The following materials have been collected from the numerous sources including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated.

Final: Hashing

Table of Contents

Hashing

Hashing is an important data structure which is designed to use a special function called the **hash function.** The hash function is used to map a given value with a particular key for **faster access** of elements. Hashing supports insertion, deletion and search in average case constant time O(1). The following table summarizes the time complexity of data structures which we have learned this semester.

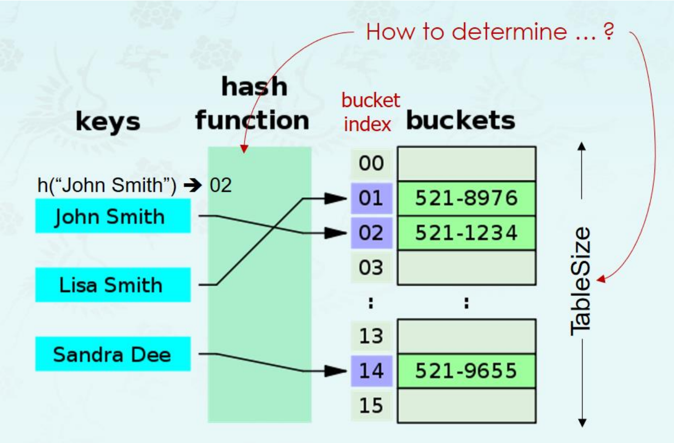
C++ STL Data Structure	Insert	Find	Delete
vector	O(n)	O(n)	O(n)
sorted vector	O(n)	O(log n)	O(n)
linked list (list, stack, queue)	O(1)	O(n)	O(1)
balanced BT (map, set)	O(log n)	O(log n)	O(log n)
Heap (priority_queue)	O(log n)	O(1) for min/max	O(log n)

hash table (unordered_map/set)	O(1)	O(1)	O(1)

Hash table main components

Thus we can say that hashing is implemented using two steps as mentioned below:

1) The value is converted into a unique integer key or hash by using a hash function. It is used as an index to store the original element, which falls into the hash table.
2) The element from the data array is stored in the hash table where it can be quickly retrieved using the hashed key.



In the above diagram, we saw that we have stored all the elements in the hash table after computing their respective locations using a hash function. We can use the following expressions to retrieve hash values and index.

hash = hash_func(key)
index = hash % table_size

Hash Function

The efficiency of mapping depends of the efficiency of the hash function used. A hash function basically should fulfill the following requirements:

- Easy to Compute: A hash function, should be easy to compute the unique keys.
- Less Collision: When elements equate to the same key values, there occurs a collision. There should be minimum collisions as far as possible in the hash function that is used. As collisions are bound to occur, we have to use appropriate collision resolution techniques to take care of the collisions.
- Uniform Distribution: Hash function should result in a uniform distribution of data across the hash table and thereby prevent clustering.

Hash Table

The Hash table or a hash map is a data structure that stores pointers to the elements of the original data array. Having entries in the hash table makes it easier to search for a particular element in the array. The hash table uses a hash function to compute the index into the array of buckets or slots using which the desired value can be found.

Collision

In the case of hashing, even if we have a hash table of very large size then a collision is bound to be there. This is because we find a small unique value for a large key in general, hence it is completely possible for one or more value to have the same hash code at any given time.

Given that a collision is inevitable in hashing, we should always look for ways to prevent or resolve the collision. There are various collision resolution techniques that we can employ to resolve the collision occurring during hashing.

