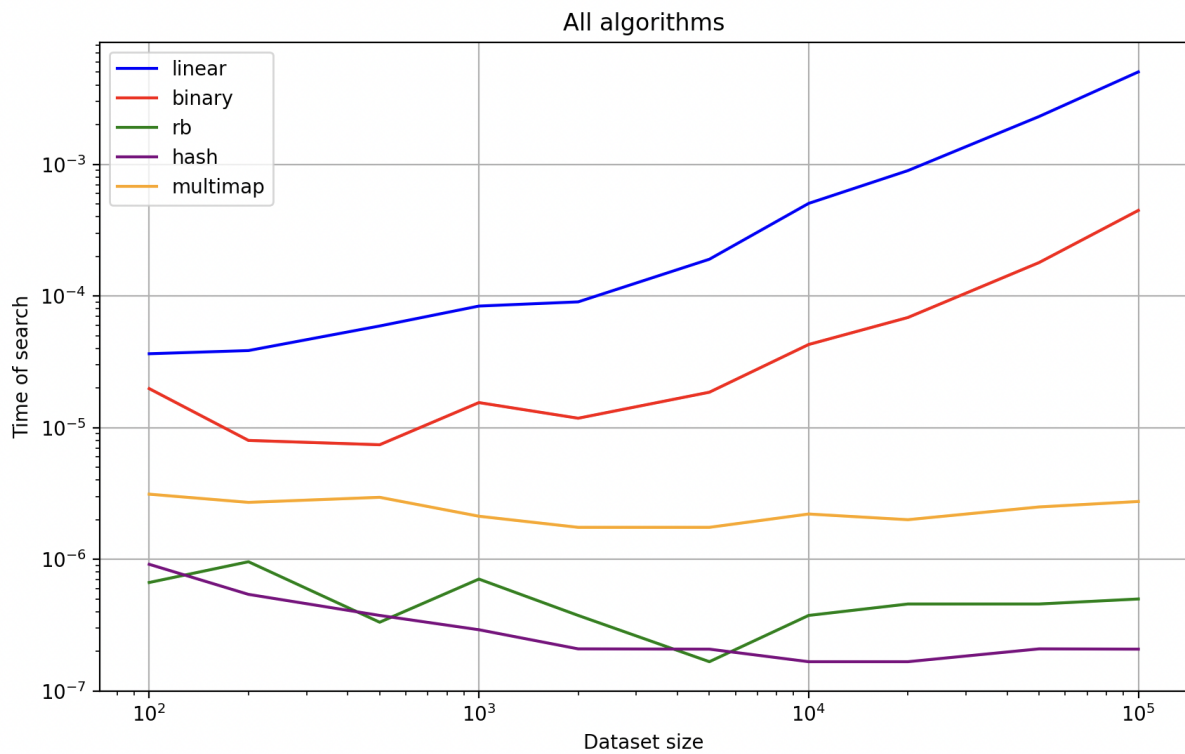
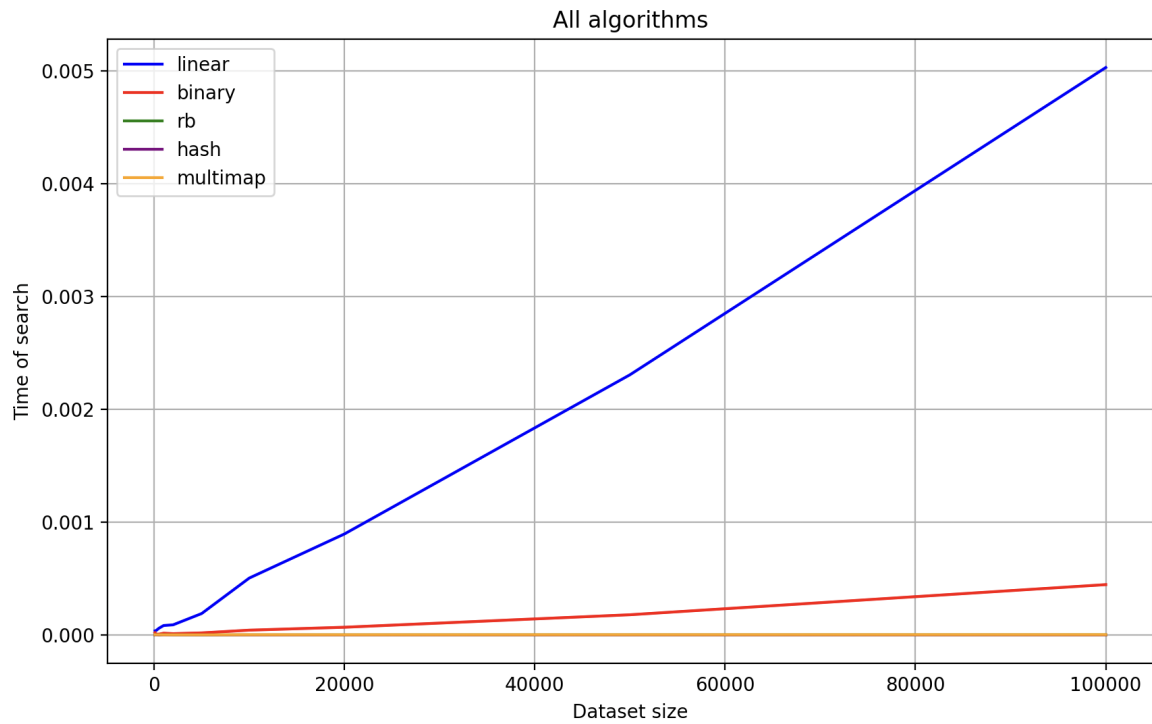
Наиболее эффективен и самый быстрый — поиск по собственной хеш-таблице (константное $O(1)$).

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
00008
00009 #ifndef HASH_H
00010 #define HASH_H
00011
00012 #include <string>
00013 #include <vector>
00014 #include <iostream>
00015 #include "flower.h"
00016
00018 #define SIZE 14
00019 using namespace std;
00020
00030 unsigned int hashFunc_rs(string key) {
00031     unsigned int a = 63689;
00032     unsigned int b = 378551;
00033     unsigned int hash = 0;
00034     unsigned int i = 0;
00035
00036     int len = key.length();
00037
00038     for (i = 0; i < len; ++i) {
00039         hash = hash * a + (unsigned char)key[i];
00040         a = a * b;
00041     }
00042
00043     return (hash % SIZE);
00044 }
00045
00055 class Item {
00056 public:
00057     string key_;
00058     vector<Flower> *values_;
00059     Item *next_;
00060
00061 public:
00064     Item() {
00065         key_ = "";
00066         values_ = new vector<Flower>();
00067         next_ = nullptr;
00068     }
00069
00070     Item(const string& key, const Flower& value) {
00071         key_ = key;
00072         values_ = new vector<Flower>();
00073         values_>push_back(value);
00074         next_ = nullptr;
00075     }
00077 };
00078
00090 class HashTable {
00091 public:
00104     HashTable(const vector<Flower>& data) {
00105         NullTable();
00106
00107         count = data.size();
00108
00109         for (size_t i = 0; i < count; ++i) {
00110             string key = data[i].GetName();
00111             Flower value = data[i];
00112             unsigned int hash = hashFunc_rs(key);
00113
00114             if (!items_[hash]) {
00115                 items_[hash] = new Item(key, value);
00116                 unq_count += 1;
00117             } else {
00118                 Item *where = items_[hash];
00119                 int is_collis = 1;
00120
00121                 while (true) {
00122                     if (key == where->key_) {
00123                         where->values_>push_back(value);
00124                         is_collis = 0;
00125                         break;
00126                     } else {
00127                         if (where->next_) {
00128                             where = where->next_;
00129                         } else {
00130                             break;
00131                         }
00132                     }
00133                 }
00134             }
00135         }
00136     }
00137
00138     ~HashTable() {
00139         for (size_t i = 0; i < count; ++i) {
00140             delete items_[i];
00141         }
00142     }
00143
00144     void NullTable() {
00145         items_.clear();
00146         unq_count = 0;
00147     }
00148
00149     void Add(const Flower& f) {
00150         string key = f.GetName();
00151         unsigned int hash = hashFunc_rs(key);
00152         if (!items_[hash]) {
00153             items_[hash] = new Item(key, f);
00154             unq_count += 1;
00155         } else {
00156             Item *where = items_[hash];
00157             int is_collis = 1;
00158
00159             while (true) {
00160                 if (key == where->key_) {
00161                     where->values_>push_back(f);
00162                     is_collis = 0;
00163                     break;
00164                 } else {
00165                     if (where->next_) {
00166                         where = where->next_;
00167                     } else {
00168                         break;
00169                     }
00170                 }
00171             }
00172         }
00173     }
00174
00175     void Print() const {
