Word Frequency Database

C++

Chelsea Wang

COMP 15

Prof. Tomoki Shibata

Table of Contents

1 Brief Introduction and Getting Started

This *Word Frequency Database* is an application designed to allow users to search the frequencies of words based on a corpus dataset. This specific application was designed based on three datasets created by the Google (Books) Ngram Viewer team:
`American English > Version 20090715 > 1-grams > 0 - 8`
`American English > Version 20090715 > 1-grams > 9`. This application used 3,738,012 records from the dataset files 0-8 and 414,941 from the dataset file 9, to test functionality. The datasets in total are approximately 50MB in size and are available for download here: `http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`.

This application was created under the guidance of Prof. Shibata. The bulk of the requirements and specifications were designated by Tomoki Shibata and Matt Russell for the purpose of a course project at Tufts University. The design and code of the application was written by Chelsea Wang in response to the given project, its requirements, and its specifications.

1.1 Downloading *Word Frequency Database*

The program is available to download to the admins of the Tufts community here: `https://github.cs.tufts.edu/cwang17/comp15/tree/master/project5`. It will later be available for public download here: `https://github.com/chxw/`.

1.2 Files

The files required for running the *Word Frequency Database* program correctly include:

```
Entries.cpp
Entries.hpp
Hash.cpp
Hash.hpp
Node.cpp
```

```
Node.hpp
main.cpp
makefile
```

The following files are additional for the purposes of running tests on this program:

```
test.cpp
time_test.cpp
```

1.3 Compilation

To compile the **database program** correctly, type `make build` which, as defined in the makefile, is effectively:

```
clang++ -std=c++11 Entries.cpp Node.cpp Hash.cpp main.cpp -o database
```

This creates the executable file `database`. To compile the **testing programs** correctly, type `make test` and `make time` for the module testing and time testing respectively.

```
clang++ -std=c++11 Entries.cpp Node.cpp Hash.cpp test.cpp -o test
clang++ -std=c++11 Entries.cpp Node.cpp Hash.cpp time_test.cpp -o time
```

They will create `test` and `time` executable files. For creating executable files for debugging purposes run the above compilation with the following flags included:

- `-Wall` : enable all warnings
- `-Wextra` : enable all extra warnings
- `-g` : generate source-level debug information
- `-O0` : specify no optimization (i.e. this level compiles the fastest and generates the most debuggable code)

2 Using Database Application

2.1 Arguments

The database executable takes precisely one argument. The argument must be the path to a specified dataset file the user would like to deposit into the *Word Frequency Database*. **Requirements**:

1. the argument must be a path to a file,
2. that given file must exist,
3. only one file must be given, and
4. the data set file must be formatted in the following specified way:

```
<word> <tab> <frequency> <newline control>
```

If any of the requirements are not met or any other unexpected errors happen with the given argument, the application prints **stdout** `Error` and terminates. An example of this formatting of the dataset file as taken from the Google Books dataset this program was based on `American English > Version 20090715 > 1-grams > 0 - 9`:

```
zymoses	52
zymosterol	190
zymotics	71
zyrd	65
zyt	129
zyx	70
zyzzyva	48
zz	11573
zz9	56
zzie	62
```

```
    zziii   298
    zzled   50
    zzvi    160
    zzxi    56
    zzzt    71
    zzzzzzzzzzzzz   57

```

An example of inputing an argument correctly to the database executable file:

`./database /comp/15/files/p5/dataset_large.tsv`

If the run is successful, the application will deposit all records in given file to the *Word Frequency Database* and then wait for **stdin** user input.

2.2 User input

If any command given is not the below or does not meet the requirements listed below, the application prints out the error message to **stdout**: `Unknown command` followed by newline control.

2.2.1 `:q`

(**q**)uit. Terminate the application.

2.2.2 `:p word frequency`

(**p**)ut. Add the word and its frequency record to the database.**Requirements:**

1. There must lie a space after `:p` and `word`.
2. `word` cannot contain any spaces.
3. If `frequency` is not a positive integer (including zero), then the application prints out the error message to **stdout**: `Invalid` followed by newline control.
4. If entry is successfully added to the database, the application prints out the confirmation message to **stdout**: `Added` followed by newline control.
5. No duplicate words can exist in the database. If the given `word` exists already in the database, then the application replaces the existing record with the new record.

