Lab 5: Hashing --- Building Your Own Hash Tables

By Dr. Yingwu Zhu, Seattle University

1. Goals

- Understand hashing as an efficient search algorithm, trading space for search performance
- Understand the collision problem of hashing
- Understand collision resolution techniques such as linear probing.
- Implement a strawman hash table by providing basic ***put*** and ***get*** interfaces

2. Tasks

In this lab, you will implement a strawman hash table **MyHashmap** using a hash function:

h(k) = k % N

The strawman hash table **MyHashmap** stores ***(key, value)*** pairs, where *key* is **unsigned int** and *value* is **string**.

Specifically, you need to implement two basic operations provided by a hash table:

- *put(unsigned int k, string v)*, which stores the pair (k, v) under the key *k*. If the key *k* has already exists on the hash table, then replace the old value with *v*. If there is a collision, you need to apply linear probing to resolve it.
- *get(unsigned int k)*, which returns the string value under the key *k* if it exists. Otherwise it returns an empty string

I myself have coded a framework for you to implement these two operations and test your implementation. You will do your work under this framework.

Obtaining the framework files

You will download the framework by executing the command in your lab6 directory:

/home/fac/testzhuy/CPSC250/hash_lab/download

There are six source files.

- myhashmap.h

  - This is our strawman hash table header file. It declares our hash table class **MyHashmap**
  - You do not need to modify it!

- myhashmap.cpp
  - This is our strawman hash table implementation file. All member functions except put() and get() have been implemented
  - You need to implement put() and get() while not touching other functions.

- genrandomstring.h
  - You do not need to touch this file. It is OK if you do not get it.
  - It allows you to generate a random string consisting of letters 'a'-'z'
  - How to use it?

```
//a generator generating string with a max length of 10 letters
RandomStringGenerator   generator(10);
string  s = generator.gen_string();
```

- shadowmap.h & shadowmap.cpp
  - You do not need to touch them. It is OK if you do not get it
  - It is a shadow hash table doing the same work like your strawman hash table **MyHashmap**
  - You can use this shadow hash table to test if your strawman hash table **MyHashmap** works correctly upon put() and get() operations.

- test.cpp
  - This is a driver program to test your strawman hash table
  - You need to modify this file in order to test your hash table

- Makefile
  - You do not need to touch it!

In summary, you only need to modify two files: **myhashmap.cpp** and **test.cpp**

Understand class MyHashmap

Read **myhashmap.h** to understand how this hash table class is declared, especially the data members and member functions.

Implementing put() operation

In this function, you need to store a key-value pair into the hash table. Several questions need to be answered before coding:

- What if the table is full?
- If the key does not exist, you need to store the pair into the table:
  - How to handle collision using linear probing?
  - What data members need to be updated?
- If the key exists, you need to update with the new value

Implementing get() operation

In this function, you need to retrieve the value for a key if it exists. Otherwise return an empty string. Understand how linear probing makes this operation complicated.

Testing your strawman hash table

In the **test.cpp**, all required header files are included.
You need to test if your hash table works correctly. Here **ShadowMap** comes into play!
In order to use **ShadowMap**, you need to define an object. For example:

```
ShadowMap    smap;
```

Assume you defined an object for your own hash table, say

```
MyHashmap  my_map(10); // with 10 slots in the hash table
```

Now, whenever you insert a key-value pair into your hash table my_map, you also store it into the ShadowMap smap. For example:

```
string  s = generator.gen_string();
if (!my_map.full()) {
        my_map.put(10, s);
        smap.put(10, s);
}
```

Then, you can test if your hash table stores the pair correctly as the ShadowMap does. We always trust ShadowMap performs correctly. For example:

```
assert(my_map.get(10) == smap.get(10));
```

If they do not match, the program will halt. It means your hash table has problems in put() and/or get() operations. Then fix them!

You need to test your MyHashmap comprehensively using a set of use cases!