00176         for (size_t i = 0; i < count; ++i) {
00177             if (items_[i]) {
00178                 cout << "Key: " << items_[i]->key_ << endl;
00179                 for (size_t j = 0; j < items_[i]->values_>size(); ++j) {
00180                     cout << "Value: " << items_[i]->values_[j].GetName() << endl;
00181                 }
00182             }
00183         }
00184     }
00185
00186     size_t GetCount() const {
00187         return count;
00188     }
00189
00190     int GetUnqCount() const {
00191         return unq_count;
00192     }
00193
00194     const vector<Flower> *GetItems() const {
00195         return items_;
00196     }
00197
00198     const vector<Flower> *GetValues(const string& key) const {
00199         unsigned int hash = hashFunc_rs(key);
00200         if (items_[hash]) {
00201             return items_[hash]->values_;
00202         }
00203         return nullptr;
00204     }
00205
00206     Item *GetItem(const string& key) const {
00207         unsigned int hash = hashFunc_rs(key);
00208         if (items_[hash]) {
00209             return items_[hash];
00210         }
00211         return nullptr;
00212     }
00213
00214     void Remove(const string& key) {
00215         unsigned int hash = hashFunc_rs(key);
00216         if (items_[hash]) {
00217             delete items_[hash];
00218             items_[hash] = nullptr;
00219             unq_count -= 1;
00220         }
00221     }
00222
00223     void RemoveAll() {
00224         NullTable();
00225     }
00226
00227     void Clear() {
00228         NullTable();
00229     }
00230
00231     void SetCount(size_t count) {
00232         this->count = count;
00233     }
00234
00235     void SetUnqCount(int unq_count) {
00236         this->unq_count = unq_count;
00237     }
00238
00239     void SetItems(const vector<Flower> *items) {
00240         items_ = *items;
00241     }
00242
00243     void SetValues(const vector<Flower> *values) {
00244         values_ = *values;
00245     }
00246
00247     void SetKey(const string& key) {
00248         key_ = key;
00249     }
00250
00251     void SetValue(const Flower& value) {
00252         values_>push_back(value);
00253     }
00254
00255     void SetNext(Item *next) {
00256         next_ = next;
00257     }
00258
00259     const string& GetKey() const {
00260         return key_;
00261     }
00262
00263     const vector<Flower> *GetValues() const {
00264         return values_;
00265     }
00266
00267     Item *GetNext() const {
00268         return next_;
00269     }
00270
00271     const string& GetKey_() const {
00272         return key_;
00273     }
00274
00275     const vector<Flower> *GetValues_() const {
00276         return values_;
00277     }
00278
00279     Item *GetNext_() const {
00280         return next_;
00281     }
00282
00283     const string& GetKey__() const {
00284         return key_;
00285     }
00286
00287     const vector<Flower> *GetValues__() const {
00288         return values_;
00289     }
00290
00291     Item *GetNext__() const {
00292         return next_;
00293     }
00294
00295     const string& GetKey___() const {
00296         return key_;
00297     }
00298
00299     const vector<Flower> *GetValues___() const {
00300         return values_;
00301     }
00302
00303     Item *GetNext___() const {
00304         return next_;
00305     }
00306
00307     const string& GetKey____() const {
00308         return key_;
00309     }
00310
00311     const vector<Flower> *GetValues____() const {
00312         return values_;
00313     }
00314
00315     Item *GetNext____() const {
00316         return next_;
00317     }
00318
00319     const string& GetKey_____() const {
00320         return key_;
00321     }
00322
