Assignment 3: Hashing and hash functions

COL106: Data Structures and Algorithms, Semester-I 2023–2024

Deadline: 5th September 2023

Honor Code

Following is the Honor Code for this assignment.

- Copying programming code in whole or in part from another or allowing your own code (in whole or in part) to be used by another in an assignment meant to be done individually is a serious offence.
- The use of publicly available codes on the internet such as GeeksForGeeks is **not allowed** for this assignment.
- The use of ChatGPT and other AI tools is **strictly forbidden** for this assignment. If any student has been found guilty, it will result into disciplinary action.
- Collaboration with any other person or team would be construed as academic misconduct.
- Outsourcing the assignment to another person or "service" is a very serious academic offence and could result in disciplinary action.
- Please ensure that your assignment directories and files are well-protected against copying by others. Deliberate or inadvertent copying of code will result in penalty for all parties who have "highly similar" code. Note that all the files of an assignment will be screened manually or by a program for similarity before being evaluated. In case such similarities are found, the origin as well as destination of the code will be held equally responsible (as the software cannot distinguish between the two).

Introduction

Welcome to this data structures assignment where you will step into the shoes of a database manager overseeing a country's intricate banking system. Your role involves maintaining a highly efficient and reliable hashmap to store essential bank account information. To make this endeavor feasible, you'll harness the power of hash functions to manage a vast array of bank accounts, each characterized by a combination of its IFSC code and account number.

Consider this scenario: with millions of bank accounts spread across the nation, it's paramount to ensure that each account's data is accurately stored, retrievable, and secure. The hashmap you're building will act as a fundamental pillar of the banking infrastructure, serving as a gateway to access account balances, update financial information, and validate account existence. Hashing, in this context, is the key that unlocks streamlined and efficient banking operations.

To tackle the complexity of real-world scenarios, you'll delve into the realm of collision resolution strategies. These strategies are like safety nets for ensuring that when two or more accounts share the same hash code due to hash function collisions, their data remains organized and accessible. In this assignment, you will not only implement the core functions required for managing bank accounts but also experiment with various collision resolution techniques to uncover the optimal approach for maintaining order amidst potential chaos.

1 Getting Started

To help you get started, we provide a basic skeleton code. The table 2 gives a detailed description of the methods of the `BankData` class which you need to provide definitions for. There's a `Account` struct that has been provided in the boilerplate code itself.

In total, there are 4 strategies that need to be implemented and the competitive part:

- Chaining
- Linear Probing
- Quadratic Probing
- Cubic Probing
- Competitive Part (free to use any above or other optimised strategies)

In the `BankData` class, there are 2 vectors namely *bankStorage*1*d* and *bankStorage*2*d*. You have to use *bankStorage*2*d* for the chaining strategy and *bankStorage*1*d* for probing strategies.

You are **not** allowed to use any other STL data structures than *std::vector* for this assignment. If we find you using them, you will get a zero in the entire assignment.

2 Functions to be implemented

The table below lists down all the functions that you need to implement. You are **not** allowed to change the definition of any function specified. You are free to define any helper function inside the `BankData` class

3 Strategies to be implemented

In Practice Sheet 1 Question 1, we calculated the hash table using Chaining, Linear Probing, Quadratic Chaining, and Cubic Chaining. It's time to implement these strategies in the code.

Note that the size of the hash table i.e. number of unique inputs will be no greater than **100000**

total number of accounts will be less than 100000

3.1 Chaining (30 marks)

In the chaining strategy, the goal is to manage collisions by creating linked lists at each hash table index. Each list handles all accounts that share the same hash code. Through this approach, you will ensure that data remains organized and accessible even when hash function collisions arise.

3.2 Linear Probing (30 marks)

Linear probing is your next tool, allowing you to navigate hash table slots in a linear fashion to find the next available space when a collision occurs. This strategy emphasizes proximity and aims to avoid gaps in your hashmap.

3.3 Quadratic Probing (10 marks)

The quadratic probing strategy introduces a more sophisticated approach by using quadratic increments to find the next available slot. This technique strikes a balance between proximity and distribution, providing an alternative to linear probing.

Function	Description
`void createAccount(string id, int count)`	This function creates a new account with the given `id` and initial `count` value. The `id` uniquely identifies the account, and the `count` parameter represents the initial balance of the account.
`vector<int> getTopK(int k)` do we have to print all the balances?	Retrieves the top `k` balances values from the database. Returns a vector containing at most `k` balances values.
`int getBalance(string id)`	Returns the current balance of the account identified by the provided `id`. In case the id is not present return -1
`void addTransaction(string id, int count)` what should we do if transactions leads to negative	Adds a transaction of `count` value to the account specified by the `id`. If the account is not already present, create a new one and then add this transaction. The transaction amount can be positive or negative, representing deposits or withdrawals, respectively. Transactions will not lead to a negative balance.
`bool doesExist(string id)`	Checks if an account with the given `id` exists in the database. Returns `true` if the account exists, and `false` otherwise.
`int databaseSize()`	Returns the total number of accounts currently stored in the database.
`int hash(string id)`	This function computes the hash value for the given `id`. Use this hash function **only** in your strategies.
`bool delete(string id)`	This function deletes the key stored in database. It will return true after deletion. If the key was not present, then it should return false

3.4 Cubic Probing (10 marks)

Taking the concept further, cubic probing employs cubic increments to search for open slots. This strategy adds an extra layer of complexity to the probe sequence, aiming to distribute data evenly and mitigate clustering issues.

3.5 Competitive Part (20 marks)

To encourage friendly competition among participants, we've introduced a competitive segment where you have the creative freedom to devise and implement your own strategies. The objective is to develop a hash function that effectively reduces the occurrence of collisions. During this segment, your code will undergo testing using a specific data distribution, which will be shared with everyone through Piazza.

here we have to define a new hash function on our own...

Your task involves experimenting with different approaches to discover an optimal hash function that aligns with the provided data distribution. Feel free to explore various techniques and ideas to formulate a hash function that suits the unique characteristics of the given data distribution. The performance of your hash function will be evaluated based on its ability to handle the specified data distribution.

4 Evaluation guidelines

Evaluation will mainly focus on the correctness of your strategies and then efficiency. For the competitive part, emphasis will be given to the efficiency. You may be required to explain your code, complexity, handling of corner cases, and choice of strategy for the competitive part.

5 Submission Instructions

You will submit the zip containing all the files as present in the boilerplate code. Namely : “‘BaseClass.h, Chaining.h, Comp.h, CubicProbing.h, LinearProbing.h, QuadraticProbing.h, Chaining.cpp, Comp.cpp, CubicProbing.cpp, LinearProbing.cpp, QuadraticProbing.cpp“‘ **Carefully adhere to the submission instructions, failure to comply will result in a 10 % penalty.**

6 Points to Note

1. For each part, read the instructions carefully and use only the data structure specified. You will receive a zero in the assignment if you try to use STL data structures and functionalities, or try to get around the assignment (eg. not using linked lists in Part 3)
2. Follow good main memory management practices. We will stress test your submissions. Submissions which do not manage memory properly will likely not receive full marks.