# Data Structures and Algorithms

Lab 4 on Hashing

ENSIA 2022-2023

## Objectives

- Implement different methods to represent hash tables.
- Compare the different methods analytically.
- Implement different hash functions.
- Implement different methods to handle collisions.
- Explore different applications of hashing.
- Compare hash tables with binary search trees.

## Prerequisites

C++ Classes (1.4), C++ Details (1.5), and Template (1.6) from the course textbook[1]

## Exercise 1

Create a basic hash table class that supports the following operations: insertion, search, and removal of values from the table.
You can implement the hash table as an array and use a simple hashing function, such as: $key =$ value % TABLE_SIZE, also for this exercise you don't need to find solutions to address the cases of collisions. You are only asked to compare the number of collisions produced by the hashing function following this experiment Setup.
**Experiment Setup:**

- Set TABLE_SIZE to 1000.
- Perform insertion of random values, filling 20%, 50%, and 80% of the hash table capacity.
- Record and compare collision occurrences produced for each filling percentage.

[1]Data Structures and Algorithm Analysis in C++, Fourth Edition, Mark Allen Weiss

## Exercise 2

- Analyze the provided implementation of separate chaining hash table (`SeparateChaining.h, SeparateChaining.cpp, TestSeparateChaining.cpp`) and answer the following questions:
  1. What are the data structures used to implement this hash table?
  2. How are insert and delete operations implemented in the hash table?
  3. Why is rehashing used, and how is it implemented?
- Reimplement the separate chaining hash table using a vector of singly linked lists.

## Exercise 3

- Analyze the provided implementation of separate chaining hash table (`QuadraticProbing.h, QuadraticProbing.cpp, TestQuadraticProbing.cpp`) and answer the following questions:
  1. What are the data structures used to implement this hash table?
  2. How are insert and delete operations implemented in the hash table?
  3. How is collision handling implemented?
  4. Why is rehashing used, and how is it implemented?
- Implement Linear probing and Double hashing.

## Exercise 4

- Write a program to compute the number of collisions that occur in a long random sequence of insertions using Linear probing, Quadratic probing, and Double hashing. To do this, use five experiments while changing the random seed each time to get an unbiased estimate.
- Extend the previous program to: count the number of collisions with different load factors (0.10, 0.15, ...0.95), profile the algorithms, and plot the curve of number of collisions vs sequence length.
- Do the same profiling, but this time based on the sequence insertion execution time.
- Which of the three algorithms performs better?
- Compare the winning algorithm with the provided implementation of the Cuckoo hash method. What do you notice?

## Exercise 5

(a) Modify the word puzzle program of Lab 2 by using a hash table of words to reduce the time of looking up a word in the dictionary.

(b) We can get a big speed increase by storing, in addition to each word W, all of W's prefixes. (If one of W's prefixes is another word in the dictionary, it is stored as a real word.) Although this may seem to increase the size of the hash table drastically, it does not, because many words have the same prefixes. When a scan is performed in a particular direction, if the word that is looked up is not even in the hash table as a prefix, then the scan in that direction can be terminated early. Use this idea to write an improved program to solve the word puzzle.

(c) If we are willing to sacrifice the sanctity of the hash table ADT, we can speed up the program in part (b) by noting that if, for example, we have just computed the hash function for "excel," we do not need to compute the hash function for "excels" from scratch. Adjust your hash function so that it can take advantage of its previous calculation.

(d) Compare the above hash solution binary search tree based solution (or binary search), which is better?

## Exercise 6

Write a program to implement the following strategy for multiplying two sparse polynomials $P1$, $P2$ of size $M$ and $N$, respectively. Each polynomial is represented as a list of objects consisting of a coefficient and an exponent. We multiply each term in $P1$ by a term in $P2$ for a total of $MN$ operations. One method is to sort these terms and combine like terms, but this requires sorting $MN$ records, which could be expensive, especially in small-memory environments. Alternatively, we could merge terms as they are computed and then sort the result.

a. Write a program to implement the alternative strategy.

b. If the output polynomial has about $O(M+N)$ terms, what is the running time of both methods?

## Exercise 7

Implement a spelling checker by using a hash table. Assume that the dictionary comes from two sources: an existing large dictionary and a second file containing a personal dictionary. Output all misspelled words and the line numbers on which they occur. Also, for each misspelled word, list any words in the dictionary that are obtainable by applying any of the following rules:

a. Add one character.

b. Remove one character.

c. Exchange adjacent characters.

## Exercise 8

Implement the classic cuckoo hash table in which two separate tables are maintained[2]. The simplest way to do this is to use a single array and modify the hash function to access either the top half or the bottom half.

[2]Details on this cuckoo implementation can be found in the textbook

## Exercise 9

Implement a hopscotch hash table[3] and compare its performance with linear probing, separate chaining, and cuckoo hashing.

[3]Details on this hopscotch implementation can be found in the textbook