Middle East Technical University | Department of Computer Engineering

CENG 213

Data Structures

Fall '2017-2018

Programming Assignment 3

Due date: 02.01.2018, 23:55

1 Introduction

This assignment aims to get you familiar with hash tables by implementing a word usage statistics engine.

Keywords: *Hash Tables, Double Hashing, Classes, File IO*

2 Problem Definition

Hi, Neo. After you have gone to distant lands, the world slowly began to sink into darkness. There is silence everywhere. Now, the children are not laughing, the birds are not around. But we, a group of scientists, want to reverse this situation. We want to reach as many people as we can and tell them about our research. Also, we want to do this secretly without getting caught by the surrounding robots. We want to store our research digitally with extraordinary methods with your help. You have to help us for the salvation of mankind! You can find a new way to create a link for communication between the Matrix and the real world!

In this last task for this semester, you, Neo, have planned to implement a word usage statistic engine for transfering the articles of this group. Your main purpose is analyzing their messages and creating a hash table for each of these in order to hide the main message from the robots. You can even delete some words to chage their focus. You have chosen **open addressing with double hashing** as your method to implement a **hash table**. Operations that your program should support are **inserting** to that hash table, and **removing and getting data** from hash table. To serve their needs, you will be putting their articles to your table with <std::string , int> pairs such that first elements of those table cell elements will be the word that is shown in article, second one will be the words original position on the article. For example, for an article that consists of 4 words: "Neo throw the ball" will be inserted to the hash table with the pairs as the following(Order may change):

```
   Key,   Original Index
<"Neo" ,      1>
<"throw",     2>
<"the" ,      3>
<"ball" ,     4>
```

As you might guessed, there might be multiple occurences of a word but their original position on the text must be **unique** for each entry.
At any time(after inserting a part of an article, removing a part, before anything etc.) they might want you to find out what was the original index of a "word"s n^{th} occurence on the text, which is the second element of corresponding pair.
On top of that, they might even want you to print all the hash table that you construct.
To put it simply, your program should support inserting some possibly duplicating words with unique integers which represents their original index in the text file (article), deleting some of them and giving the result of the query: "Where is the n^{th} occurence of the word $'w'$?"

3 Implementation of Hash Table

There are three files given to you to use and/or modify in order to save the humankind. As you know Neo, **YOU ARE BORN TO DO THIS!**

A little note for you Neo: These robots can trace your fingerprints in the web. Therefore you have to delete your fingerprints. **DO NOT FORGET to delete anything you got from memory.** They are doing memory check constantly. They can find you via your digital fingerprints that you left on memory.

Before inspecting each method, let's have an overview of our hash table's structure and settings briefly:

- The Article class represents the hash table. The main usage scenario is to find the n^{th} occurence of a given word from a text, in constant expected time
- It uses open addressing, with double hashing technique. If you are not familiar with the concepts, it is highly recomended to get familiar with those implementation decisions of hash table.
- Data type of the table(type of the entries) will be std::pair<std::string, int>, first is for word, second is for word's index on original text.
- Given key value will only include lowercase English letters.
- Cells of the table can be categorized under three different types:
  - Can hold an entry, as described above
  - Can be empty, as initially, meaning that it has not marked as a deleted cell. This means that cell has the value pair std::pair<EMPTY_KEY, EMPTY_INDEX> (values defined in .h file)
  - Can be marked, meaning that it had an entry sometime but the entry's removed from the cell. In this case, cell has the value pair std::pair<EMPTY_KEY,MARKED_INDEX> (values defined in .h file)
- An instance of the Article,
  - Can be given as a text, or requested to insert given words to its table.
  - Can remove entries from its table.
  - Can be requested to find the n^{th} occurence of a given word from a text.

