CPTS 223 Advanced Data Structure C/C++ Fall 2024

PA3: Hash Table

1 Learning Objectives

At the conclusion of this programming assignment, participants should be able to:

- Develop a C++ program that applies hash table data structure;
- Design and implement a hash table with separate chaining and linear probing to handle the collision.

2 Prerequisite

Before starting this programming assignment, participants should be able to:

- Apply and implement class templates;
- Design, implement, and test medium programs in C++;
- Cross-platform compilation with CMake;
- Maintain coding projects with Git on github.
- Finish reading §5.1 to §5.5 of Textbook.

3 Overview and Requirements

3.1 Overview

In this assignment, you will implement two hash tables with two classic collision techniques: chaining and linear probing. Hash tables are data structures that provide efficient access to data by associating keys with values, offering average-time complexity of $O(1)$ for insertion, deletion and search operations. However, when two keys hash to the same index, a collision occurs, i.e., $hash(x) = hash(y)$ where $x \neq y$ (recall that we usually do not allow duplicate keys in hash tables). This assignment will guide you through implementing solutions to handle such collisions.

Chaining (or separate chaining) is a closed addressing method that handles collisions by linking elements with the same hash key into a linked list at each hash table slot/bucket. When a collision occurs, the new element is simply added to the list at the index computed by the hash function. This approach keeps the hash table slots compact while supporting dynamic expansion as the table grows.

Linear probing, on the other hand, is an open addressing method that uses sequential probing to resolve collisions. When a collision occurs, the algorithm checks the next available slot in the array, moving linearly through the table until an empty position is found. This approach has its own advantages, particularly in terms of memory usage, as it avoids the need for linked lists. However, it also has challenges, such as (primary) clustering, which can occur when consecutive slots are filled, potentially affecting the performance of the hash table as it fills up.

3.2 Requirements

In this project, you will implement functions for two hash tables (separate chaining and linear probing). Suppose that our goal is to establish a quick lookup table for employees' salary with hashing.

The starter code provides an implementation of employees' information with `class Employee` in "Employee.h", which has two members `string name` and `double salary`. In this header file, `class hash<Employee>` is a specialization of the std::hash template class from the C++ Standard Library, by which we can define our custom hash function for the Employee.

"utils.h" declares a number of useful functions and variables as follows:

```
extern Employee emp1;
extern Employee emp2;
extern Employee emp3;
extern Employee emp4;

bool isPrime( int n );
int nextPrime( int n );
string generateARandomName(int length);
vector<string> generateRandomNames(int numNames);
int generateARandomInteger(int MAXNUM);
vector<int> generateRandomIntegers(int numNumbers);
```

We define `emp1`, `emp2`, `emp3` and `emp4` as global variables and use them as test data for `insert` and `remove` functions. `isPrime` returns whether an input integer `int n` is a prime, and `nextPrime` returns the smallest prime number that is greater than the input `int n`. `generateARandomName` returns a randomly generated string with a pre-defined length and `generateRandomNames` returns an array of such strings. `generateARandomInteger` returns a randomly generated integer upper bounded by `MAXNUM`, and `generateRandomIntegers` constructs a vector of such numbers. These four functions are used to generate random data for our project.

The starter code also includes two class templates for the target two hash tables:

- `class ChainingHash` in "SeparateChaining.h". It implements a standard interface for hash tables with separate chaining following Textbook (see §5.3).
- `class ProbingHash` in "LinearProbing.h". It follows the standarf interface for hash tables with probing, as in §5.4 of Textbook.