2.2.3 `:g word`

(**g**)et. Retreive the frequency record that corresponds to the given `word`. If record exists, then the application prints out the following message to **stdout**: `word frequency` followed by newline control. Otherwise, the application prints out the following message to **stdout**: `Not found` followed by newline control.

2.2.4 `:r`

(**r**)emove. Delete the record that corresponds to the given `word`. If the record exists and is succcessfully deleted, the application prints to **stdout**: `Deleted` followed by newline control.If the record does not exist, then the application prints to **stdout**: `Not found`.

3 Design

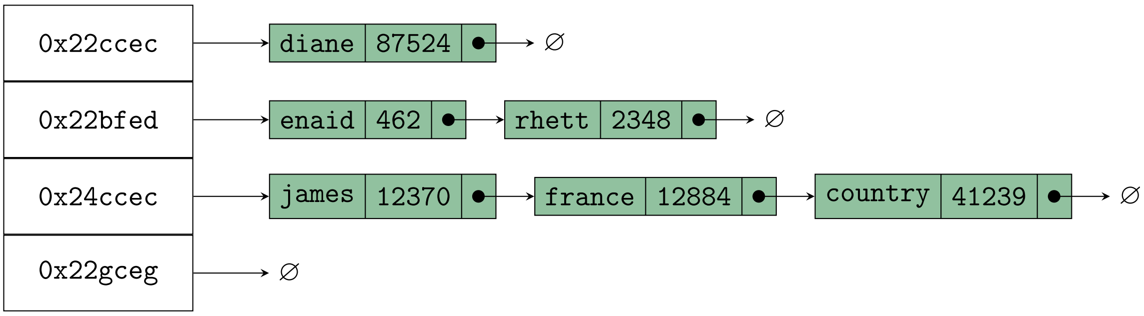
3.1 Overview

The *Word Frequency Database* is implemented using a hash table. A hashing function is used to determine array indices (hash code values) of given words (keys) the user inputs into the database. The table itself is represented by an array of pointers to nodes or "entries" to retrieve any given "record" (word-frequency pair). Records in the application are represented (i.e. stored in memory) as "nodes" in a doubly-linked list. The array indices are used to locate quickly the records in the database. Each word in the database is unique (no duplicates). If the same word is given to the application multiple times with different frequencies, the frequency associated with that word record will update accordingly. If the same word is given to the application multiple times with the same frequency, the application will realize and refuse to add or change anything in the database.

Within each of the nodes are: *word (key) value*, *frequency value*, and *pointers to previous and next node*. The pointers to previous and next node are necessary for "overfill entries", which occur when collisions happen.

3.2 Chaining

In this implementation of a hash table, *chaining* is used for handling collisions. Collisions occur when the hashing function returns the same hash code value (index) of a given key (word). In a hash table with chaining, the solution to the given collision problem is to chain overfill entries to the indices of the array. The below diagram is a representation of how chaining is used to handle collisions.



The left part of the above diagram is an array of pointers, which represents the private variable `table` in the **Hash** class. They point to instances of the class **Entries**. The Entries class is a doubly-linked list of instances of the **Node** class. If there exists overfill entries at an array index, the instance of Entries will have more than 1 node. Otherwise the doubly-linked list will have exactly 1 node. If no entries exist at an array index, the pointer the array holds at that index points to `nullptr`.

3.3 Singly-Linked List vs. Doubly-Linked List

In this implementation of the application, Entries instances hold pointers to both its head and tail (which are not shown in the diagram only for visual clarity). For the purpose of this application, the Entries class could have been implemented with a singly-linked list with only a pointer to only its head. The application would have been just as effective as well as time and space efficient. The original intention of implementing Entries as a doubly-linked list was to eventually improve the time efficiency of the `get` and `remove` functions by implementing a sorted `put` function. With sorted overfill entries, it would be easier to access specific nodes for the purposes of `get` and `remove`. The idea of sorting strings, however, would be a whole other project in its own right, so this implementation was not used for this project.

3.4 Hashing function

The hashing function in this application went through several iterations before it was finalized. The partition was created once I realized multiplication creates more randomness than just summing the int cast of each char in a string. The partition value was chosen at random.