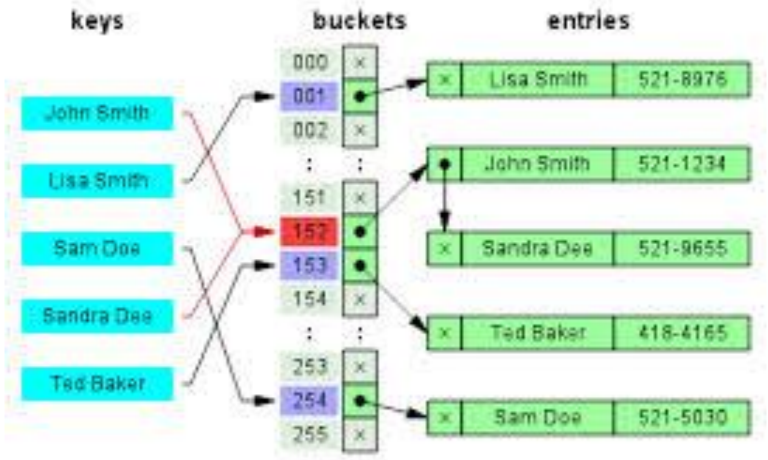
Collision Resolution Techniques

Mainly there are two kinds of collision resolution techniques. The first one is open hashing or separate chaining. The other is called closed hashing or open addressing.

- Separate **Chaining** (or Open hashing)
- **Open Addressing** (or Closed hashing)
  - Linear Probing
  - Quadratic probing
  - Double hashing

Open hashing or Separate Chaining

Open Hashing is a technique in which the data is not directly stored at the hash key index (k) of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked.



Each entry in the hash table is a linked list. When the key matches the hash code, it is entered into a list corresponding to that particular hash code. Thus when two keys have the same hash code, then both the entries are entered into the linked list. The worst-case for separate chaining is when all the keys equate to the same hash code and thus are inserted in one linked list only. Hence, we need to look up for all the entries in the hash table and the cost which are proportional to the number of keys in the table.

Closed hashing or Open Addressing

In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is adopted.

Liner Probing

In open addressing or linear probing technique, all the entry records are stored in the hash table itself. When key-value maps to a hash code and the position pointed to by hash code is unoccupied, then the key value is inserted at that location. If the position is already occupied, then using a probing sequence the key value is inserted in the next position which is unoccupied in the hash table.

For linear probing, the hash function may change as shown below:

hash = hash % hashTableSize
hash = (hash + 1) % hashTableSize
hash = (hash + 2) % hashTableSize
hash = (hash + 3) % hashTableSize

Linear probing may suffer from the problem of "Primary Clustering" in which there is a chance that the continuous cells may get occupied and the probability of inserting a new element gets reduced.

Quadratic Probing

Quadratic probing is the same as linear probing with the only difference being the interval used for probing. As the name suggests, this technique uses non-linear or quadratic distance to occupy slots when a collision occurs instead of linear distance.

In quadratic probing, the interval between the slots is computed by adding an arbitrary polynomial value to the already hashed index. This technique reduces primary clustering to a significant extent but does not improve upon secondary clustering.

hash = hash % hashTableSize
hash = (hash + 1) % hashTableSize
hash = (hash + 4) % hashTableSize
hash = (hash + 9) % hashTableSize

Double Hashing

The double hashing technique is similar to linear probing. The only difference between double hashing and linear probing is that in double hashing technique the interval used for probing is computed using two hash functions. Since we apply the hash function to the key one after the other, it eliminates primary clustering as well as secondary clustering.

hash(x) = hash % hashTableSize
hash(x) = (hash + 1 * (R – (x % R)) % hashTableSize, R is prime number less than hashTableSize
hash(x) = (hash + 2 * (R – (x % R)) % hashTableSize
hash(x) = (hash + 3 * (R – (x % R)) % hashTableSize

Rehashing

Rehashing is the process of re-calculating the hash value of already stored entries (Key-Value pairs), to move them to another bigger size hash table when the threshold is reached/crossed.

Rehashing is done because whenever a new key value pair is inserted into map, the load factor increases and due to which complexity also increases. And if complexity increases our HashMap will not have constant O(1) time complexity.

How to:

- For every new entry into the map, check the load factor.
- If the load factor is greater than its threshold value (default 1.0 for Hash Table), then start Rehash.
  - For Rehashing, initialize a new array of double the size and its next prime of the previous one.
  - Reinsert all elements into a new array and make it the new bucket array.

Getting Started

This PSet implements three types of hash table.