00323     const vector<Flower> *GetValues_____() const {
00324         return values_;
00325     }
00326
00327     Item *GetNext_____() const {
00328         return next_;
00329     }
00330
00331     const string& GetKey_____.() const {
00332         return key_;
00333     }
00334
00335     const vector<Flower> *GetValues_____.() const {
00336         return values_;
00337     }
00338
00339     Item *GetNext_____.() const {
00340         return next_;
00341     }
00342
00343     const string& GetKey_____.() const {
00344         return key_;
00345     }
00346
00347     const vector<Flower> *GetValues_____.() const {
00348         return values_;
00349     }
00350
00351     Item *GetNext_____.() const {
00352         return next_;
00353     }
00354
00355     const string& GetKey_____.() const {
00356         return key_;
00357     }
00358
00359     const vector<Flower> *GetValues_____.() const {
00360         return values_;
00361     }
00362
00363     Item *GetNext_____.() const {
00364         return next_;
00365     }
00366
00367     const string& GetKey_____.() const {
00368         return key_;
00369     }
00370
00371     const vector<Flower> *GetValues_____.() const {
00372         return values_;
00373     }
00374
00375     Item *GetNext_____.() const {
00376         return next_;
00377     }
00378
00379     const string& GetKey_____.() const {
00380         return key_;
00381     }
00382
00383     const vector<Flower> *GetValues_____.() const {
00384         return values_;
00385     }
00386
00387     Item *GetNext_____.() const {
00388         return next_;
00389     }
00390
00391     const string& GetKey_____.() const {
00392         return key_;
00393     }
00394
00395     const vector<Flower> *GetValues_____.() const {
00396         return values_;
00397     }
00398
00399     Item *GetNext_____.() const {
00400         return next_;
00401     }
00402
00403     const string& GetKey_____.() const {
00404         return key_;
00405     }
00406
00407     const vector<Flower> *GetValues_____.() const {
00408         return values_;
00409     }
00410
00411     Item *GetNext_____.() const {
00412         return next_;
00413     }
00414
00415     const string& GetKey_____.() const {
00416         return key_;
00417     }
00418
00419     const vector<Flower> *GetValues_____.() const {
00420         return values_;
00421     }
00422
00423     Item *GetNext_____.() const {
00424         return next_;
00425     }
00426
00427     const string& GetKey_____.() const {
00428         return key_;
00429     }
00430
00431     const vector<Flower> *GetValues_____.() const {
00432         return values_;
00433     }
00434
00435     Item *GetNext_____.() const {
00436         return next_;
00437     }
00438
00439     const string& GetKey_____.() const {
00440         return key_;
00441     }
00442
00443     const vector<Flower> *GetValues_____.() const {
00444         return values_;
00445     }
00446
00447     Item *GetNext_____.() const {
00448         return next_;
00449     }
00450
00451     const string& GetKey_____.() const {
00452         return key_;
00453     }
00454
00455     const vector<Flower> *GetValues_____.() const {
00456         return values_;
00457     }
00458
00459     Item *GetNext_____.() const {
00460         return next_;
00461     }
00462
00463     const string& GetKey_____.() const {
00464         return key_;
00465     }
00466
00467     const vector<Flower> *GetValues_____.() const {
00468         return values_;
00469     }
00470
00471     Item *GetNext_____.() const {
00472         return next_;
00473     }
00474
00475     const string& GetKey_____.() const {
00476         return key_;
00477     }
00478
00479     const vector<Flower> *GetValues_____.() const {
