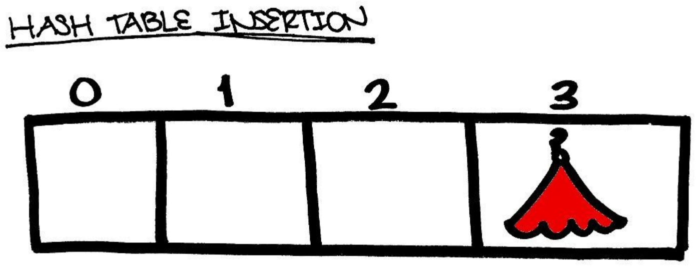
CS 202-Data Structures

Dr. Ihsan Ayyub Qazi

Respective TA(s): Muhammad Haseeb && Abu Bakr Aziz

Assignment 3

Hash Tables



In this assignment, you are required to implement different types of hash tables.

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 are expected to implement (a) the polynomial hash code from the textbook (Section 9.2.3) and (b) bitwise hash code function as shown below:

str = $s_1s_2s_3\ldots s_{n-1}s_n$

e.g., (in case of str = "Hello", 'H' is s_1, 'e' is s_2 … 'o' is s_5)

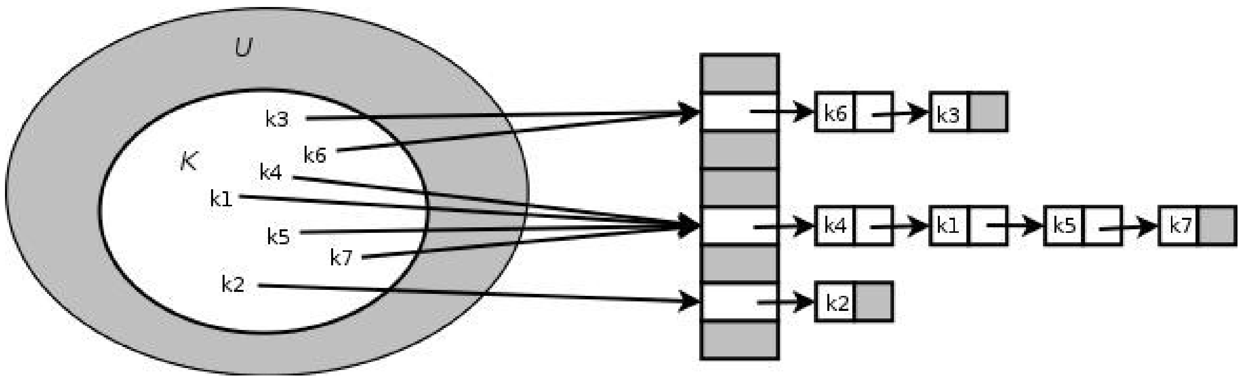
```
Initialize bitwise_hash = 0
For every $s_i$ in str
    bitwise_hash ^= (bitwise_hash << 5) + (bitwise_hash >> 2) + $s_i$
```

along with the division method compression function for both (a) and (b). The value of the parameter **a** in the polynomial hash code should be configurable.

Task 2:

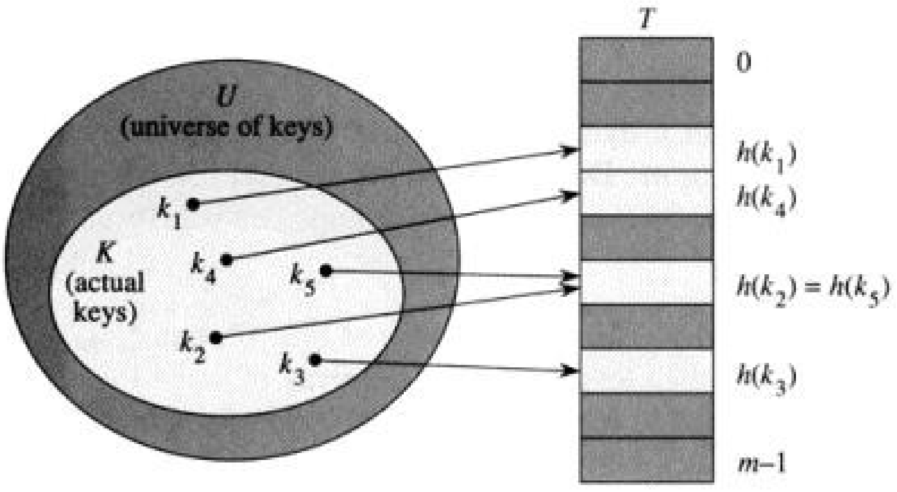
The specifics of the first hash table are as follows:

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, deletion and lookup commands. The constructor should take the size of the table as a parameter. Use hash functions implemented in the Task 1.



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 Hash Table will initially be created with a small size; it must support resizing along with insert, deletion and lookup. Use hash functions implemented in the Task 1.



Bonus: Also implement this with quadratic probing.

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 two functions you made in Task1.

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(key) + i*d(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.

Task 5:

In this task you are going to implement a cache by using your implementation of hash tables. For this part, use the implementation which you think is best (you can also modify that). The details of this part are up to you to decide. Use your best knowledge and optimize your cache.

Your code should read a space separated sequence of codes (numbers) from files *secret1.txt, secret2.txt, secret3.txt* and retrieve the words corresponding to the codes from *dictionary.txt* and print them on the screen. For example, given a sequence "599454 34247 69702 85130", the screen should show "THIS COURSE IS LOVE". Please see the file dictionary.txt for clarification.

You should cache the words once they are fetched. Next time they appear in the sequence, retrieve the corresponding word from the cache instead of *dictionary.txt*. You need to use hash table with size 1000 for your cache. Note that your cache will be limited in size so you cannot store every word in that and you will have to implement any policy for accommodating new values in the cache when it is already full. We would recommend using the policy LFU, *Least Frequently Used (*refer to https://en.wikipedia.org/wiki/Least_frequently_used for details*).*

Your program should run two modes, one in which cache is used and the other one in which cache is not used. Observe the time taken in decoding secrets (numbers) in both modes and print them on screen. You will be graded on the basis of your implementation and understanding of this part. You can use any previous implementations for this part and can include the header files.

* https://en.wikipedia.org/wiki/Cache_(computing)
(This is an open-ended question as you can try to optimize this as you like but do not forget to explain your approach in a separate doc file.)

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. Implementation of the Task 5 in cache.cpp file with a brief doc file explaining your implementation and the time taken for reading sequences from the secret files with cache and without cache.

Note: In order to compile the test cases, you will be required to give the following flags:

```
-pthread –std=c++11
```

As shown in the following example:

```
g++ test_chaining.cpp -pthread –std=c++11
```