- The first one uses an array of the **list** STL for the hash table and string type keys.
- The second one uses an array of the list for the hash table. Each element consists of a pair of string and int type data for the key and the value, respectively.
- The third one uses **unordered_map** class in STL to replace hash2.cpp functionality.

The list of files are provided as follows:

- hashing.pdf – this file
- hash1.h, **hash1.cpp**, hash1Driver.cpp - Hash table storing string type elements
- hash2.h, **hash2.cpp**, hash2Driver.cpp - Hash table storing a user-defined data type
- **hashmap.cpp** – The functionality of hash2 is simulated using **unordered_map** in STL.
- hash1x.exe, hash2x.exe, hashmapx.exe - example solutions
- makefile – a makefile to build three executables – provided for your convenience only
- The following text(ASCII) files are provided for your testing. To access this files directly, place these files where your executables or MS project files exist.
  ps23.txt – psalm 23 (NIV)
  1co13.txt – 1 Corinthian 13 (NIV)
  kjv.txt – King James Version of the Bible
  shakespeare.txt – Shakespeare's "A midsummer-night's dream"

You are supposed to complete your coding in **hash1.cpp, hash2.cpp,** and **hashmap.cpp.**

Step1: Complete hash1.cpp

This skeleton code, hash1.cpp, implements a hash table using the **list** class in STL. The Hash structure is defined **hash1.h** as shown below:

```
struct Hash {
 int            tablesize;             // hash table size
 list<string>*  hashtable;             // pointer to an array of buckets
 int            nelements;             // number of elements in table
 double         threshold;             // max_loadfactor

 Hash(int size = 2, double lf = 1.0) {// a magic number, use a small prime
    tablesize = size;                   // using list<string> for pedagogical purpose
    hashtable = new list<string>[size];// but vector<list<string>> may be used
    nelements = 0;
    threshold = lf;                     // rehashes if loadfactor >= threshold
 }
 ~Hash() {
    delete[] hashtable;
 }
};
```

Three ways to build executables

There are a few different ways of building the executables. For example, let us build hash1.exe;

1. Without hash1Driver.cpp – This is for a fast testing of your code.
   A. Turn on the macro as #if 1 above main() in hash1.cpp
   B. Use the following build command: add **-std=c++11** for OSX.

   ```
   g++ hash1.cpp -o hash1 -I../../include
   ```

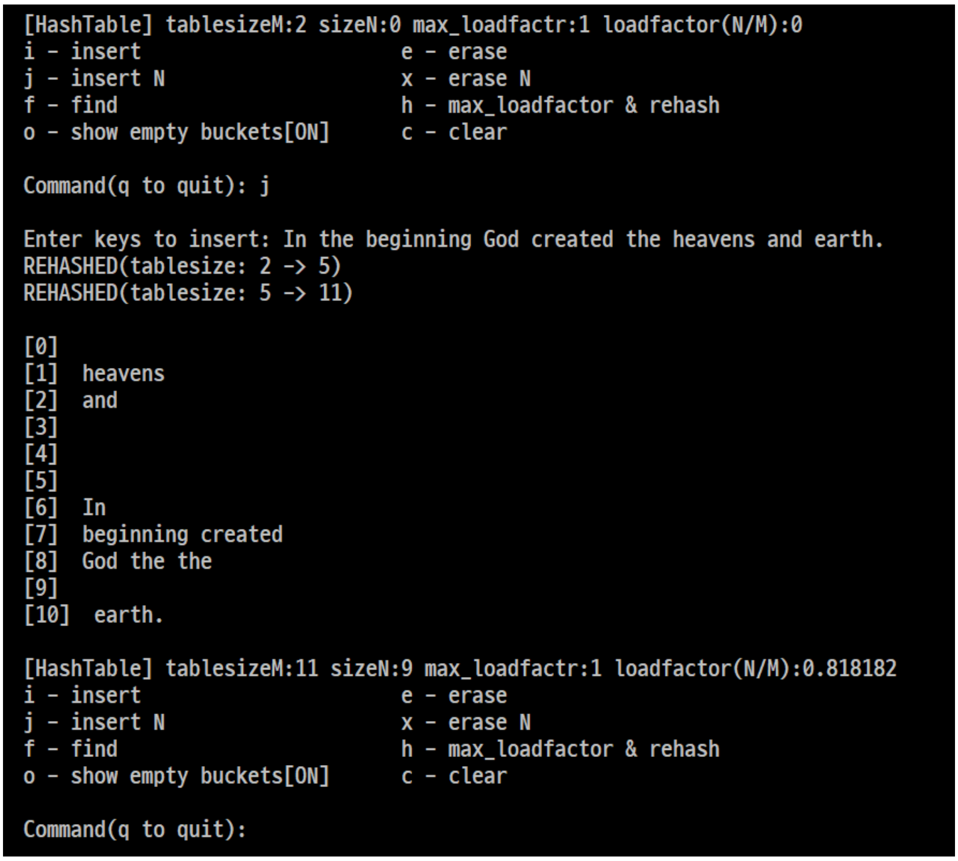
2. With hash1Driver.cpp – This is for your development
   A. Turn off the macro as #if 0 above main() in hash1.cpp
   B. Use the following build command
   For OSX, add **-std=c++11**, and use **-lnowic_mac** instead of **-lnowic** for OSX.