00480         return values_;
00481     }
00482
00483     Item *GetNext_____.() const {
00484         return next_;
00485     }
00486
00487     const string& GetKey_____.() const {
00488         return key_;
00489     }
00490
00491     const vector<Flower> *GetValues_____.() const {
00492         return values_;
00493     }
00494
00495     Item *GetNext_____.() const {
00496         return next_;
00497     }
00498
00499     const string& GetKey_____.() const {
00500         return key_;
00501     }
00502
00503     const vector<Flower> *GetValues_____.() const {
00504         return values_;
00505     }
00506
00507     Item *GetNext_____.() const {
00508         return next_;
00509     }
00510
00511     const string& GetKey_____.() const {
00512         return key_;
00513     }
00514
00515     const vector<Flower> *GetValues_____.() const {
00516         return values_;
00517     }
00518
00519     Item *GetNext_____.() const {
00520         return next_;
00521     }
00522
00523     const string& GetKey_____.() const {
00524         return key_;
00525     }
00526
00527     const vector<Flower> *GetValues_____.() const {
00528         return values_;
00529     }
00530
00531     Item *GetNext_____.() const {
00532         return next_;
00533     }
00534
00535     const string& GetKey_____.() const {
00536         return key_;
00537     }
00538
00539     const vector<Flower> *GetValues_____.() const {
00540         return values_;
00541     }
00542
00543     Item *GetNext_____.() const {
00544         return next_;
00545     }
00546
00547     const string& GetKey_____.() const {
00548         return key_;
00549     }
00550
00551     const vector<Flower> *GetValues_____.() const {
00552         return values_;
00553     }
00554
00555     Item *GetNext_____.() const {
00556         return next_;
00557     }
00558
00559     const string& GetKey_____.() const {
00560         return key_;
00561     }
00562
00563     const vector<Flower> *GetValues_____.() const {
00564         return values_;
00565     }
00566
00567     Item *GetNext_____.() const {
00568         return next_;
00569     }
00570
00571     const string& GetKey_____.() const {
00572         return key_;
00573     }
00574
00575     const vector<Flower> *GetValues_____.() const {
00576         return values_;
00577     }
00578
00579     Item *GetNext_____.() const {
00580         return next_;
00581     }
00582
00583     const string& GetKey_____.() const {
00584         return key_;
00585     }
00586
00587     const vector<Flower> *GetValues_____.() const {
00588         return values_;
00589     }
00590
00591     Item *GetNext_____.() const {
00592         return next_;
00593     }
00594
00595     const string& GetKey_____.() const {
00596         return key_;
00597     }
00598
00599     const vector<Flower> *GetValues_____.() const {
00600         return values_;
00601     }
00602
00603     Item *GetNext_____.() const {
00604         return next_;
00605     }
00606
00607     const string& GetKey_____.() const {
00608         return key_;
00609     }
00610
00611     const vector<Flower> *GetValues_____.() const {
00612         return values_;
00613     }
00614
00615     Item *GetNext_____.() const {
00616         return next_;
00617     }
00618
00619     const string& GetKey_____.() const {
00620         return key_;
00621     }
00622
00623     const vector<Flower> *GetValues_____.() const {
00624         return values_;
00625     }
00626
00627     Item *GetNext_____.() const {
00628         return next_;
00629     }
00630
00631     const string& GetKey_____.() const {
00632         return key_;
00633     }
00634
00635     const vector<Flower> *GetValues_____.() const {
00636         return values_;