Their public interfaces may not be changed mostly, but you could add more public/private variables/methods as you see fit. For both HashTables, you need to implement the standard operations and rehashing logic. For separate chaining, rehashing is done when the load factor $\lambda \geq 1$ (load factor is defined on Page 198 in Textbook. Rehashing for separate chaining in Figure 5.10 of Textbook). For linear probing, rehashing is automatically performed at a load factor λ of 0.5 or larger (see Figure 5.17 in Textbook). Rehashing will re-distribute all existing values to the new buckets. During the rehashing, you should find a new *prime number* as the new number of buckets to make the load factor $\lambda < 0.5$. The specific requirements for the implementation is:

- **(45 pts) Hash table with chaining.** This is a hash table where the data is stored in *a vector of lists*. Feel free to use the C++ STL `std::vector` and `std::list` classes. You will also need to keep track of the total number of elements stored in the hash for calculating "size."

  ```
  bool contains( const HashedObj & x ) const;
  void makeEmpty( );
  bool insert( const HashedObj & x );
  bool insert( HashedObj && x );
  bool remove( const HashedObj & x );
  void rehash( );
  double loadFactor();
  ```

- `contains` (5 pts): refer to Figure 5.9 in textbook
- `makeEmpty` (5 pts): refer to Figure 5.9 in textbook
- `insert` (10 pts): refer to Figure 5.10 in textbook. There are two versions for "insert" that takes Lvalue and Rvalue as the input, respectively.
- `remove` (10 pts): refer to Figure 5.9 in textbook
- `rehash` (10 pts): refer to Figure 5.22 in textbook
- `loadFactor` (5 pts): compute the load factor of hash table, defined on Page 198 of textbook

- **(55 pts) Hash table with linear probing.** This is a hash table with a single vector of elements, but it MUST do *lazy deletion* (see the detailed introduction to lazy deletion on Page 204 in Textbook, as well as the corresponding code in Figure 5.14 therein). It means that we need to flag a value as DELETED, but only physically delete it when a new value is inserted at the same place/bucket in hash table. This hash table needs to keep track if a bucket is EMPTY, ACTIVE, or DELETED, and your code should use that information when probing and rehashing. You will need to keep track of the total size of the hash table (excludes those marked "DELETED" values). Following the notations in §5.4 of Textbook, this is doing linear probing:

$$h_i(x) = \Big(hash(x) + f(i)\Big) \bmod TableSize, \text{ where } f(i) = i.$$

```
bool contains( const HashedObj & x ) const;
void makeEmpty( );
bool insert( const HashedObj & x );
bool insert( HashedObj && x );
bool remove( const HashedObj & x );
int findPos( const HashedObj & x ) const;
void rehash( );
double loadFactor();
```

- `contains` (5 pts): refer to Figure 5.16 in textbook for quadratic probing
- `makeEmpty` (5 pts): refer to Figure 5.15 in textbook for quadratic probing
- `insert` (10 pts): refer to Figure 5.17 in textbook for quadratic probing
- `remove` (10 pts): refer to Figure 5.17 in textbook for quadratic probing
- `findPos` (10 pts): refer to Figure 5.16 in textbook for quadratic probing
- `rehash` (10 pts): refer to Figure 5.22 in textbook for qudratic probing
- `loaFactor` (5 pts): compute the load factor of hash table, defined on Page 198 of textbook.

The "main.cpp" already has the following test cases (You may not change the "main.cpp"), and you need to make sure these test cases can run properly:

```
void testChainingHash()
{
    ChainingHash<Employee> employeeChainingHash;
    initializeHash(employeeChainingHash);
    testInsertToHash(employeeChainingHash);
    testRemoveFromHash(employeeChainingHash);
    testRehash(employeeChainingHash);
}
```