3.4.1 First iteration

```
#include <iostream>
#include <string>

unsigned int generateHashCodeOf(const std::string& key){
        int h = 0;
        int partition = 64; // this was chosen at random

        for (std::string::size_type i = 0; i < key.size(); i++){
                // char->int * partition
                h += (key.at(i) - '0')*partition;
        }

        return h % 16384;
}

```

Using the `hasher` package of tools for testing hashing functions, the collision and time results were as follows:

```
## Unsigned int, base 64
### Collision testing
The number of buckets: 16384
The total number of items: 479828
The length of the longest chain (score): 24341
### Duration
319.684u 0.188s 5:19.88 99.9%   0+0k 0+0io 0pf+0w

```

For mapping 479828 items to 16384 buckets, it was obvious that the above hashing function could be improved to minimize the number of collisions (i.e. lower the length of the longest chain).

3.4.2 Second iteration

This iteration came about when I realized multiplying each char->int value by a different number would create even more randomness. The catch is that the factor each char->int value would be multiplied by would have to be the same each time a string was hashed, i.e. the factor could not be random. I was stuck on this problem, until after doing some light research, I came across the *polynomial hash*. See formula below, where x_n represents the components of the key you're trying to hash and where a^n is some constant.

$$x_{k-1}a^{k-1} + x_{k-2}a^{k-2} + ... + x_0 a^0$$

The below is my first attempt implementation of this hash. I incorporated my own `mod` (modular) function after noticing that the `%` function in C++ will return negative values, see below snippet.

> The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second. If the second operand of / or % is zero the behavior is undefined. For integral operands the / operator yields the algebraic quotient with any fractional part discarded; if the quotient a/b is representable in the type of the result, (a/b)*b + a%b is equal to a.[1]

Hesitant, also, to evaluate an expression with an `int` and `std::string::size_type`, I ran a for loop to derive an `int` value of the length of the string given. Later, I would recognize this as an unnecessary step.

```cpp
#include <iostream>
#include <string>
#include <math.h>

int mod(int k, int n){
  return (k%n + n)%n;
}

int generateHashCodeOf(std::string word){
  int h = 0;
  int length = 0;
  int base = 33;

  for (std::string::size_type i = 0; i < word.size(); i++){
    length++;
```

[1] [1, 235]

```
    }

    for (int i = 0; i < length; i++){
      h += (word.at(i) - '0') * pow(base, length - (i+1));
    }

    return mod(h, num_buckets);
  }

```

Using the `hasher` package of tools for testing hashing functions, the collision test results were as follows:

```
## int + user-designed mod, base = 33
The number of buckets: 16384
The total number of items: 479828
The length of the longest chain (score): 2736

```

After the results of this collision test, it was decided that this improved roughly one order magnitude from the first iteration.

3.4.3 Third iteration

In this iteration, I removed the `mod` function after realizing that I could assign the return value of the hashing function to be an `unsigned int`, which effectively "wraps around" if it is assigned a negative value below `INT_MIN` or a positive value above `INT_MAX`, preventing undefined behavior in the function. In this iteration, I also realized I could evaluate an expression between two values of `std::string::size_type`. Since, no other changes were made outside these, I continued testing this iteration of function with varying base values (i.e. I stopped testing with the second iteration of the hashing function).

```
#include <iostream>
#include <string>
#include <math.h>

unsigned int Hash::hasher(std::string word){
  unsigned int h = 0;
  unsigned int base = 33;

  for (std::string::size_type i = 0; i < word.size(); i++){
    h += (word.at(i) - '0') * pow(base, word.size() - (i+1));
  }

  return h % num_buckets;
}

```

Using the `hasher` package of tools for testing hashing functions, the collision test results were as follows:

```
## Unsigned int, base = 33
### Collision testing
The number of buckets: 16384
The total number of items: 479828
The length of the longest chain (score): 60
### Duration
4.038u 0.099s 0:04.14 99.5%     0+0k 0+0io 0pf+0w

```

After the results of this collision test, it was decided that this, again, drastically improved from the second iteration.