00637     }
00638
00639     Item *GetNext_____.() const {
00640         return next_;
00641     }
00642
00643     const string& GetKey_____.() const {
00644         return key_;
00645     }
00646
00647     const vector<Flower> *GetValues_____.() const {
00648         return values_;
00649     }
00650
00651     Item *GetNext_____.() const {
00652         return next_;
00653     }
00654
00655     const string& GetKey_____.() const {
00656         return key_;
00657     }
00658
00659     const vector<Flower> *GetValues_____.() const {
00660         return values_;
00661     }
00662
00663     Item *GetNext_____.() const {
00664         return next_;
00665     }
00666
00667     const string& GetKey_____.() const {
00668         return key_;
00669     }
00670
00671     const vector<Flower> *GetValues_____.() const {
00672         return values_;
00673     }
00674
00675     Item *GetNext_____.() const {
00676         return next_;
00677     }
00678
00679     const string& GetKey_____.() const {
00680         return key_;
00681     }
00682
00683     const vector<Flower> *GetValues_____.() const {
00684         return values_;
00685     }
00686
00687     Item *GetNext_____.() const {
00688         return next_;
00689     }
00690
00691     const string& GetKey_____.() const {
00692         return key_;
00693     }
00694
00695     const vector<Flower> *GetValues_____.() const {
00696         return values_;
00697     }
00698
00699     Item *GetNext_____.() const {
00700         return next_;
00701     }
00702
00703     const string& GetKey_____.() const {
00704         return key_;
00705     }
00706
00707     const vector<Flower> *GetValues_____.() const {
00708         return values_;
00709     }
00710
00711     Item *GetNext_____.() const {
00712         return next_;
00713     }
00714
00715     const string& GetKey_____.() const {
00716         return key_;
00717     }
00718
00719     const vector<Flower> *GetValues_____.() const {
00720         return values_;
00721     }
00722
00723     Item *GetNext_____.() const {
00724         return next_;
00725     }
00726
00727     const string& GetKey_____.() const {
00728         return key_;
00729     }
00730
00731     const vector<Flower> *GetValues_____.() const {
00732         return values_;
00733     }
00734
00735     Item *GetNext_____.() const {
00736         return next_;
00737     }
00738
00739     const string& GetKey_____.() const {
00740         return key_;
00741     }
00742
00743     const vector<Flower> *GetValues_____.() const {
00744         return values_;
00745     }
00746
00747     Item *GetNext_____.() const {
00748         return next_;
00749     }
00750
00751     const string& GetKey_____.() const {
00752         return key_;
00753     }
00754
00755     const vector<Flower> *GetValues_____.() const {
00756         return values_;
00757     }
00758
00759     Item *GetNext_____.() const {
00760         return next_;
00761     }
00762
00763     const string& GetKey_____.() const {
00764         return key_;
00765     }
00766
00767     const vector<Flower> *GetValues_____.() const {
00768         return values_;
00769     }
00770
00771     Item *GetNext_____.() const {
00772         return next_;
00773     }
00774
00775     const string& GetKey_____.() const {
00776         return key_;
00777     }
00778
00779     const vector<Flower> *GetValues_____.() const {
00780         return values_;
00781     }
00782
00783     Item *GetNext_____.() const {
00784         return next_;
00785     }
00786
00787     const string& GetKey_____.() const {
00788         return key_;
00789     }
00790
00791     const vector<Flower> *GetValues_____.() const {
00792         return values_;
00793     }
00794