- 'get' operation is requested to return n^{th} occurence of a given word in the original text. It is easy to decide n^{th} occurence of a word from the input text since it is already ordered by means of this operation but since there is no obvious/natural ordering in hash table, you should preserve it while inserting and expanding your table. It is requested that in any of the operations, the cells that you encountered(while probing) should store the index of the original text occurence in ascending order, per word. This means that you can find $< "focus", 121 >$ pair before $< "gregor", 1 >$ since they are different words but by **no** means, probing path of $< "focus", 121 >$ has an element thats original index is less than 121. In other words, if probing path of $< "x", y >$ includes an entry $< "x", z >$, $z < y$ must hold. Fortunately, to have and preserve this property, several lines of code will be enough, hoping that you can find out the lines and where they should be. You are expected to preserve that property by not increasing the time complexity of your program.

3.1 Article.h

This is the only header file that you are given and expected to implement the missing method bodies. The scientist tried to figure out the missing parts. However... None of them has ever accomplished it. In a simpler manner, it is just a hash table implementation that uses an array of std::pair<std::string, int> to store the data which size is table_size
It should look like the following:

```
class Article
{
public:
  // DONT CHANGE PUBLIC PART
  Article( int table_size ,int h1_param, int h2_param );
  ~Article();

  int get( std::string key, int nth, std::vector<int> &path ) const;
  int insert( std::string key, int original_index );
  int remove( std::string key, int nth );

  double getLoadFactor() const;
  void getAllWordsFromFile( std::string filepath );

  void printTable() const;
  // DONT CHANGE PUBLIC PART
private:
  // YOU CAN ADD PRIVATE MEMBERS AND VARIABLES TO THE PRIVATE PART
  std::pair<std::string, int>* table;

  int n; // Total number of current words existing in the hash table
  int table_size ;
  int h1_param;
  int h2_param;

  void expand_table();
  int hash_function( std::string& key, int i) const;
  int h1( int key ) const;
  int h2( int key ) const;

  int convertStrToInt( const std::string &key ) const;

  bool isPrime(int n) const;
  int nextPrimeAfter(int n) const;
```

```
  int firstPrimeBefore(int n) const;
  // YOU CAN ADD PRIVATE MEMBERS AND VARIABLES TO THE PRIVATE PART
};
```

→ One important note to remark is the absence of the copy constructor and the assignment operator overloading functions in Article class.
This two functions are not expected to keep the focus more on implementing an hash table although a correct implementation of a class should have those.
You have a limited time before getting caught by the robots, but remember! Those are very important functions to build a Matrix!
From now on, you should implement these requested functions in the *Article.cpp* file.

3.1.1 Article(int table_size ,int h1_param, int h2_param)

This is your only *constructor.* It gets three initial parameters for *table size, h1 param and h2 param.* You should initialize your table in this function.

- int table_size defines your initial hash table size.
- int h1_param defines a generic parameter to use in the first hash function, namely h1(). Implementation of that function will be explained in further sections.
- int h2_param defines a generic parameter to use in the second hash function, namely h2(). Implementaion of that function will also be explained in further sections.

3.1.2 ~Article()

This is the *destructor* of your *Article* class.

3.1.3 double getLoadFactor() const

This function returns the load factor of the hash table. Where the load factor is calculated by the division of total number of existing words(keys) on the hash table to table_size. Pay attention to the casting.

3.1.4 int h1(int key) const

This is the first hashing function for you to trick the robots. This function has a generic member which is named $h1\_param$, defined in the private section of the *Article* class. Thus, the return value of the function depends on that value. The function does the following,

- Calculates the pop count of the *key*
- Multiplies the pop count value with the generic member $h1\_param$
- Returns the result

But...? What is the pop count? Pop count can be defined as the total number of the $1's$ in the bit representation of an integer, in this case *key*. As a result your pop count function should count the existing $1's$ in the bit representation of the given key(the integer one).

→ **Important note**: This function is going to be used in the main hashing function,
int hash_function(std :: string& key, int i) which does the double hashing.

3.1.5 int h2(int key) const

This is the second hashing function for you to trick the robots. This function also has a generic member which is named *h2_param*, defined in the private section of the *Article* class. Thus, the return value of the function depends on that value. The function does the following,

- Calculates modulo of the *key* by *h2_param*
- Subtracts the calculated value from *h2_param*
- Returns the result

→ **Important note**: This function is going to be used in the main hashing function, `int hash_function( std::string& key, int i)` which does the double hashing.

