VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CIY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Data Structures and Algorithms

Exercise 2 Report

Implementing Hash Table from scratch

Student Name	Student ID
Nguyen Thi Khanh Linh	23127082

Lecturers:

Bui Duy Dang

Truong Tan Khoa

Nguyen Thanh Tinh

Submission Date : August 16 2024

Contents

1 Introduction

Importance of Hash Table: Hash tables are a fundamental data structure in computer science, widely used for efficient data storage and retrieval. Understanding different collision handling methods is crucial for developing robust and efficient hash tables.
Purpose of the Exercise: In this exercise, the focus is on implementing a Hash Table, a fundamental data structure widely used for efficient data storage and retrieval. The task involves exploring five different collision handling techniques: Linear Probing, Quadratic Probing, Chaining using Linked Lists, Chaining using AVL Trees, and Double Hashing.

2 Student Information

Student ID: 23127082
Full Name: Nguyen Thi Khanh Linh
Class: 23CLC09

3 Linear Probing

3.1 Implementation

A `hashTable` structure consists of:

- 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.
  - ```
    struct hashNode
    {
    K key;
    V value;
    };
    ```
  - `int capacity`: size of the hash table
- 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.
    - To hash a string, I used polynomial rolling hash function. The formula is:

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

      - s: The key as a string of length n.
      - $s[i]$: ASCII code of the character at position i from s.
    ```

  - $p = 31$.
  - $m = 10^9 + 9$.
- `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.
- `void displayTable()`: display the hash table (only display the index storing data).

- Hash function: $h(k, i) = (h'(k) + i) \mod capacity$. $h'(k)$ is the hash value of key k.

File Structure:

- `hash.h`: Containing the declaration of the hash table.
- `hash.cpp`: Containing the implementation for the hash table.
- `main.cpp`: Using the hash table as a library
- `experiement.cpp`: Measuring and compare the query time of the implemented hash table versions above with linear search algorithm.

3.2 Test with small dataset

- Code for testing:

```cpp
hashTable<string, string> h;
h.init(10);
h.add("one direction", "1d");
h.add("selena gomez", "sg");
h.add("justin bieber", "jb");
h.add("taylor swift", "ts");
h.add("ariana grande", "ag");
h.add("shawn mendes", "sm");
h.add("ed sheeran", "es");
cout << "Initial table" << endl;
h.displayTable();
cout << "-----" << endl;
h.removeKey("selena gomez");
cout << "Table after removing 'selena gomez'" << endl;
h.displayTable();
cout << "-----" << endl;
string* search = h.searchValue("taylor swift");
if (search != nullptr) {
  cout << "Value found: " << *search << endl;
} else {
  cout << "Value not found" << endl;
}
h.release();

```

Figure 1: The code to test the functions

- Result:

```
Initial table
Index: 0 Key: shawn mendes Value: sm
Index: 1 Key: ed sheeran Value: es
Index: 4 Key: taylor swift Value: ts
Index: 6 Key: one direction Value: 1d
Index: 7 Key: selena gomez Value: sg
Index: 8 Key: justin bieber Value: jb
Index: 9 Key: ariana grande Value: ag
-----
Table after removing 'selena gomez'
Index: 0 Key: shawn mendes Value: sm
Index: 1 Key: ed sheeran Value: es
Index: 4 Key: taylor swift Value: ts
Index: 6 Key: one direction Value: 1d
Index: 8 Key: justin bieber Value: jb
Index: 9 Key: ariana grande Value: ag
-----
Value found: ts
```

Figure 2: Result displayed

3.3 Experiment

- An approximately 300000 elements dataset was used for experimenting with this probing method. I compare the time to search for the beginning, middle, end and unfound data pieces in this dataset between linear search and using Hash Table with Linear Probing.

- Theoretical time complexity:

Function	**Time Complexity**
Hash Table (Linear Probing)	$O(n)$
Linear Search	$O(n)$

Table 1: Summary of Time Complexities

- Experiment:

```
Time compare:

Data located at the beginning of the dataset:
Hash Table (Linear Probing): 5 microseconds
Linear Search: 0 microseconds

Data located at the middle of the dataset:
Hash Table (Linear Probing): 1 microseconds
Linear Search: 846 microseconds

Data located at the end of the dataset:
Hash Table (Linear Probing): 8 microseconds
Linear Search: 2838 microseconds