00795     Item *GetNext_____.() const {
00796         return next_;
00797     }
00798
00799     const string& GetKey_____.() const {
00800         return key_;
00801     }
00802
00803     const vector<Flower> *GetValues_____.() const {
00804         return values_;
00805     }
00806
00807     Item *GetNext_____.() const {
00808         return next_;
00809     }
00810
00811     const string& GetKey_____.() const {
00812         return key_;
00813     }
00814
00815     const vector<Flower> *GetValues_____.() const {
00816         return values_;
00817     }
00818
00819     Item *GetNext_____.() const {
00820         return next_;
00821     }
00822
00823     const string& GetKey_____.() const {
00824         return key_;
00825     }
00826
00827     const vector<Flower> *GetValues_____.() const {
00828         return values_;
00829     }
00830
00831     Item *GetNext_____.() const {
00832         return next_;
00833     }
00834
00835     const string& GetKey_____.() const {
00836         return key_;
00837     }
00838
00839     const vector<Flower> *GetValues_____.() const {
00840         return values_;
00841     }
00842
00843     Item *GetNext_____.() const {
00844         return next_;
00845     }
00846
00847     const string& GetKey_____.() const {
00848         return key_;
00849     }
00850
00851     const vector<Flower> *GetValues_____.() const {
00852         return values_;
00853     }
00854
00855     Item *GetNext_____.() const {
00856         return next_;
00857     }
00858
00859     const string& GetKey_____.() const {
00860         return key_;
00861     }
00862
00863     const vector<Flower> *GetValues_____.() const {
00864         return values_;
00865     }
00866
00867     Item *GetNext_____.() const {
00868         return next_;
00869     }
00870
00871     const string& GetKey_____.() const {
00872         return key_;
00873     }
00874
00875     const vector<Flower> *GetValues_____.() const {
00876         return values_;
00877     }
00878
00879     Item *GetNext_____.() const {
00880         return next_;
00881     }
00882
00883     const string& GetKey_____.() const {
00884         return key_;
00885     }
00886
00887     const vector<Flower> *GetValues_____.() const {
00888         return values_;
00889     }
00890
00891     Item *GetNext_____.() const {
00892         return next_;
00893     }
00894
00895     const string& GetKey_____.() const {
00896         return key_;
00897     }
00898
00899     const vector<Flower> *GetValues_____.() const {
00900         return values_;
00901     }
00902
00903     Item *GetNext_____.() const {
00904         return next_;
00905     }
00906
00907     const string& GetKey_____.() const {
00908         return key_;
00909     }
00910
00911     const vector<Flower> *GetValues_____.() const {
00912         return values_;
00913     }
00914
00915     Item *GetNext_____.() const {
00916         return next_;
00917     }
00918
00919     const string& GetKey_____.() const {
00920         return key_;
00921     }
00922
00923     const vector<Flower> *GetValues_____.() const {
00924         return values_;
00925     }
00926
00927     Item *GetNext_____.() const {
00928         return next_;
00929     }
00930
00931     const string& GetKey_____.() const {
00932         return key_;
00933     }
00934
00935     const vector<Flower> *GetValues_____.() const {
00936         return values_;
00937     }
00938
00939     Item *GetNext_____.() const {
00940         return next_;
00941     }
00942
00943     const string& GetKey_____.() const {
00944         return key_;
00945     }
00946
00947     const vector<Flower> *GetValues_____.() const {
00948         return values_;
00949     }
00950
00951     Item *GetNext_____.() const {