   ```
   g++ hash1.cpp hash1Driver.cpp -o hash1 -I../../include -L../../lib -lnowic
   ```

3. Use makefile. Simply type "make" at the console - This checks all three executables (hash1.exe, hash2.exe, hashmap.exe) and builds them if necessary.
   You may need to edit makefile provided according your need or your filenames.

   ```
   make
   ```

Sample Run (with a hash1Driver.cpp):



```
[HashTable] tablesizeM:2 sizeN:0 max_loadfactr:1 loadfactor(N/M):0
i - insert                     e - erase
j - insert N                   x - erase N
f - find                       h - max_loadfactor & rehash
o - show empty buckets[ON]     c - clear

Command(q to quit): j

Enter keys to insert: In the beginning God created the heavens and earth.
REHASHED(tablesize: 2 -> 5)
REHASHED(tablesize: 5 -> 11)

[0]
[1]  heavens
[2]  and
[3]
[4]
[5]
[6]  In
[7]  beginning created
[8]  God the the
[9]
[10]  earth.

[HashTable] tablesizeM:11 sizeN:9 max_loadfactr:1 loadfactor(N/M):0.818182
i - insert                     e - erase
j - insert N                   x - erase N
f - find                       h - max_loadfactor & rehash
o - show empty buckets[ON]     c - clear

Command(q to quit):
```

Implement a typical set of commands to handle the hash table such as insert(), erase() and find() and rehash(). For simplicity, this implementation just works with the keys, but not the values. To resolve the collision, it uses the separate **chaining** scheme.

1. **insert**
   A. It takes a key from the user and insert it into a hash table.
   B. It rehashes if loadfactor becomes greater than max_loadfactor (or `threshold` in the code) **after** the key is inserted.
2. **erase**
   A. It takes a key and erase it from the hash table. No need to rehash. The hash table does not rehash to get smaller even the number of elements are reduced.
3. insert N and erase N
   A. It works like insert or erase, but takes multiple input keys
4. **find**
   A. It takes a key from the user and search for it in the table, and display the result.
   B. It returns a bucket of which have the same as the key. It may have multiple elements with the same hash value. The main function driver prints all the elements in the bucket.
5. max_loadfactor & **rehash**
   A. It takes a float point number and sets the max_loadfactor.
   B. It compares max_loadfactor and loadfactor,
      then rehashes if loadfactor ≥ max_loadfactor
   C. The max_loadfactor is set and loadfactor is displayed properly after rehashing.
6. clear
   A. It removes all the elements in the hash table and start a new hash table.
7. show empty bucket [ON]
   A. This toggles on/off whether or not it displays the blank buckets.