3.1.6 int hash_function(std::string& key, int i) const

For each operation that is done on hash table, you should hash your *key* values, which are the words in the article in this case, to find/insert/remove the corresponding value properly. This will be done by double hashing in this homework. You should calculate the index of your hash table for that given *key* by calling `int hash_function( std::string& key, int i) const` to calculate the hashed positions. Since we are using open addressing method, if your hashed index is not available(occupied by a different entry etc.), you should call your hash function with same *key* while increasing the second parameter *i* by one. The procedure looks more or less similar as the following:

```
...
// Convert std::string& key to an integer key
// by Article::convertStrToInt
int tableIndex = 0, i = 0;
do
{
  //Do Hashing
  tableIndex = hash_function(key, i);
  i++;

}while( /*calculated tableIndex is not avaible for the operation*/ );
...
```

General approach for this function is the following,

- Convert the `std::string key` to an integer `int key` using `int Article::convertStrToInt( const std::string &key ) const`
- Calculate the integer value, $h1$, returned from `h1(int key)`
- Calculate the integer value, $h2$, returned from `h2(int key)`
- Sum up the first value $h1$ by the multiplication of second parameter i and $h2$
- Return the summation modulo table_size.

3.1.7 int get(std::string key, int nth, std::vector<int> &path) const

This function will be used to retrieve data from your *hash table*. As its **first parameter**, it gets the **"word"** that is being searched, as its **second parameter** it gets an integer value n^{th} and a vector<int>& to save your probing path until you find the requested "word"s n^{th} occurence as its

third parameter.

This function should return original index (in the article text) of n^{th} "word" in the hashing table.
For example in an article, the word "well" repeated two times as 300^{th} word and 450^{th} word. Calling get as your_object.get("well",2,path) will return you 450, that is the original index of the 2nd occurence of key "word" in the hash table.

- The second parameter starts from 1, meaning that the first occurence of a word in the text is requested by passing n^{th} as 1.
- You should return the **original index** of the given "word"s n^{th} occurence on the hash table.
- If the function cannot found the n^{th} occurence of a given word, it returns -1
- In every case, third parameter should contain all of the visited table cells indexes, except for first cell since it is not counted as probing operation.
- In order to save some time to deliver Articles, you have to iterate at most table_size times on the hash table. In other words, you should do at most table_size -1 probes. Actually, if the hashing function works appropriate, this number guaranties that every cell is traversed since the hash table is a bijection to $Z^{0->(ts-1)}$ when the domain is taken in modulo ts where ts is table size
- If you still cannot find the *key*, return -1

3.1.8 int insert (std :: string key, int original_index)

This function will be used for inserting new data to the hash table. It gets std :: string key as its **first parameter** which is the "word" that is to be inserted and an **index** which is the original index of that given "word" as its **second parameter**. Except for a little trick, all you have to do is find an empty slot(of course by using our hash function) for your new entry and insert the new entry there.
So what was that little tricky thing? As you are already mentioned, the ordering of the words in the original text must be directly reflected to the hash table too. For example if there exists three "armadillo" words in text with originally occurs in text as 5th, 17th and 29th words, their order in the hash table should be so that the entry with the lower **original index**(in that case they are 5,17,29) must come first. As you might guess, there is no ordinary order in hash table but, we keep this information by adjusting the table so that for all same words, if one can be found with less **probing count** than the other, its original index must be smaller. By keeping the entries in this way, we can easily give an accurate answer to a request in get function like "what is the original index of third occurence of the word 'armadillo' "
To visualize that behaviour, consider this table and the commands below:

```
    Key,      Original Index
...
<"armadillo",      29>
...
<"armadillo",       5>
...
```

```
// prints 29 since currently, the 2nd "armadillo"s index is 29
std::cout << instance.get( "armadillo", 2, path ); // 29

// insert operation
instance.insert( "armadillo", 17 ); // ..

// now, get returns 17, since currently, the 2nd "armadillo"s index
// becomes 17, remember that original index order and probing
// order must preserved to allow get fucntion to
// return the correct occurence
std::cout << instance.get( "armadillo", 2, path ); // 17

// the one with the original index 29 becomes 3rd
// this get returns 29
std::cout << instance.get( "armadillo", 3, path ); // 29
```