00952         return next_;
00953     }
00954
00955     const string& GetKey_____.() const {
00956         return key_;
00957     }
00958
00959     const vector<Flower> *GetValues_____.() const {
00960         return values_;
00961     }
00962
00963     Item *GetNext_____.() const {
00964         return next_;
00965     }
00966
00967     const string& GetKey_____.() const {
00968         return key_;
00969     }
00970
00971     const vector<Flower> *GetValues_____.() const {
00972         return values_;
00973     }
00974
00975     Item *GetNext_____.() const {
00976         return next_;
00977     }
00978
00979     const string& GetKey_____.() const {
00980         return key_;
00981     }
00982
00983     const vector<Flower> *GetValues_____.() const {
00984         return values_;
00985     }
00986
00987     Item *GetNext_____.() const {
00988         return next_;
00989     }
00990
00991     const string& GetKey_____.() const {
00992         return key_;
00993     }
00994
00995     const vector<Flower> *GetValues_____.() const {
00996         return values_;
00997     }
00998
00999     Item *GetNext_____.() const {
01000         return next_;
01001     }
01002
01003     const string& GetKey_____.() const {
01004         return key_;
01005     }
01006
01007     const vector<Flower> *GetValues_____.() const {
01008         return values_;
01009     }
01010
01011     Item *GetNext_____.() const {
01012         return next_;
01013     }
01014
01015     const string& GetKey_____.() const {
01016         return key_;
01017     }
01018
01019     const vector<Flower> *GetValues_____.() const {
01020         return values_;
01021     }
01022
01023     Item *GetNext_____.() const {
01024         return next_;
01025     }
01026
01027     const string& GetKey_____.() const {
01028         return key_;
01029     }
01030
01031     const vector<Flower> *GetValues_____.() const {
01032         return values_;
01033     }
01034
01035     Item *GetNext_____.() const {
01036         return next_;
01037     }
01038
01039     const string& GetKey_____.() const {
01040         return key_;
01041     }
01042
01043     const vector<Flower> *GetValues_____.() const {
01044         return values_;
01045     }
01046
01047     Item *GetNext_____.() const {
01048         return next_;
01049     }
01050
01051     const string& GetKey_____.() const {
01052         return key_;
01053     }
01054
01055     const vector<Flower> *GetValues_____.() const {
01056         return values_;
01057     }
01058
01059     Item *GetNext_____.() const {
01060         return next_;
01061     }
01062
01063     const string& GetKey_____.() const {
01064         return key_;
01065     }
01066
01067     const vector<Flower> *GetValues_____.() const {
01068         return values_;
01069     }
01070
01071     Item *GetNext_____.() const {
01072         return next_;
01073     }
01074
01075     const string& GetKey_____.() const {
01076         return key_;
01077     }
01078
01079     const vector<Flower> *GetValues_____.() const {
01080         return values_;
01081     }
01082
01083     Item *GetNext_____.() const {
01084         return next_;
01085     }
01086
01087     const string& GetKey_____.() const {
01088         return key_;
01089     }
01090
01091     const vector<Flower> *GetValues_____.() const {
01092         return values_;
01093     }
01094
01095     Item *GetNext_____.() const {
01096         return next_;
01097     }
01098
01099     const string& GetKey_____.() const {
01100         return key_;
01101     }
01102
01103     const vector<Flower> *GetValues_____.() const {
01104         return values_;
01105     }
01106
01107     Item *GetNext_____.() const {
01108         return next_;
01109     }
01110
