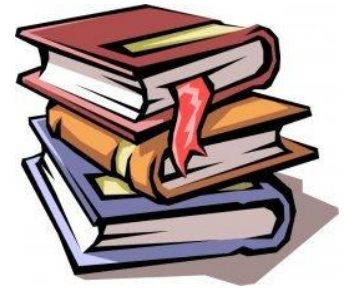


CS 202-Data Structures

Dr. Ihsan Ayyub Qazi



Assignment 3

Hash Tables

(Due: 11pm on Friday, March 25th, 2016)

In this assignment, you are required to implement different types of hash tables. You will then store a dictionary of words and compare the lookup times for the different types of hash tables. The dictionary file is provided to you.

The course policy about plagiarism is as follows:

1. Students must not share actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection.

Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee



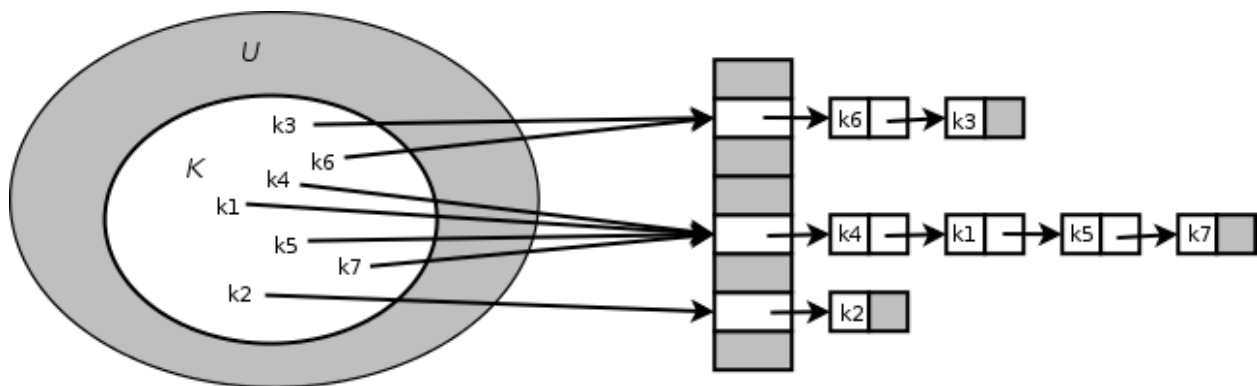
Task 1:

In the first task of this assignment, you have to implement the [polynomial hash code](#) from the textbook (Section 9.2.3) along with the [division method](#) compression function. The value of the parameter a should be configurable.

Task 2:

The specifics of the first hash table are as follows:

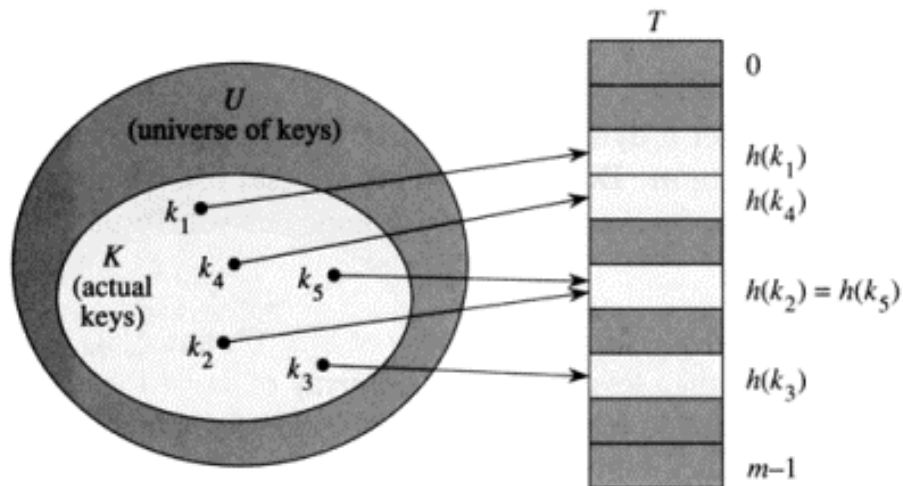
The first hash table will use [chaining](#), where you will be required to use the [LinkedList](#) from previous assignments. This [HashTable](#) will be created with a [fixed size](#). It should support the [insert](#) and [lookup](#) commands. The constructor should take, as a parameter, the value of a .



Counting collisions: As far as collisions are concerned, if you hash a value, and the index it hashes to has a non-empty linkedlist, the collision is counted only once. Do not think of the length of the linked list at that slot as the number of collisions for this assignment.

Task 3:

Now you will try out the same hash function with a different hash table, which should use [open addressing](#) with [linear probing](#). This [HashTable](#) will initially be created with a small size; it must support [resizing](#) along with [insert](#) and [lookup](#). Similar to the previous hash table, the constructor should take, as a parameter, the value of the parameter a .



Counting collisions: As far as collisions are concerned, if you hash a value and the index the value hashes to is occupied, count that as one collision. When you start linear probing to find the next free slot, however many occupied slots you encounter, each of them will also count as collisions.

Task 4:

As you have seen in the implementations of linear probing and chaining, the issue of collisions was addressed by storing both the colliding values, but these techniques increase the look up time. So, in order to improve this, in this task you will be implementing *double hashing* as discussed in the class using the following functions:

str = $a_1 a_2 a_3 \dots a_{n-1} a_n$

e.g., (in case of **str** = "Hello", 'H' is a_1 , 'e' is a_2 ... 'o' is a_5)

Initialize Hash1 = 0

For every a_i in str

Hash1 \wedge = (Hash1 \ll 5) + (Hash1 \gg 2) + a_i

Hash2(str) = Hash function used in Task 1

Note: Double hashing cannot completely eliminate collisions. To obtain full credit in this task, you will have to devise and implement a method to handle the case when both functions result in a collision.

One method to adopt, for example, would be the following:

Index = $h(\text{key}) + i * d(\text{key})$, where **$h()$** is the first hash function, **$d()$** is the second hash function and **i** is an integer zero onwards (0,1,2,3.....)

Hence to compute the index to insert the value at, use the above formula but keep the value of **i** as 0. If you get a collision then use 1 as the value of **i** . If you get a collision again, use 2 as the value for **i** and so on.

Counting collisions: As far as collisions are concerned, if you hash a value and the index the value hashes to is occupied, count that as one collision. Every time you hash it again, and you get a collision, it will be counted as well.

Task 5:

In this task you are to determine whether a student is enrolled in a specific course or not.

Here is how you go about it:

You are given a roster of students ('names.txt'). First, you need to store all these names in hash tables. You are to make three hash tables. One using chaining, one using open addressing, and one using double hashing. To get the value of a , simply prompt the user to enter the value. The value of a should range from 10 to 50.

Print out the time it takes to store the roster of students into each of these hash tables. Also print out the number of collisions for each hash table.

Once the tables are made, prompt the user to enter the name of the student the user wants to search for. Search for this student in all three hash tables. Print out the time it took for searching in each hash table.

The user should be able to search for as many students as he/she wants. When the user enters the 'quit' command, only then should the program terminate.

Deliverables:

You are required to submit the following:

1. Implementation of the hash table with chaining
2. Implementation of the hash table with open addressing (linear probing)
3. Implementation of the hash table with open addressing (double hashing)
4. Function/interface implemented in Task 5