Step2: Complete hash2.cpp

This skeleton code, hash2.cpp, also implements a hash table using the **list** class in STL. Each element consists of a pair of data such as string type data and int type data of which is called "wordcount". It can be defined as shown below:

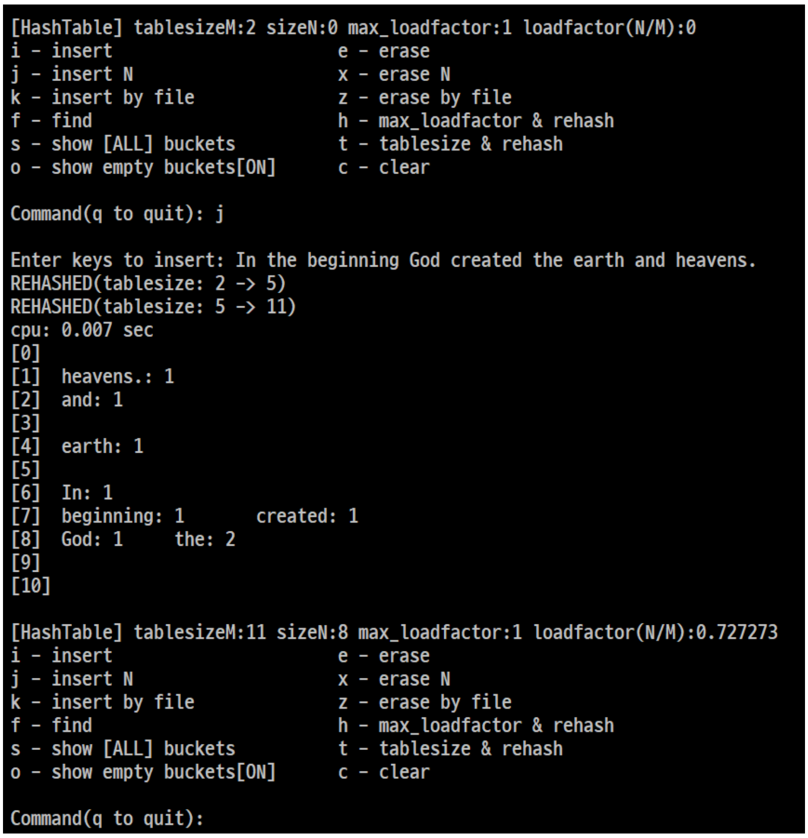
The Hash structure is defined in **hash2.h** as shown below:

```
typedef std::pair<string, int> wordcount;

struct Hash {
 int              tablesize;          // hash table size or bucket_count()
 list<wordcount>* hashtable;          // pointer to an array of buckets
 int              nelements;          // number of elements in table or size()
 double           threshold;          // threshold(or max_loadfactor)

 Hash(int size = 2, double lf = 1.0) {  // a magic number, use a small prime
    tablesize = size;
    hashtable = new list<wordcount>[size];
    nelements = 0;
    threshold = lf;                     // rehashes if loadfactor >= threshold
 }
 ~Hash() {
    delete[] hashtable;
 }
};
```

Sample Run:



```
[HashTable] tablesizeM:2 sizeN:0 max_loadfactor:1 loadfactor(N/M):0
i - insert                      e - erase
j - insert N                    x - erase N
k - insert by file              z - erase by file
f - find                        h - max_loadfactor & rehash
s - show [ALL] buckets          t - tablesize & rehash
o - show empty buckets[ON]      c - clear

Command(q to quit): j

Enter keys to insert: In the beginning God created the earth and heavens.
REHASHED(tablesize: 2 -> 5)
REHASHED(tablesize: 5 -> 11)
cpu: 0.007 sec
[0]
[1]  heavens.: 1
[2]  and: 1
[3]
[4]  earth: 1
[5]
[6]  In: 1
[7]  beginning: 1       created: 1
[8]  God: 1     the: 2
[9]
[10]

[HashTable] tablesizeM:11 sizeN:8 max_loadfactor:1 loadfactor(N/M):0.727273
i - insert                      e - erase
j - insert N                    x - erase N
k - insert by file              z - erase by file
f - find                        h - max_loadfactor & rehash
s - show [ALL] buckets          t - tablesize & rehash
o - show empty buckets[ON]      c - clear

Command(q to quit):
```

Implement a typical set of commands to handle the hash table such as insert(), erase() and find() and rehash(). Use the separate chaining scheme to resolve the collision.

