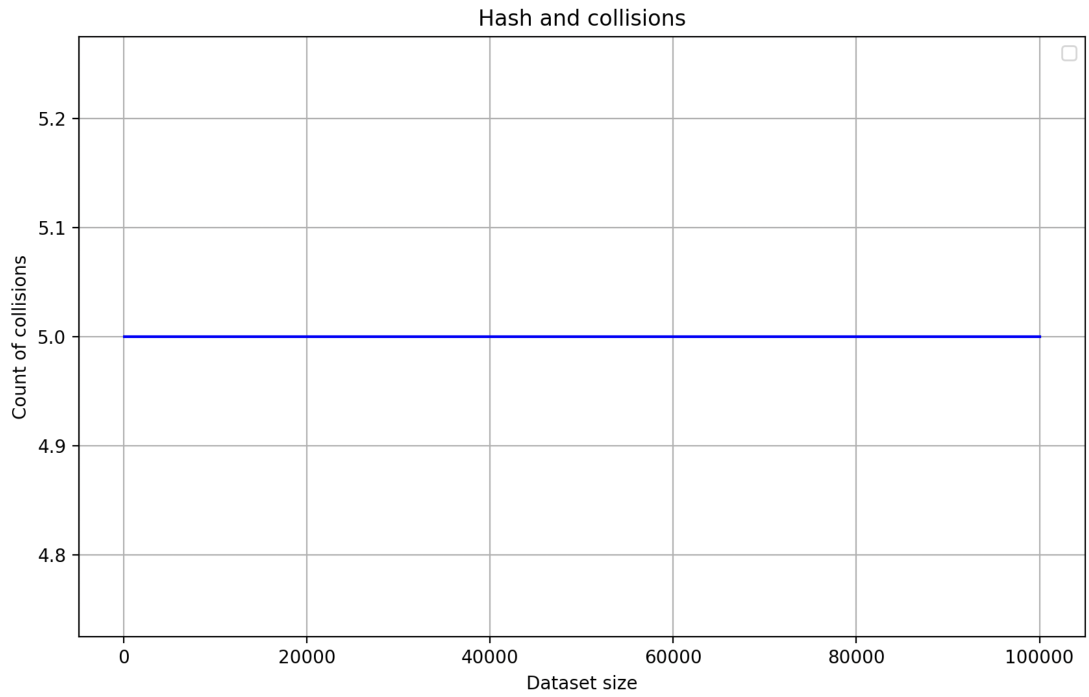
Отчёт

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:

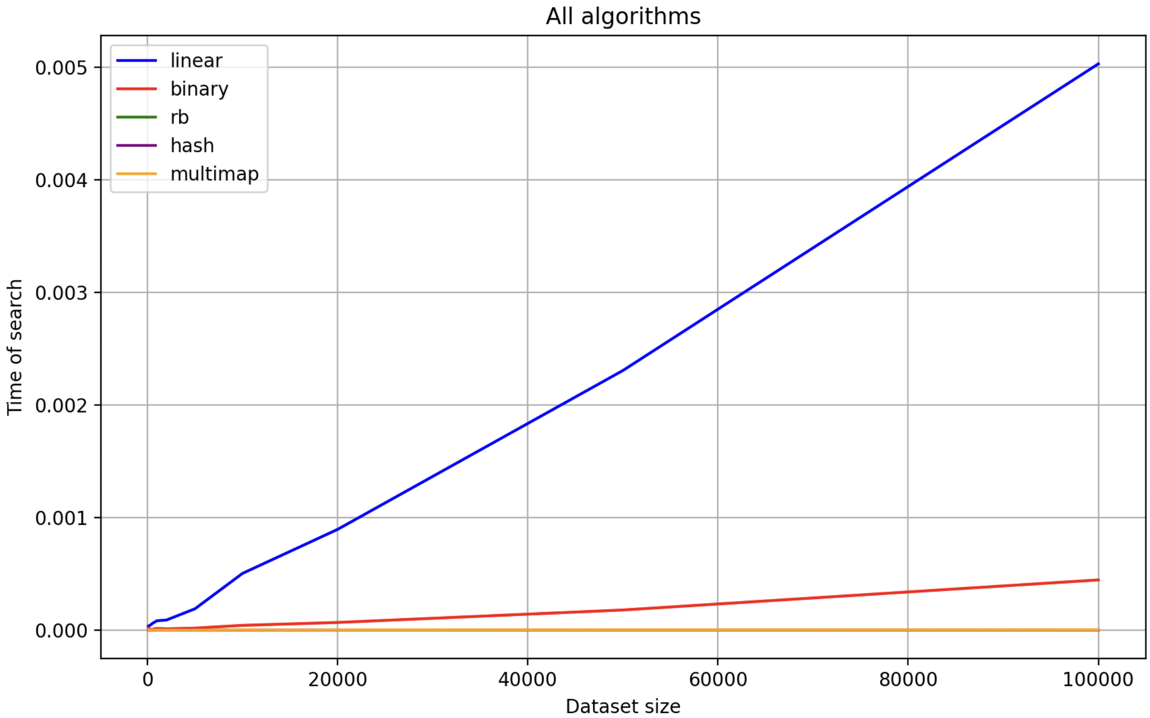
Наиболее эффективен и самый быстрый — поиск по собственной **хеш-таблице** (O(1)).