Data does not exist in the dataset
Hash Table (Linear Probing):  1 microseconds
Search: 1789 microseconds
```

Figure 3: Result displayed

- It is clear that overall, the actual execution time of searching using Hash Table (Linear Probing) is faster than using Linear Search except for the data piece at the beginning of the data set. This is because Linear Search checks each element sequentially, so if the target element is one of the first few in the list, it can be found very quickly. Despite this, as the size of the data set grows, the benefits of using a Hash Table with Linear Probing become more pronounced, making it a far more efficient choice in scenarios involving large-scale data or frequent searches.

4 Quadratic Probing

4.1 Implementation

- The `hashTable` structure is almost the same with Linear Probing. Except that, I added a `rehash()` function.
  - `void rehash()`: Double the capacity and import all the keys and values into the Hash Table again.
- File Structure: Same with Linear Probing.
- Hash function: $h(k, i) = (h'(k) + i^2) \mod capacity$. $h'(k)$ is the hash value of key k.

4.2 Test with small dataset

- Code for testing: Similar to Linear Probing.
- Result:

```
Index: 1 Key: ed sheeran Value: es
Index: 2 Key: shawn mendes Value: sm
Index: 4 Key: taylor swift Value: ts
Index: 6 Key: one direction Value: 1d
Index: 7 Key: selena gomez Value: sg
Index: 8 Key: justin bieber Value: jb
Index: 9 Key: ariana grande Value: ag
-----
Index: 1 Key: ed sheeran Value: es
Index: 2 Key: shawn mendes Value: sm
Index: 4 Key: taylor swift Value: ts
Index: 6 Key: one direction Value: 1d
Index: 8 Key: justin bieber Value: jb
Index: 9 Key: ariana grande Value: ag
Value found: ts
```

Figure 4: Result displayed

4.3 Experiment

- An approximately 300000 elements dataset was used for experimenting with this probing method. I compare the time to search for the beginning, middle, end and unfound data pieces in this dataset between Linear Search and using Hash Table with Quadratic Probing.
- Theoretical time complexity:

Function	**Time Complexity**
Hash Table (Quadratic Probing)	$O(n)$
Linear Search	$O(n)$

Table 2: Summary of Time Complexities

- Experiment:

```
Time compare:

Data located at the beginning of the dataset:
Hash Table (Quadratic Probing): 9 microseconds
Linear Search: 0 microseconds

Data located at the middle of the dataset:
Hash Table (Quadratic Probing):2 microseconds
Linear Search: 1206 microseconds

Data located at the end of the dataset:
Hash Table (Quadratic Probing): 12 microseconds
Linear Search: 4638 microseconds

Data does not exist in the dataset
Hash Table (Quadratic Probing): 5 microseconds
Search: 20912 microseconds
```

Figure 5: Result displayed

- It is clear that overall, the actual execution time of searching using Hash Table (Quadratic Probing) is faster than using Linear Search except for the data piece at the beginning of the data set. The explanation is similar to Linear Probing.

5 Chaining using Linked List

5.1 Implementation

- The `hashTable` structure is almost the same with Linear Probing. However, the struct hashNode is different to make the code run properly.

  –
  ```
  struct hashNode
  {
  K key;
  V value;
  hashNode* next = nullptr;
  };
  ```

- File Structure: Same with Linear Probing.

- Hash function: $h(k, i) = (h'(k) + i) \mod capacity$. $h'(k)$ is the hash value of key k. Each element of the hash table is a linked list. When a collision occurs, the new element is inserted at the end of the linked list.

5.2 Test with small dataset

- Code for testing: Similar to Linear Probing.

- Result:

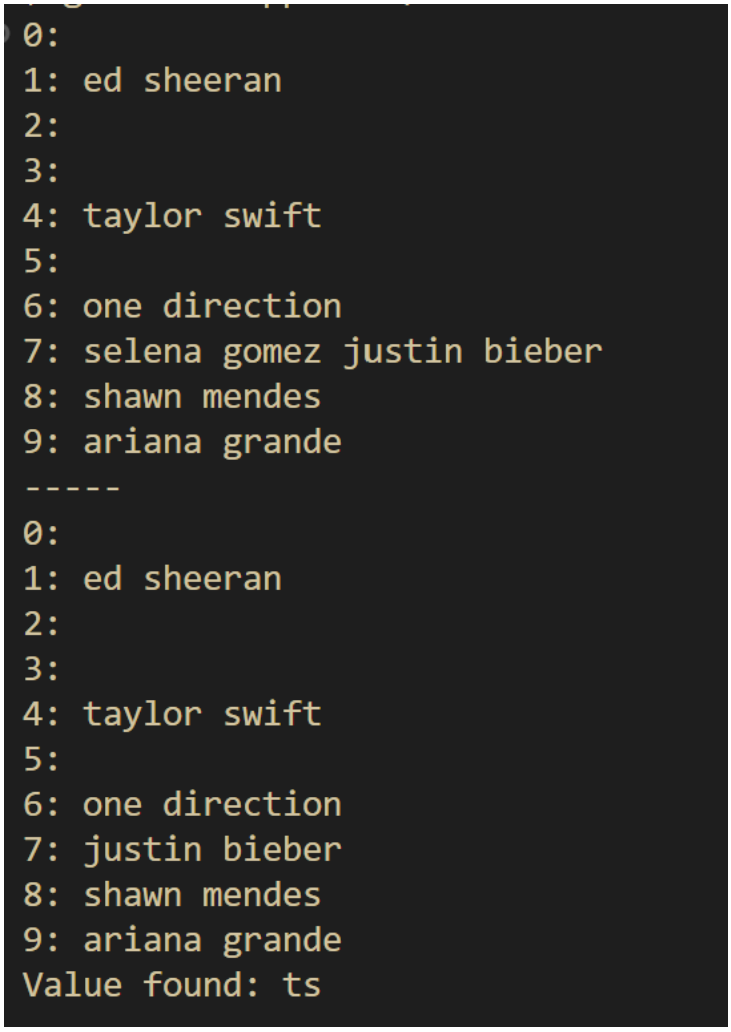

Figure 6: Result displayed

5.3 Experiment

- An approximately 300000 elements dataset was used for experimenting with this probing method. I compare the time to search for the beginning, middle, end and unfound data pieces in this dataset between Linear Search and using Hash Table with Chaining using Linked List.
- Theoretical time complexity:

Function	Time Complexity
Hash Table (Chaining Linked List)	$O(n)$
Linear Search	$O(n)$

Table 3: Summary of Time Complexities

- Experiment:

```
Time compare:

Data located at the beginning of the dataset:
Hash Table (Chaining Linked List): 7 microseconds
Linear Search: 0 microseconds

Data located at the middle of the dataset:
Hash Table (Chaining Linked List): 4 microseconds
Linear Search: 1456 microseconds

Data located at the end of the dataset:
Hash Table (Chaining Linked List): 12 microseconds
Linear Search: 4280 microseconds

Data does not exist in the dataset
Hash Table (Chaining Linked List): 3 microseconds
Search: 2752 microseconds
```

Figure 7: Result displayed

- It is clear that overall, the actual execution time of searching using Hash Table (Chaining using Linked List) is faster than using Linear Search except for the data piece at the beginning of the data set. The explanation is similar to Linear Probing.

6 Chaining using AVL

6.1 Implementation

- The `hashTable` structure is almost the same with Linear Probing. However, the struct hashNode is different and a few more functions were added to make the code run properly.
  - Change in struct:
    - ```
      struct hashNode
      {
      K key;
      V value;
      hashNode* left;
      hashNode* right;
      };
      ```
  - Added functions:
    - `hashNode* rightRotate(hashNode* y)`: right rotate the imbalanced tree branch.
    - `hashNode* leftRotate(hashNode* y)`: left rotate the imbalanced tree branch.
    - `hashNode* minNode(hashNode* node)`: get node with minimum value.
    - `void AVLInsert(hashNode*& root, hashNode* newNode)`: Insert node into AVL tree.
    - `int getBalance(hashNode* node)`: get balanced factor.
    - `int getHeight(hashNode* node)`: get height of a tree.
      ```

    - `hashNode* removeAVL(hashNode* root, K key)`: remove AVL tree.
    - `void NLR(hashNode* root)`: travel pre-order.

- File Structure: Same with Linear Probing.
- Hash function: $h(k, i) = (h'(k) + i) \mod capacity$. $h'(k)$ is the hash value of key k. Each element of the hash table is an AVL tree. When a collision occurs, new element is inserted into the AVL tree.

6.2 Test with small dataset

- Code for testing: Similar to Linear Probing.
- Result:

```
0:
1: ed sheeran
2:
3:
4: taylor swift
5:
6: one direction
7: selena gomez justin bieber
8: shawn mendes
9: ariana grande
-----
0:
1: ed sheeran
2:
3:
4: taylor swift
5:
6: one direction
7: justin bieber
8: shawn mendes
9: ariana grande
Value found: ts
```

Figure 8: Result displayed

6.3 Experiment

- An approximately 300000 elements dataset was used for experimenting with this probing method. I compare the time to search for the beginning, middle, end and

unfound data pieces in this dataset between Linear Search and using Hash Table with Chaining using AVL.

- Theoretical time complexity:

Function	Time Complexity
Hash Table (Chaining AVL)	$O(logn)$
Linear Search	$O(n)$

Table 4: Summary of Time Complexities

- Experiment:

```
Time compare:

Data located at the beginning of the dataset:
Hash Table (Chaining AVL): 187 microseconds
Linear Search: 0 microseconds