- You must control if the load factor is greater than MAX_LOAD_FACTOR, if it is, you should expand your table **BEFORE** you insert.
- Given key value will only include lowercase English letters.
- The second parameter starts from 1, meaning that the first word of a text will be inserted as 1.
- You should return the probing count, which is the number of trials for finding an appropriate empty spot for your new entry. Don't forget that the first call of hash function should not count as probing.

3.1.9 void expand_table()

Table expansion is a critical point for a hash table with open adressing. If there were no expansion, in an environment that you have no idea how many entries will be stored in your hash table, you may stuck with the new entries that you can not place, or maybe you could allocate so much space in the beginning that you will only use its 0.01 percent.
Expansion solves the majority of these problems with trading needless space consumption and fixedness with time. In every insertion query, if the load factor of your table is greater than MAX_LOAD_FACTOR, you should call this function.
Procedure is rather simpler:

- New table size becomes the smallest prime number greater than 2×(table size)
- $h2\_param$ becomes the biggest prime number less than table size(according to its new value)
- Obvious step: allocate bigger table, insert the ones on the previous etc.

As you can realize again, implementing insert function such that it preserves the order of the same words on the hashing table while inserting helps much on this function.

3.1.10 int remove(std::string key, int nth)

This function is used to delete entries from hash table. In order to determine the which key is going to be deleted, function takes two parameters. First one std::string key specifies the word which is going to be deleted. Second and the last parameter int nth specifies the n^{th} occurence of the key

on the hash table. Return type should be an integer which corresponds the total number of probes to find and delete the word.
To show the behaviour of the remove operation, consider the following settings:

```
      Key,      Original Index
...
<"armadillo",      29>
...
<"armadillo",      5 >
...
<"armadillo",      17>
...
<"neo",            2 >
...
<"convention",     34>
...
```

```
...
// everything is normal, it returns 2, the first "neo"s index
std::cout << instance.get( "neo", 1, path ); // 2

// removes the first neo according to the original order,
// <"neo",2 > is replaced with <EMPTY_KEY,MARKED_INDEX>
instance.remove( "neo", 1 );

// returns -1, since there are no "neo" keys left
std::cout << instance.get( "neo", 1, path ); // -1

...

// currently,3rd occurence of the "armadiilo" is at
// 29th position of the original text
std::cout << instance.get( "armadillo", 3, path ); // 29

// removes the first armadillo according to the original order,
// <"armadillo",5 > is replaced with <EMPTY_KEY,MARKED_INDEX>
instance.remove( "armadillo", 1 );

// this one returns 17, since the one that is previously
// 2nd becomes 1st, old first was deleted
std::cout << instance.get( "armadillo", 1, path ); // 17

// this one returns 29, since the one that is previously
// 3nd becomes 2st, old first was deleted, old third
// is now the Second
std::cout << instance.get( "armadillo", 2, path ); // 29

// this one returns -1, there exists only 2 "armadillo"
// words in the table after the removal
std::cout << instance.get( "armadillo", 3, path ); // -1
```

Actually, remove operation does not have to do anything for the index shifting to happen. After a removal, get operation will not count the deleted key while searching for that key, so naturally, deleting the n^{th} occurence of a key will shift that keys all occurences which are greater than n

Thus, this task asks the followings from you,

- Until you find the correct word (n^{th} on the hash table), do hashing.
- Remove operation does **not** change the original indexes of table entries except the one it removes.
- Remember that if you encounter with a EMPTY_KEY on the hash table while iterating, it does not mean that it is always an empty data. Firstly, you need to check if it is marked or empty. You should behave marked data as a normal data.
- If you find it,
  - To delete the corresponding hash data from the table, you will just **mark** it as deleted. In order to mark a data, you will use pre-defined EMPTY_KEY for the first part of the pair item and MARKED_INDEX for the second part of the pair. These values are defined in the *Article.h* file.
  - After you mark the data, return the total number of probes.
  - Remember that first result of the hashing is not a probe. Probes are the other hashing results that starting from the second try.
- If you cannot find it in table_size iterations, return -1