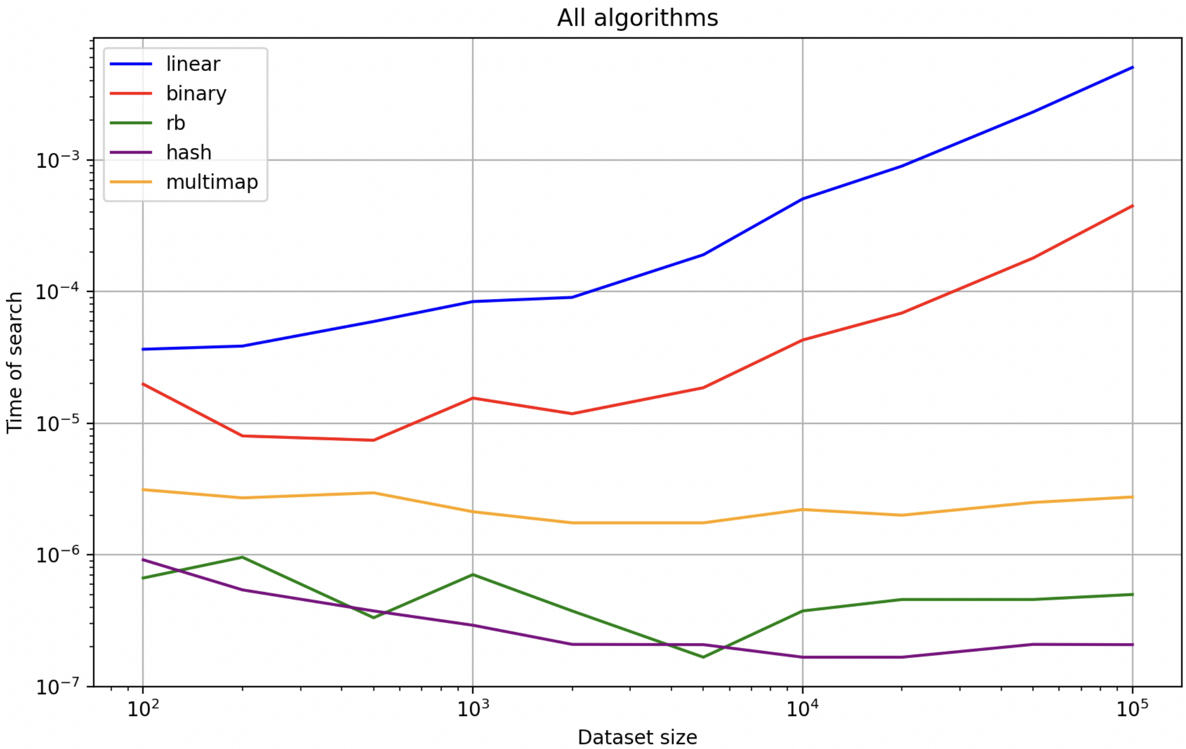
Почти не уступает ему по скорости **RB-дерево (O(log n)).**

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(log2(n))).

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


All algorithms
linear
binary
rb
hash
multimap
Time of search
0.005
0.004
0.003
0.002
0.001
0.000
0
20000
40000
60000
80000
100000
Dataset size


All algorithms
linear
binary
rb
hash
multimap
Time of search
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{2}
10^{3}
10^{4}
10^{5}
Dataset size

Laboratory work №2

Generated by Doxygen 1.13.2

Chapter 1

Topic Index

1.1 Topics

Here is a list of all topics with brief descriptions:

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

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

value	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

value	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

value	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

value	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

value	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

other	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

data	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

key	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

T	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

T	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

other	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

T	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

T	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.

```

#ifndef BINARY_TREE_H
#define BINARY_TREE_H

#include <vector>
#include <iostream>
using namespace std;

template <typename T>
class Node {
public:
    Node() = default;
    Node(T value) { value_ = value; }

public:
    T value_;
    Node *left_ = nullptr;
    Node *right_ = nullptr;
};

template <typename T>
class Tree {
public:

```

```
    Tree() { root_ = nullptr; }
    Tree(T value) { root_ = new Node<T>(value); }
    ~Tree() { DeleteTree(root_); }


    void Insert(const T& value) {
        if (!root_) {
            root_ = new Node<T>(value);
            return;
        }

        Node<T> *cur = root_;
        while (true) {
            if (value < cur->value_) {
                if (cur->left_ == nullptr) {
                    cur->left_ = new Node<T>(value);
                    break;
                }
                cur = cur->left_;
            } else {
                if (cur->right_ == nullptr) {
                    cur->right_ = new Node<T>(value);
                    break;
                }
                cur = cur->right_;
            }
        }
    }

    Node<T>* Search(const T& value) const { return SupportSearch(root_, value); }

    vector<Node<T>*> SearchAll(const T& value) {
        vector<Node<T>*> res;
        Node<T> *tmp = SupportSearch(root_, value);

        while (tmp) {
            res.push_back(tmp);
            tmp = SupportSearch(tmp->right_, value);
        }

        return res;
    }

    void PrintTree() { SupportPrint(root_); }

private:
    Node<T> *root_ = nullptr;

private:

    void SupportPrint(Node<T> *root) {
        if (root) {
            cout « root->value_ « endl;
            SupportPrint(root->left_);
            SupportPrint(root->right_);
        }
    }

    Node<T>* SupportSearch(Node<T>* root, const T& value) {
        Node<T> *cur = root;

        while (cur) {
            if (cur->value_ == value) { return cur; }

            if (value < cur->value_) {
                cur = cur->left_;
            } else {
                cur = cur->right_;
            }
        }

        return nullptr;
    }

    void DeleteTree(Node<T> *root) {
        if (root) {
            DeleteTree(root->left_);
            DeleteTree(root->right_);
            delete root;
        }
    }
};

#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.

```
#ifndef FLOWER_H
#define FLOWER_H

#include <string>
#include <vector>

using namespace std;

class Flower {
public:
    Flower() = default;
    Flower(string name, string color, string smell, vector<string> regions);
    Flower(const Flower& other) = default;
    Flower(Flower&& other) = default;
    ~Flower() = default;

    string GetName() const { return name_; }
    string GetColor() const { return color_; }
    string GetSmell() const { return smell_; }
    vector<string> GetRegions() const { return regions_; }

    void SetName(string name) { name_ = name; }
    void SetColor(string color) { color_ = color; }
    void SetSmell(string smell) { smell_ = smell; }
    void SetRegions(vector<string> regions) { regions_ = regions; }

    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);

    bool EqFlowers(const Flower& other) const;

private:
    string name_;
    string color_;
    string smell_;
    vector<string> regions_;
};