1. **insert**
   A. It takes a key from the user and insert it into a hash table and sets its value to 1 if it is the new key. If it is a duplicate key, just increment the value by one. For example, there are must be two "the" since we see "the 2" in the screen-capture shown above. The bucket or an element works like a word counter.
   B. It rehashes if loadfactor becomes equal to or greater than max_loadfactor (or `threshold` in the code) **after** the key is inserted.
2. **erase**
   A. It takes a key and erase it from the hash table. It does not bother to decrement count by one, but remove the element itself. No need to rehash. The hash table does **not** rehash to get smaller even the number of elements are reduced.
3. **insert N and erase N**
   A. It works like insert or erase, but takes multiple input keys
4. **insert by file, erase by file**
   A. It inserts or erases all the words in the hash table as it reads them from the file.
   B. It excludes words that begin with numbers
   C. It includes words that end with a punctuation after removed.
5. **find**
   A. It takes a key from the user and search for it in the table, and display the result.
   B. It returns a bucket of which has the same as the key's hash value. It may have multiple elements with the same hash value. The main function driver prints all the elements in the bucket like [7] and [8] bucket.
6. max_loadfactor & **rehash**
   A. It takes a float point number and sets the max_loadfactor.
   B. It compares max_loadfactor and loadfactor,
   then rehashes if loadfactor ≥ max_loadfactor
   C. The max_loadfactor is set and loadfactor is displayed properly after rehashing.
7. tablesize & **rehash**
   A. It takes an integer number and **sets it as the new tablesize.**
   In this case, the tablesize may not be a prime number.
   B. It must rehash if the new table size is different from the old one.
8. show empty bucket [ON]
   A. This toggles on/off whether or not it displays the blank buckets.

Step3: Complete hashmap.cpp

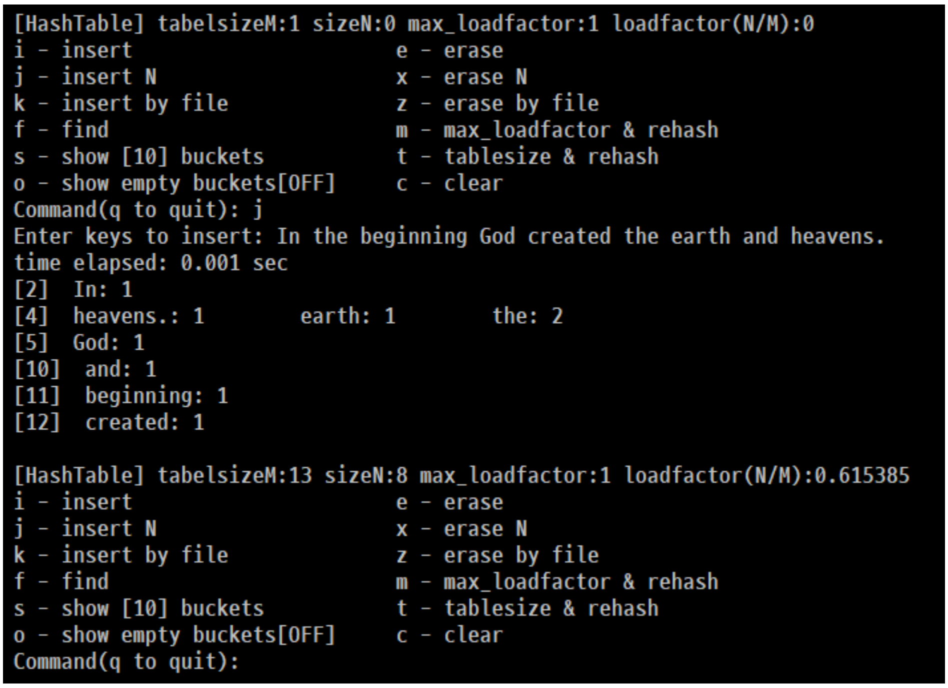
Implement the same functionality you implemented in the previous step while using **unordered_map** in STL. Make an every effort such that the result of hash2.cpp matches that of hashmap as much as possible except "Find" command. Remember there are something that we do not match intentionally or unintentionally.

