BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CSE 208 (Data Structures and Algorithms II Sessional)
January 2025

Deadline: July 04 11:59 PM

Offline 3: Hashing

Project VaultX: Building a Smart Storage System

The research division of **VaultX AI**, a startup building large-scale language models for low-resource languages, is developing a new subsystem to index and process pseudo-random word data. This subsystem must handle **insertion**, **search**, and **deletion** of keyword-value pairs at scale while adapting automatically to workload patterns.

You are part of the backend systems team and have been tasked with designing an efficient and adaptive **hash-based storage engine** that supports thousands of entries with minimal collision and optimal memory utilization.

The dataset consists of **10,000 randomly generated "words"** — sequences of 5 to 10 lowercase letters (not necessarily meaningful). Each word must be inserted into the table **only once** as a (key, value) pair, and assigned a **unique value starting from 1**, based on insertion order. Duplicate entries should be detected and discarded using the search mechanism.

Here's a simple example to illustrate the process:

Suppose the word generator produces the following sequence:

```
coding

algorithm

coding

datastr

hashing
```

The resulting (key, value) pairs stored in the table would be:

- (coding, 1)
- (algorithm, 2)
- (datastr, 3)
- (hashing, 4)

The second occurrence of "coding" is **discarded**, and the numbering skips ahead, so "hashing" gets the value

VaultX wants your design to support **multiple collision resolution techniques**, and report on system health and performance statistics throughout the process.

✅ Your Assignment: Designing the VaultX HashTable

To build this smart storage engine, implement a `HashTable` class with the following features:

Word Generation and Storage

- Words are randomly generated (5–10 lowercase letters).
- Assign values sequentially: the first successfully inserted word gets value 1, the second gets 2, and so on.
- Use the `search` function to discard duplicates and ensure correct value assignment.

Table Size Initialization

- The initial size **N′** is provided as input.

- Compute the **nearest prime number N >= N′** and use that as the actual table size to ensure good distribution.

Hash Functions

Implement **two distinct hash functions**, `Hash1(k)` and `Hash2(k)`. You may use well-known methods or custom ones, but your design should ensure:

- At least **60% of keys** yield unique hash values.

- You briefly **explain each function** and justify its design in your report.

Collision Resolution Techniques

VaultX's system must support three different techniques for resolving collisions. Your class should implement all of the following:

1. Separate Chaining with Balanced BSTs

- Each slot holds a pointer to a **balanced binary search tree (Red-Black tree)**.
- Colliding entries are stored in the balanced BST instead of a linked list.
- The tree must remain balanced throughout.
- **You have to use both the Hash functions with separate chaining for report generation.**

2. Linear Probing with Step Adjustment

- Formula:

$$\text{linearHash}(k, i) = (\text{Hash}(k) + i \times S) \bmod N$$

  - S is a small prime number (e.g., 3 or 5), selected at initialization.
  - i is the current probe count.
  - Here, Hash(k) is one of the hash functions `Hash1(k)` and `Hash2(k)`. **You have to use both the Hash functions for report generation.**

3. Double Hashing

- Formula:

  DoubleHash(k, i) = (Hash1(k) + i * Hash2(k)) mod N

  Where Hash1(k) and Hash2(k) are the previously chosen hash functions by you and i is the current probe count.

Performance Reporting

To compare the performance of two collision handling techniques in hash tables:

- Separate Chaining using Balanced Binary Search Trees (BSTs).

- Open Addressing (e.g., linear probing, double hashing).

Performance is measured in terms of:

- Total number of collisions.
- Average Search time.
- Average number of probes (only in open addressing).

Procedure:

1. **Word Generation:**

   - Generate all required elements (e.g.,10000 random strings of 5 to 10 lowercase characters) **before** inserting them into the hash table.

   - This ensures that timing measurements are **not affected** by data generation overhead.

2. **Load Factor Variation:**

    - For a given hash table size, gradually increase the **load factor** from `0.4` to `0.9`, in steps of `0.1`.

    - For each load factor α:
      Number of elements to insert = α × Size of the Hash Table

3. **Insertion:**

    - Insert the generated elements into the hash table:

        - You have to use both the **separate chaining with balanced BSTs** and **open addressing** seperately.
        - Measure and Report total number of collisions.

4. **Search Test (Before Deletion):**

    - Randomly select **10% of the inserted elements**.

    - Search for these elements in the hash table.

    - Measure and report:

        - **Average search time** in:

            - Separate chaining

            - Open addressing

        - **Average number of probes** (only for open addressing)

5. **Deletion:**

    - Randomly delete **10% of the inserted elements** from the hash table.

6. **Search Test (After Deletion):**
    - Construct a new search set equal in size to the previous one (i.e., same number of elements as before deletion).
    - This time, ensure that:
        - **Half** of the searched elements were deleted (expected to **not be found**)
        - **Half** are still present in the hash table (expected to **be found**)
    - Measure and report:
        - **Average search time** in both methods
        - **Average number of probes** (only for open addressing)

Present results in the following format for a fixed table size N'.

For load factor 0.4

Method	Hash1 Function					Hash2 Function				
	# of Collisions during insertion	Before Deletion		After Deletion		# of Collisions during insertion	Before Deletion		After Deletion	
		Avg Search Time	Avg Probes	Avg Search Time	Avg Probes		Avg Search Time	Avg Probes	Avg Search Time	Avg Probes
Separate Chaining with balanced BST			N/A		N/A			N/A		N/A
Linear Probing with Step Adjustment										
Double Hashing										

Table 1

(Do the same for other load factors 0.5, 0.6, 0.7, 0.8, 0.9 and generate the tables)

Report Contents

- Short description about the hash functions you used (Hash1 and Hash2) and why you used it.
- Any constants used.
- **Table 1** for each of the load factors (0.4 to 0.9).
- Explain briefly about the impact of the load factors on the results.

Submission Instructions

- Create a folder named with your **7-digit BUET roll number**.
- Include:
  - All **source files**
  - A **report in pdf format** explaining hash functions, performance stats table and constants used.
- Zip it as: `<roll_number>.zip`
- Submit via Moodle by **July 04, 11:59 PM**

Important Notes:

- **Plagiarism** (either side) will result in **100% penalty**
- Failure to follow submission format = **10% penalty**