#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

key	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.

```
#ifndef HASH_H
#define HASH_H

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

#define SIZE 14
using namespace std;

unsigned int hashFunc_rs(string key) {
    unsigned int a = 63689;
    unsigned int b = 378551;
    unsigned int hash = 0;
    unsigned int i = 0;

    int len = key.length();

    for (i = 0; i < len; ++i) {
        hash = hash * a + (unsigned char)key[i];
        a = a * b;
    }

    return (hash % SIZE);
}

class Item {
public:
    string key_;
    vector<Flower> *values_;
    Item *next_;

public:
    Item() {
        key_ = "";
        values_ = new vector<Flower>();
        next_ = nullptr;
    }

    Item(const string& key, const Flower& value) {
        key_ = key;
        values_ = new vector<Flower>();
        values_->push_back(value);
        next_ = nullptr;
    }
};

class HashTable {
public:
    HashTable(const vector<Flower>& data) {
        NullTable();

        count = data.size();

        for (size_t i = 0; i < count; ++i) {
            string key = data[i].GetName();
            Flower value = data[i];
            unsigned int hash = hashFunc_rs(key);

            if (!items_[hash]) {
                items_[hash] = new Item(key, value);
                unq_count += 1;
            } else {
                Item *where = items_[hash];
                int is_collis = 1;

                while (true) {
                    if (key == where->key_) {
                        where->values_->push_back(value);
                        is_collis = 0;
                        break;
                    } else {
                        if (where->next_) {
                            where = where->next_;
                        } else {
                            break;
                        }
                    }
                }
```

```
                if (is_collis) {
                    collisions += 1;
                    unq_count += 1;
                    where->next_ = new Item(key, value);
                }
            }
        }
    }

    ~HashTable() {
        for (int i = 0; i < SIZE; ++i) {
            SupportDelete(items_[i]);
        }
    }

    vector<Flower>* Search(const string& key) const {
        vector<Flower> *res = nullptr;

        unsigned int hash = hashFunc_rs(key);
        Item *cur = items_[hash];
        while(cur) {
            if (cur->key_ == key) {
                res = cur->values_;
                break;
            }
            cur = cur->next_;
        }

        return res;
    }

    long long GetCount() { return count; }
    long long GetCountUnq() { return unq_count; }
    long long GetCollisions() { return collisions; }
    Item** GetItems() { return items_; }

    void GetTable() {
        for (int i = 0; i < SIZE; ++i) {
            Item *cur = items_[i];

            cout « i « "   \t";
            if (!cur) {
                cout « "-" « endl;
            } else {
                while (cur) {
                    cout « cur->key_ « "(" « cur->values_->size() « ")" « "   \t";
                    cur = cur->next_;
                }
                cout « endl;
            }
        }

        cout « endl « "Count: " « count « endl « "Collisions: " « collisions;
    }

private:
    Item *items_[SIZE];
    long long count;
    long long unq_count = 0;
    long long collisions = 0;

private:
    void NullTable() {
        for (int i = 0; i < SIZE; ++i) {
            items_[i] = nullptr;
        }
    }

    void SupportDelete(Item *cur) {
        while (cur) {
            Item *next_node = cur->next_;
            delete cur->values_;
            delete cur;
            cur = next_node;
        }
    }
};

#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

filename	Path to the input CSV file.

Exceptions

runtime_error	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

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

Exceptions

std::runtime_error	If any output file cannot be opened for writing.

6.8 io.h

Go to the documentation of this file.

```
#ifndef IO_H
#define IO_H

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

using namespace std;

vector<Flower> parserCSV(string filename);

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

#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

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

a	Pointer to the array of elements of type T.
size	Total number of elements in the array.
b	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.

```
#ifndef LINEAR_H
#define LINEAR_H

#include <vector>
using namespace std;

template<class T>
int linearSearch(T a[], long start, long size, T b) {
    for (long i = start; i < size; ++i) {
        if (a[i] == b) {
            return i;
        }
    }
    return -1;
}

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

    int i = linearSearch(a, 0, size, b);
    while (i != -1) {
        res.push_back(i);
        i = linearSearch(a, i+1, size, b);
    }

    return res;
}

#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.

```
#ifndef RB_TREE_H
#define RB_TREE_H

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

using namespace std;

typedef enum { RED, BLACK } Color;


template <typename T>
class RBNode {
public:
    vector<T> values_;
    Color color_;

    RBNode *left_, *right_, *parent_;

public:
    RBNode() {
        color_ = RED;
        left_ = nullptr;
        right_ = nullptr;
        parent_ = nullptr;
        values_ = T();
    }

    RBNode(const T value) {
        values_.push_back(value);
        color_ = RED;
        left_ = nullptr;
        right_ = nullptr;
        parent_ = nullptr;
    }

    RBNode& operator=(const RBNode& other) {
        values_ = other.values_;
        color_ = other.color_;
        left_ = other.left_;
        right_ = other.right_;
        parent_ = other.parent_;

        return *this;
    }
};

template <typename T>
class RBTree {
public:
```

```

    RBTree() { root_ = nullptr; }
    RBTree(T value) {
        root_ = new RBNode<T>(value);
        root_->color_ = BLACK;
    }
    ~RBTree() { DelTree(root_); }


    void Insert(const T& value) {
        if (!root_) {
            root_ = new RBNode<T>(value);
            root_->color_ = BLACK;
            return;
        }

        RBNode<T> *target = root_;
        RBNode<T> *parent = nullptr;

        while(target) {
            parent = target;
            if (value < target->values_[0]) {
                target = target->left_;
            } else if (value > target->values_[0]){
                target = target->right_;
            } else {
                target->values_.push_back(value);
                return;
            }
        }

        target = new RBNode<T>(value);
        target->parent_ = parent;

        if (value < parent->values_[0]) {
            parent->left_ = target;
        } else {
            parent->right_ = target;
        }

        Balance(target);
    }

    // /// @brief  Search for all nodes containing a given value.
    // /// @param value Reference to the value to search for.
    // /// @return Pointer to the node containing the value, or nullptr if not found.
    // RBNode<T>* Search(const T& value) {
    //     return SupSearch(root_, value);
    // }