01111     const string& GetKey_____.() const {
01112         return key_;
01113     }
01114
01115     const vector<Flower> *GetValues_____.() const {
01116         return values_;
01117     }
01118
01119     Item *GetNext_____.() const {
01120         return next_;
01121     }
01122
01123     const string& GetKey_____.() const {
01124         return key_;
01125     }
01126
01127     const vector<Flower> *GetValues_____.() const {
01128         return values_;
01129     }
01130
01131     Item *GetNext_____.() const {
01132         return next_;
01133     }
01134
01135     const string& GetKey_____.() const {
01136         return key_;
01137     }
01138
01139     const vector<Flower> *GetValues_____.() const {
01140         return values_;
01141     }
01142
01143     Item *GetNext_____.() const {
01144         return next_;
01145     }
01146
01147     const string& GetKey_____.() const {
01148         return key_;
01149     }
01150
01151     const vector<Flower> *GetValues_____.() const {
01152         return values_;
01153     }
01154
01155     Item *GetNext_____.() const {
01156         return next_;
01157     }
01158
01159     const string& GetKey_____.() const {
01160         return key_;
01161     }
01162
01163     const vector<Flower> *GetValues_____.() const {
01164         return values_;
01
```

```

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 void Insert(const T& value) {
00100     if (!root_) {
00101         root_ = new RBNode<T>(value);
00102         root_>color_ = BLACK;
00103         return;
00104     }
00105
00106     RBNode<T> *target = root_;
00107     RBNode<T> *parent = nullptr;
00108
00109     while(target) {
00110         parent = target;
00111         if (value < target->values_[0]) {
00112             target = target->left_;
00113         } else if (value > target->values_[0]){
00114             target = target->right_;
00115         } else {
00116             target->values_.push_back(value);
00117             return;
00118         }
00119     }
00120
00121     target = new RBNode<T>(value);
00122     target->parent_ = parent;
00123
00124     if (value < parent->values_[0]) {
00125         parent->left_ = target;
00126     } else {
00127         parent->right_ = target;
00128     }
00129
00130     Balance(target);
00131 }
00132
00133 // /// @brief Search for all nodes containing a given value.
00134 // /// @param value Reference to the value to search for.
00135 // /// @return Pointer to the node containing the value, or nullptr if not found.
00136 // RBNode<T>* Search(const T& value) {
00137 //     return SupSearch(root_, value);
00138 // }
00139
00140 RBNode<T>* SearchAll(const T& value) {
00141     if (root_) {
00142         RBNode<T> *cur = root_;
00143
00144         while (cur) {
00145             if (cur->values_[0] == value) {
00146                 return cur;
00147             }
00148
00149             if (value < cur->values_[0]) {
00150                 cur = cur->left_;
00151             } else {
00152                 cur = cur->right_;
00153             }
00154         }
00155     }
00156
00157     return nullptr;
00158 }
00159
00160 void PrintTree() {
00161     SupportPrint(root_);
00162 }
00163
00164 private:
00165     RBNode<T> *root_;
00166
00167 private:
00168     // /**
00169     // * \brief Helper function to search for a value starting at a given node.
00170     // *
00171     // * Traverses the subtree like a standard BST search: if the current node's value
00172     // * matches, returns it; if the value is less, goes left; otherwise, goes right.
00173     // *
00174     // * \param node Pointer to the current node from which to start searching.
00175     // * \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](#)