- **"find" option:**
  - "find" command returns not a list of <string, int> but, **the element <string, int> matched.**
  This is different from hash2.cpp
- **"show [ALL/N] buckets" option:**
  - Implement this option just like you have used in the previous step.
- The max_loadfactor is set and loadfactor is displayed properly after rehashing.

Some observations of unordered_map:

- The hash function used in unordered_map is different from that of hash1 and hash2.
- The table size and successive doubling scheme are different since it does not use prime numbers quite often.
- You may adjust the table size (bucket count) using **reserve()** function. According to the c++ reference listed below, however, you may **not be able to reduce it** even if N ≪ M.
  - Sets the number of buckets in the container (bucket_count) to the most appropriate to contain at least n elements.
  - If n is greater than the current bucket_count multiplied by the max_load_factor, the container's bucket_count is increased and a rehash is forced.
  - **If n is lower than that, the function may have no effect.**

Notice that hashmap.cpp does not depend on other files since we use the unordered_map class in STL except nowic.h.



```
[HashTable] tabelsizeM:1 sizeN:0 max_loadfactor:1 loadfactor(N/M):0
i - insert                      e - erase
j - insert N                    x - erase N
k - insert by file              z - erase by file
f - find                        m - max_loadfactor & rehash
s - show [10] buckets           t - tablesize & rehash
o - show empty buckets[OFF]     c - clear
Command(q to quit): j
Enter keys to insert: In the beginning God created the earth and heavens.
time elapsed: 0.001 sec
[2]  In: 1
[4]  heavens.: 1        earth: 1        the: 2
[5]  God: 1
[10]  and: 1
[11]  beginning: 1
[12]  created: 1

[HashTable] tabelsizeM:13 sizeN:8 max_loadfactor:1 loadfactor(N/M):0.615385
i - insert                      e - erase
j - insert N                    x - erase N
k - insert by file              z - erase by file
f - find                        m - max_loadfactor & rehash
s - show [10] buckets           t - tablesize & rehash
o - show empty buckets[OFF]     c - clear
Command(q to quit):
```

A brief and fast way to test unordered_map class

A simple snippet code is included to test unordered_map class at the bottom of hashmap.cpp. Use this test code to learn unordered_map class if you wish.

You can turn off the macro as **#if 0** defined above in main() that uses nowic.cpp functions. Then you can build the hashmap.cpp without nowic.cpp. You may use the makefile provided as you wish.

Submitting your solution

- Include the following line at the top of your every file with your name signed.
  On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment. Signed: _____________

- Make sure your code **compiles** and **runs** right before you submit it. Don't make "a tiny last-minute change" and assume your code still compiles. You will not receive sympathy for code that "almost" works.
- If you only manage to work out the homework partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

Submit the following files at the **final folder.** Make sure that your code should compile and run with g++ on console.

- Step 1: hash1.cpp
- Step 2: hash2.cpp
- Step 3: hashmap.cpp
- 08-1 HashingFinal.pptx (Edit this file for your answers.)

Due

- **No late work** will be accepted since this is considered as the final exam.

Free Online Lectures on Hashing

The first two lectures listed below are recommended. Let me know if you find and share better lectures online.

1. 컴퓨터 알고리즘 기초 10 강 해쉬 알고리즘(1) | T 아카데미
2. 컴퓨터 알고리즘 기초 11 강 해쉬 알고리즘(2) | T 아카데미