Exercise 2

Implementing Hash Table from scratch

1 Description

Hash Table is a popular data structure in programming, used for efficient data storage and retrieval. In this Exercise 2, you are required to implement a hash table with 5 different collision handling: Linear Probing, Quadratic Probing, Chaining using Linked Listed, Chaining using AVL Tree, and Double Hashing. Just like Exercise 1, please use `struct` and `template`.

1.1 Linear Probing

Implement a hash table using **linear probing** with the following variables:

- `vector<hashNode*> table`: This is an array containing the data for the hash table, where each item is a `hashNode` storing a key-value pair. For example:

  ```
  struct hashNode
  {
      K key;
      V value;
  };
  ```

- `int capacity`: size of the hash table.

and functions:

- `void init(unsigned int hashSize)`: initialize an empty hash table with the given size.
- `void release()`: free all dynamically allocated memory in the hash table.
- `hashFunctions`: hash functions to compute the index for a given key.
- `void add(K key, V value)`: add a new element. If the key existed, update the old value.
- `V* searchValue(K key)`: search an element in the table. If not existed, return `NULL`.
- `void removeKey(K key)`: remove an element from the hash table.

And other variables, functions if necessary.

Please note that the program should be organized into (at least) 3 separate files, including `hash.h` (containing the declaration of the hash table), `hash.cpp` (containing the implementation for the hash table), and `main.cpp` (to use the hash table as a library).
Figure 1 and Figure 2 below are examples of `hash.h` and `hash.cpp` files.

```
#pragma once
#include <string>
#include <vector>
using namespace std;

template<typename K, typename V>
struct hashTable
{
    // Variables (Attributes)
    struct hashNode
    {
        K key;
        V value;
    };

    int capacity;
    vector<hashNode*> table;

    // Functions (Methods)
    void init(unsigned int hashSize);
    void release();
    unsigned int hashFunction(int key);    // If key is int
    unsigned int hashFunction(string key); // If key is string
    void add(K key, V value);
    V* searchValue(K key);
    void removeKey(K key);
};

#include "hash.cpp"
```

Figure 1: Example of header file for hash using linear probing.

```
#include "hash.h"

template<typename K, typename V>
void hashTable<K, V>::init(unsigned int hashSize)
{
    capacity = hashSize;
    table = vector<hashNode*>(hashSize, NULL);
}
```

Figure 2: Example of implementation file for hash using linear probing.

In `main.cpp`, try using hash table as a library. You can initialize the hash table with a small size, then perform add, search, and remove operations on it before releasing the memory and ending the program. **No menu is required.**
Please **capture the results** and include it in the **report**.

1.2 Quadratic Probing

Implement a hash table using **quadratic probing** with the same variables and functions as in Linear Probing.

1.3 Chaining using Linked List

Implement a hash table using **chaining** with the same variables and functions as in Linear Probing. However, the `hashNode` will need to be modified a bit to implement a **linked list**, as shown below:

```
struct hashNode
{
    K key;
    V value;
    hashNode* next; // Add this line
};
```

Also, you will need to implement some additional functions to work with linked lists.

1.4 Chaining using AVL Tree

Implement a hash table using **chaining** with the same variables and functions as in Linear Probing. However, the `hashNode` will need to be modified a bit to implement a **AVL Tree**, as shown below:

```
struct hashNode
{
    K key;
    V value;
    hashNode* left;  // Add this line
    hashNode* right; // And add this line, too!
};
```

Also, you will need to implement some additional functions to work with AVL Trees.

1.5 Double Hashing

Implement a hash table using **double hashing** with the same variables as in Linear Probing, and the same functions but adding a second hash function.

2 Experiment

In this section, you are required to measure and compare the **query time** of the implemented hash table versions above with linear search algorithm.

Dataset description

The input is read from the **provided** text file `books.txt`, which has the content as Figure 3:

```
bookTitle|authorName
Classical Mythology|Mark P. O. Morford
Clara Callan|Richard Bruce Wright
Decision in Normandy|Carlo D'Este
Flu: The Story of the Great Influenza Pandemic of 1918 and the Search for the Virus That Caused It|Gina Bari Kolata
The Mummies of Urumchi|E. J. W. Barber
The Kitchen God's Wife|Amy Tan
What If?: The World's Foremost Military Historians Imagine What Might Have Been|Robert Cowley
PLEADING GUILTY|Scott Turow
Under the Black Flag: The Romance and the Reality of Life Among the Pirates|David Cordingly
Where You'll Find Me: And Other Stories|Ann Beattie
Nights Below Station Street|David Adams Richards
Hitler's Secret Bankers: The Myth of Swiss Neutrality During the Holocaust|Adam Lebor
The Middle Stories|Sheila Heti
Jane Doe|R. J. Kaiser
```

Figure 3: Example of the book dataset. (Data source URL: (Kaggle).

where:

- The first line provides the included information fields.
- For the next lines, each line contains the book title and the author name, separated by a vertical bar character (|).

Requirements

Read information about books and store it in a hash table with a size of 300,000 items, using the book title as the key and the author name as the value.
Additionally, store the dataset in a normal array to apply linear search algorithm.
Analyze the **theoretical time complexity** and compare it with the **actual execution time** for searching the author name of the book title that:

- Located at the beginning of the dataset.
- Located in the middle of the dataset.
- Located at the end of the dataset.
- Not present in the dataset.

Please record the results and include them in the report along with your comments.

3 Submission

Source code folder and report file must be submitted in the form of a compressed file (.zip, .rar) and named according to your Student ID. For example:

```
StudentID.zip
├── Source
└── Report.pdf
```

3.1 Source code

The source code folder contains the programming files. For example:

```
Source
├── Linear Probing
│   ├── hash.h
│   ├── hash.cpp
│   ├── main.cpp
│   └── experiment.cpp
├── Quadratic Probing
├── Chaining Linked List
├── Chaining AVL
└── Double Hashing
```

3.2 Report

The report must include the following information:

- Student information (Student ID, Full name, etc.).
- Self-evaluation.
- How you implemented the requirements.
- Detailed experiments, results and your comments.
- Exercise feedback (what you learned, what was difficult).
- The report needs to be well-formatted and exported to PDF file. If there are figures cut off by the page break, etc., points will be deducted.
- References (if any).

4 Assessment

No.	Details	Score
1	Linear Probing	15%
2	Quadratic Probing	15%
3	Chaining using Linked List	15%
4	Chaining using AVL Tree	15%
5	Double Hashing	15%
6	Experiments	10%
7	Report	15%
	Total	**100%**

5 Notices

Please pay attention to the following notices:

- This is an **INDIVIDUAL** assignment.
- Duration: about 2 weeks.
- Any plagiarism, any tricks, or any lie will have a 0 point for the course grade.

6 Hints

To hash a string, you can use polynomial rolling hash function. The formula is:

$$hash(s) = \left(\sum_{i=0}^{n-1} (s[i] \times p^i)\right) \mod m$$

which:

- s: The key as a string of length n.
- $s[i]$: ASCII code of the character at position i from s.
- $p = 31$.
- $m = 10^9 + 9$.

The end.