3.1.11 void getAllWordsFromFile(std::string filepath)

This function will be your first step to trasfer your text file word by word to your hash table. By using included fstream library, you need to read words one by one and insert them with their original indexies in the original text. As a pseudocode,

```
...
//Start from index one

while (/* some condition */)
{
  //Read word by word from the file
  //Insert the word with the current index
  //Increase the index
}
...
```

3.2 Machine.cpp

Before you have started, scientists implemented a primitive machine to calculate these required statistics by hand. They give you an access to that Machine.
The methods that are given in this file are members of the main class, Article. You **shouldn't** change, remove or add anything to that file since it will not be requested from you while submitting. File consists of five function implementations.

```
/* Checks if integer n is prime or not */
bool Article::isPrime( int n ) const

/* Finds the first prime number just after
   integer n */
int Article::nextPrimeAfter( int n ) const
```

```
/* Finds the first prime number just before
     integer n */
int Article::firstPrimeBefore (int n) const

/* Converts given string to an integer that
   is used in hashing functions */
int Article::convertStrToInt( const std::string &key ) const

/* Prints hash table data in a table shaped
     format */
void Article::printTable() const
```

As it is stated, implementations of those functions are available to you.

3.2.1 bool Article::isPrime(int n) const

This function returns true if given parameter n is a prime number.

3.2.2 int Article::nextPrimeAfter(int n) const

This function returns the smallest prime number that is greater than n.

3.2.3 int Article::firstPrimeBefore (int n) const

This function returns the greatest prime number that is less than n.

3.2.4 int Article::convertStrToInt(const std::string &key) const

This function is used on every operation. As you know, in order to seek and find the correct positions in a hash table, we need to **hash** the values (refered as **keys**, words in our case) to some smaller set of values(refered as **indexes**) that are the corresponding positions in the hash table for the keys (where they are stored or will be stored).
Since we are trying to get an index in the hash table for our key, we need to somehow convert our string to integer. For this transformation, you will use this function. It gets a std::string and returns an int to be used in **the hash function**.
You don't have to look at the details of this function.
x

3.2.5 void Article::printTable() const

This function prints the table in a fancy fashioned way. Again, you don't have to have a look at it. One note for your happieness is that it can be good tool for debugging and tracing your program. It can give you an aspect of how the robots can see your transformed article. Also, this function will be playing an important role on grading.

4 Important Notes

- You can use your own input text files. In order to shape them to the form we will use to test your codes, you can use the script we provided for you. Script re-organizes whole file as full of lowercase letters/words and only whitespaces. Proper use of the script is, $\$ > python3\ inputify.py\ your\_text\_file.txt\ >\ new\_text\_file.txt$

5 Regulations

1. **Programming Language:** You will use C++.
2. You cannot use any other library that is not included in the header file by default. In other words, external libraries are not allowed.
3. Memory leaks will be tested by valgrind, on cows. Take care.
4. You can add helper variables and functions of the private part of the definition but you can **not** remove any of the given.
5. You must implement the task using hash table, described as above. Any of the solutions including different implementations will be given 0.
6. **Late Submission:** Every student has a total of 7 days for late submission of the assignments. However, one can use at most 3 late days for this assignment. No penalty will be incurred for submitting within late 3 days. Your assignment will not be accepted if you submit more than 3 days late.
7. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations.

   "Tebrik değil, tevkiftir o. Gidelim, kanundan kaçılmaz." - Erşan Kuneri, GORA

8. **R**emember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
9. **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via Moodle.
- Do not write a *main* function in any of your source files.
- A test environment will be ready in Moodle.
  - You can submit your source files (in this case only *Article.cpp* and *Article.h* files) to Moodle and test your work with a subset of evaluation inputs and outputs.
  - This will not be the actual grade you get for this assignment, additional test cases will be used for evaluation.
  - Only the last submission before the deadline will be graded.