Data located at the middle of the dataset:
Hash Table (Chaining AVL): 1 microseconds
Linear Search: 1291 microseconds

Data located at the end of the dataset:
Hash Table (Chaining AVL): 15 microseconds
Linear Search: 4007 microseconds

Data does not exist in the dataset
Hash Table (Chaining AVL): 2 microseconds
Search: 2192 microseconds
```

Figure 9: Result displayed

- It is clear that overall, the actual execution time of searching using Hash Table (Chaining using AVL) is faster than using Linear Search except for the data piece at the beginning of the data set. The explanation is similar to Linear Probing.

7 Double Hashing

7.1 Implementation

- The `hashTable` structure is almost the same with Linear Probing. Except that, I added a `rehash()` function.
  - `void rehash()`: Double the capacity and import all the keys and values into the Hash Table again.
- File Structure: Same with Linear Probing.

- Hash function: $h(k,i) = (h_1(k) + i \times h_2(k)) \mod capacity$. $h_1(k)$ is the first hash value of key k, and $h_2(k)$ is the second hash value of key k. $h_2(k) = 1 + h(k) \mod (capacity - 1)$.

7.2 Test with small dataset

- Code for testing: Similar to Linear Probing.
- Result:

```
Index: 1 Key: ed sheeran Value: es
Index: 4 Key: taylor swift Value: ts
Index: 5 Key: justin bieber Value: jb
Index: 6 Key: one direction Value: 1d
Index: 7 Key: selena gomez Value: sg
Index: 8 Key: shawn mendes Value: sm
Index: 9 Key: ariana grande Value: ag
-----
Index: 1 Key: ed sheeran Value: es
Index: 4 Key: taylor swift Value: ts
Index: 5 Key: justin bieber Value: jb
Index: 6 Key: one direction Value: 1d
Index: 8 Key: shawn mendes Value: sm
Index: 9 Key: ariana grande Value: ag
Value found: ts
```

Figure 10: Result displayed

7.3 Experiment

- An approximately 300000 elements dataset was used for experimenting with this probing method. I compare the time to search for the beginning, middle, end and unfound data pieces in this dataset between Linear Search and using Hash Table with Double Hashing.
- Theoretical time complexity:

Function	Time Complexity
Hash Table (Double Hashing)	$O(n)$
Linear Search	$O(n)$

Table 5: Summary of Time Complexities

- Experiment:

```
Table is full. Rehashing...

Time compare:

Data located at the beginning of the dataset:
Hash Table (Double Hashing): 7 microseconds
Linear Search: 0 microseconds

Data located at the middle of the dataset:
Hash Table (Double Hashing):  1 microseconds
Linear Search: 1069 microseconds

Data located at the end of the dataset:
Hash Table (Double Hashing): 7 microseconds
Linear Search: 4253 microseconds

Data does not exist in the dataset
Hash Table (Double Hashing):  1 microseconds
Search: 2182 microseconds
```

Figure 11: Result displayed

- It is clear that overall, the actual execution time of searching using Hash Table (Double Hashing) is faster than using Linear Search except for the data piece at the beginning of the data set. The explanation is similar to Linear Probing.

8 Self-Evaluation

8.1 Learning Outcomes

From working on this exercise, I have learned a lot from various aspects:

- I deepened my understanding of Hash Table, and how it can be implemented using 5 different collision handling: Linear Probing, Quadratic Probing, Chaining using Linked Listed, Chaining using AVL Tree, and Double Hashing.
- I praticeed how to measure execution time using `std::chrono` library in C++ [1].
- I had more experience to write a report using LaTeX.

8.2 Challenges

- There is new knowledge I have never approached and need to spend time researching it on many websites.
- During the implementation, managing the different edge cases for each operation, and handling load factor by rehashing is a little challenging.
- Implementing Hash Table (Chaining using AVL) took me a lot of time since AVL is challenging to implement.

9 Exercise Feedback

Overall, the exercise was both challenging and rewarding. It provided a solid foundation in working with Hash Table and allowed me to explore different methods of implementation. The hands-on approach of writing the code and conducting experiments helped reinforce theoretical concepts and enhanced my problem-solving skills.

10 Conclusion

In this report, I have detailed the process of implementing Hash Table using five collision handling techniques: Linear Probing, Quadratic Probing, Chaining with Linked Lists, Chaining with AVL Trees, and Double Hashing. Each method showcased different strengths in managing collisions, with some better suited for specific scenarios. I also compared the actual execution time between using Hash Table and Linear Search which highlighted the efficiency of Hash Table.

11 References

[1] **Measure execution time of a function in C++ library documentation** — Last Updated : 03 May, 2023 by sayan mahapatra.