    RBNode<T>* SearchAll(const T& value) {
        if (root_) {
            RBNode<T> *cur = root_;

            while (cur) {
                if (cur->values_[0] == value) {
                    return cur;
                }

                if (value < cur->values_[0]) {
                    cur = cur->left_;
                } else {
                    cur = cur->right_;
                }
            }
        }

        return nullptr;
    }

    void PrintTree() {
        SupportPrint(root_);
    }

private:
    RBNode<T> *root_;

private:

    // /**
    //  * \brief Helper function to search for a value starting at a given node.
    //  *
    //  * Traverses the subtree like a standard BST search: if the current node's value
    //  * matches, returns it; if the value is less, goes left; otherwise, goes right.
    //  *
    //  * \param node  Pointer to the current node from which to start searching.
    //  * \param value Reference to the value to search for.
```

```
    //  * \return Pointer to the node containing the value, or nullptr if not found.
    //  */
    // RBNode<T>* SupSearch(RBNode<T> *node, const T& value) {
    //     if (node) {
    //         RBNode<T> *cur = node;

    //         while (cur) {
    //             if (cur->values_[0] == value) {
    //                 return cur;
    //             }

    //             if (value < cur->value_) {
    //                 cur = cur->left_;
    //             } else {
    //                 cur = cur->right_;
    //             }
    //         }
    //     }

    //     return nullptr;
    // }

    void SupportPrint(RBNode<T> *root) {
        if (root) {
            cout « root->values_ « " " « root->color_ « endl;

            SupportPrint(root->left_);
            SupportPrint(root->right_);
        }
    }

    void DelTree(RBNode<T> *root) {
        if (root) {
            DelTree(root->left_);
            DelTree(root->right_);
            delete root;
        }
    }

    void Balance(RBNode<T> *node) {
        RBNode<T> *dad = node->parent_;

        while (dad && dad->color_ == RED) {
            RBNode<T> *grand = dad->parent_;
            RBNode<T> *uncle;

            if (!grand) {
                return;
            } else if (dad == grand->left_) {
                uncle = grand->right_;

                // 1
                if (uncle && uncle->color_ == RED) {
                    dad->color_ = BLACK;
                    uncle->color_ = BLACK;
                    grand->color_ = RED;

                    node = grand;
                    dad = node->parent_;
                    if (dad) { grand = dad->parent_; }

                    continue;
                } else { // 2
                    if (dad->right_ == node) {
                        LeftRotate(node, dad, grand);
                        node = dad;
                        dad  = node->parent_;
                    }
                    dad->color_ = BLACK;
                    grand->color_  = RED;
                    RightRotate(dad, grand, grand->parent_);
                    break;
                }
            } else {
                uncle = grand->left_;

                if (uncle && uncle->color_ == RED) {
                    dad->color_ = BLACK;
                    uncle->color_ = BLACK;
                    grand->color_ = RED;

                    node = grand;
                    dad = node->parent_;
                    if (dad) { grand = dad->parent_; }

                    continue;
                } else {
```

```
                    if (dad->left_ == node) {
                        RightRotate(node, dad, grand);
                        node = dad;
                        dad  = node->parent_;
                    }
                    dad->color_ = BLACK;
                    grand->color_  = RED;
                    LeftRotate(dad, grand, grand->parent_);
                    break;
                }
            }
        }

        root_->color_ = BLACK;
    }

    void LeftRotate(RBNode<T> *child, RBNode<T> *dad, RBNode<T> *grand) {
        RBNode<T> *grandson = child->left_;

        dad->right_ = grandson;
        if (grandson) { grandson->parent_ = dad; }

        child->left_ = dad;
        dad->parent_ = child;

        child->parent_ = grand;
        if (!grand) {
            root_ = child;
        } else if (grand->left_ == dad) {
            grand->left_ = child;
        } else {
            grand->right_ = child;
        }
    }

    void RightRotate(RBNode<T> *child, RBNode<T> *dad, RBNode<T> *grand) {
        RBNode<T> *grandson = child->right_;

        dad->left_ = grandson;
        if (grandson) { grandson->parent_ = dad; }

        dad->parent_ = child;
        child->right_ = dad;

        child->parent_ = grand;
        if (!grand) {
            root_ = child;
        } else if (grand->right_ == dad) {
            grand->right_ = child;
        } else {
            grand->left_ = child;
        }
    }
};

#endif
```

Index