3.4.4 Fourth iteration

```
#include <iostream>
#include <string>
#include <math.h>

unsigned int Hash::hasher(std::string word){
  unsigned int h = 0;
  unsigned int base = 37;

  for (std::string::size_type i = 0; i < word.size(); i++){
    h += (word.at(i) - '0') * pow(base, word.size() - (i+1));
  }

  return h % num_buckets;
}

```

Using the `hasher` package of tools for testing hashing functions, the collision test results were as follows:

```
## Unsigned int, base = 37
### Collision testing
The number of buckets: 16384
The total number of items: 479828
The length of the longest chain (score): 67
### Duration
4.024u 0.103s 0:04.13 99.7%    0+0k 0+0io 0pf+0w

```

The collision and duration tests were executed multiple times for the third iteration and the fourth iteration with consistent results, so it was decided that the third iteration of the hash function provided hashing with the least amount of collisions thus far.

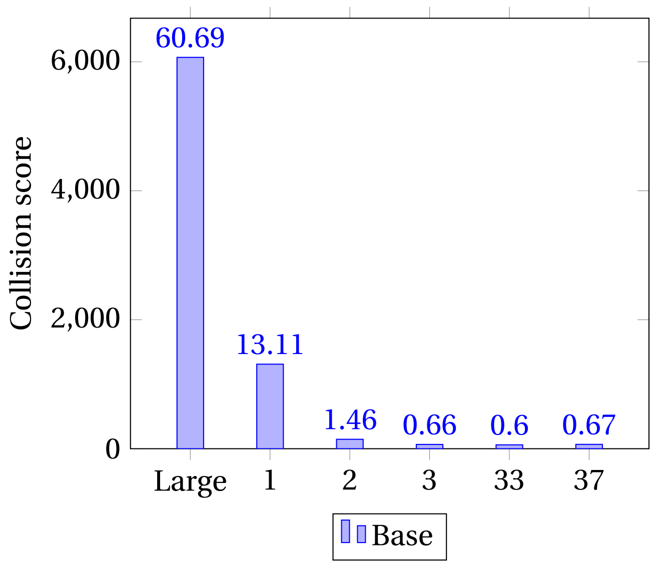
3.4.5 Further testing

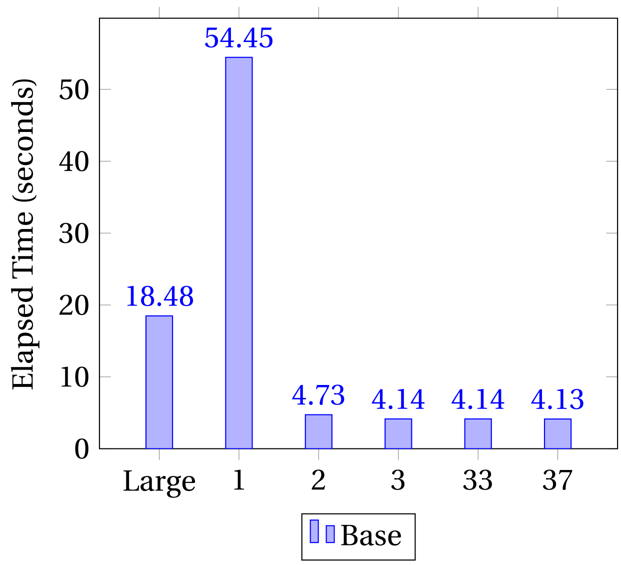
Further testing was done to see if there would be better base values for the hashing function, but it seemed that a base value of 33 provided the best collision and time test results. The other base values that were tested included: a random large prime number, 1, 2, 3, 33, 37.

```
## Unsigned int, base 3
### Collision testing
The number of buckets: 16384
The total number of items: 479828
The length of the longest chain (score): 66
### Duration
4.030u 0.108s 0:04.14 99.7%    0+0k 0+0io 0pf+0w

## Unsigned int, base 2
### Collision testing
The number of buckets: 16384
The total number of items: 479828
The length of the longest chain (score): 146
### Duration
4.474u 0.104s 0:04.73 96.6%    0+0k 6808+0io 3pf+0w
```

```
## Unsigned int, base 1
### Collision testing
The number of buckets: 16384
The total number of items: 479828
The length of the longest chain (score): 1311
### Duration
54.316u 0.124s 0:54.45 99.9%    0+0k 0+0io 0pf+0w

## Unsigned int, base random large prime number
### Collision testing
The number of buckets: 16384
The total number of items: 479828
The length of the longest chain (score): 6069
### Duration
18.326u 0.115s 0:18.48 99.7%    0+0k 0+0io 0pf+0w
```