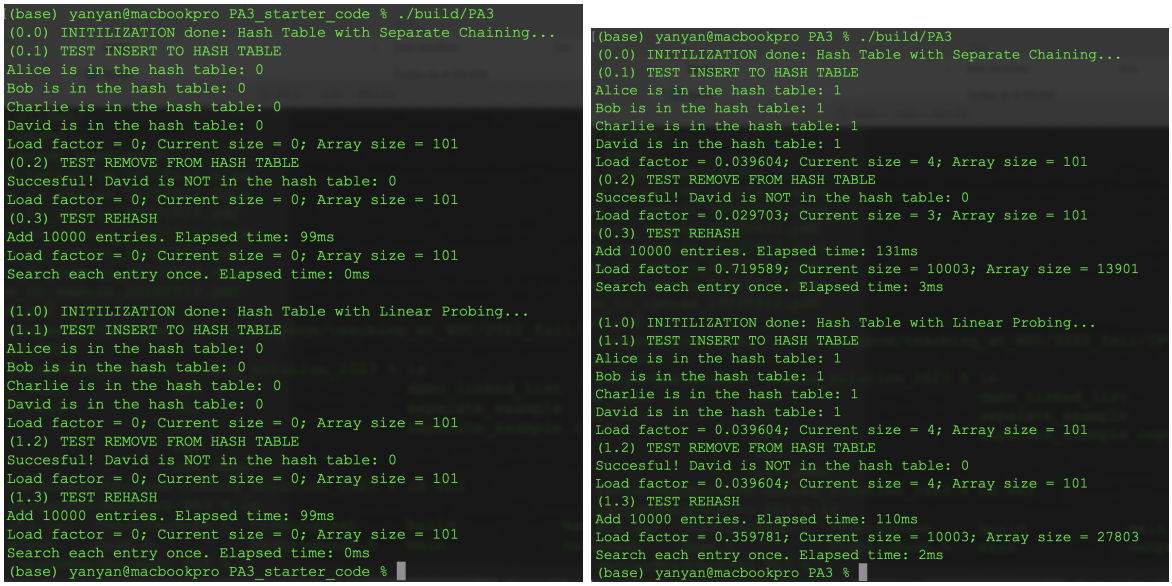

Figure 1: Example of running the starter code and final code.

```
void testProbingHash()
{
    ProbingHash<Employee> employeeProbingHash;
    initializeHash(employeeProbingHash);
    testInsertToHash(employeeProbingHash);
    testRemoveFromHash(employeeProbingHash);
    testRehash(employeeProbingHash);
}
```

Figure 1 gives the results of running the starter code on left and the result of running the final code on right.

4 How to Submit: Github (and Share with TA)

1. In your Git repository for this class's coding assignments, **create a new branch called "PA3"** (Refer to this YouTube video about how to create a branch from Github web interface or the terminal). In the current working directory, also create a new directory called "PA3" and place all PA3 files in the "PA3" directory. All files for PA3 should be added, committed and pushed to the remote origin which is your private GitHub repository created when during PA1 (NO NEED TO CREATE A NEW REPO).

2. You should submit at least the following files:

   - the header files ("Employee.h", "utils.h", "SeparateChaining.h", "LinearProbing.h" and "testSeparateChaining.h" and "testLinearProgramming.h"). You will implement the hash tables in "SeparateChaining.h" and "LinearProbing.h".
   - the main C++ source file ("main.cpp", "testSeparateChaining.cpp", "testLinearProbing.cpp"), where you will have all tests passed;
   - a "CMakeLists.txt" file containing your CMake building commands on Linux/WSL/MacOS which can compile your code.

3. You can refer the example GitHub template project for how to use CMake at https://github.com/DataOceanLab/CPTS-223-Examples.

4. Please invite the GitHub accounts of TAs (see Syllabus page and check TA's names as well as their Github usernames before submitting) as the collaborators of your repository. You should submit a URL link to the branch of your private GitHub repository on Canvas. Otherwise, we will not be able to see your repository and grade your submission.

5. Please push all commits of this branch before the due date for submitting your Github link to Canvas portal. Otherwise it might be considered as late submission.

Here is a checklist for submitting your code:

- Invite all TAs and instructor as collaborators of your created repository "CPTS223_assignments". Their github username can be found in Syllabus on Canvas.
- Commit and push your local code to your repository.
- Make sure that your repository reflects and contains all your latest files.
- Copy the link to your repository in the Canvas submission portal.

5 Grading Guidelines

This assignment is worth 100 points. We will grade according to the following criteria: See Section 3.2 for individual points.