Below is a visualization, comparing the collision score and time test results of the different bases. It was surprising that a random large prime number caused such a high (poor) collision score whereas 3 caused such a score comparable to the best (3, 33, 37). I assumed that 1 would cause the worst score, but there seems to be some benefit to lower prime numbers. Perhaps it's an issue of undefined behavior that the "wrap-around" behavior of an `unsigned int` could not solve. It's also interesting to note that between the two smallest prime numbers (2 and 3), 3 performed significantly better as a base value.





4 Classes and Methods

Many of the modules used in this program have methods or variables that aren't explicitly used for the *Word Frequency Database* application as it currently is. However, they were kept in the code for the purpose of adding or improving functionality of the application.

4.1 Node class

This class represents the "records" of the database. Its main purpose is to hold data and to be easily located by pointers. Each node holds (privately) one `std::string` *word*, representing the word of the record, and one `int` *freq*, representing the frequency of the record. If an instance of node is not holding anything, *word* is an empty string, and *freq* is `-1`. Each node also holds (privately) pointers to their next and previous node. If the node has no next or previous node (i.e. it's isolated), it's *next* and *previous* pointers point to `nullptr`. Node is able to define and retrieve its *freq* value, *word* value, *next* pointer, and *prev* pointer via its `set` and `get` functions.

4.1.1 Hpp file

The below is the `.hpp` file for the class.

```cpp
#ifndef NODE_HPP
#define NODE_HPP

#include <string>

class Node{
public:
  Node();
  Node(std::string given_word, int given_freq);
  Node(const Node& other);
  Node& operator=(const Node& other);
```

```
    ~Node();

    std::string getWord() const;
    void setWord(std::string w);
    int getFreq() const;
    void setFreq(int f);

    Node* getNext() const;
    void setNext(Node* node);
    Node* getPrev() const;
    void setPrev(Node* node);

  private:
    std::string word;
    int freq;

    Node* next;
    Node* prev;
};

#endif

```

4.1.2 Default constructor

```
Node::Node()
```

Initializes an instance of Node with *next* pointer and *prev* pointer set to `nullptr`, *word* value set to empty string `""` and *freq* value set to `-1`.

4.1.3 Parametrized constructor

```
Node::Node(std::string given_word, int given_freq)
```

Initializes an instance of Node. Sets *word* value to `std::string given_word` and *freq* value to `int given_freq`. Sets *next* and *prev* pointers to `nullptr`.

4.1.4 Copy constructor

```
Node(const Node& other);
```

Copies information over from other.

4.1.5 Assignment operator

```
Node& operator=(const Node& other);
```

Throws `std::runtime_error`: "Not Implemented".

4.1.6 `getWord`

```
std::string getWord() const
```

A `const` function that retrieves private `std::string` variable *word* value a node is holding. If it's not holding any word, it returns `""`.

4.1.7 `setWord`

```
void Node::setWord(std::string w)
```

Sets the private variable *word* value to given `std::string w`.

4.1.8 `getFreq`

```
int Node::getFreq() const
```

A `const` functions that returns private `int` variable *freq* value.

4.1.9 `setFreq`

```
void Node::setFreq(int f)
```

Sets the private variable *freq* value to given `int f`.

4.1.10 `getNext`

```
Node* Node::getNext() const
```

Returns private variable `Node` pointer *next*.

4.1.11 `setNext`

```
void Node::setNext(Node* node)
```

Sets the private variable `Node` pointer *next* to given `Node* node`.

4.1.12 `getPrev`

```
Node* Node::getPrev() const
```

Returns private variable `Node` pointer *prev*.

4.1.13 `setPrev`

```
void Node::setPrev(Node* node)
```

Sets private variable `Node` pointer *prev* to given `Node* node`.

4.2 Entries class

The Entries class is a doubly-linked list (DLL) made of instances of the **Node** class. An instance of Entries holds two Node pointers, *head* and *tail*, which is used to keep track of its DLL. Entries uses its public methods: `add`, `findFreq`, `updateFreq`, and `remove` to update, modify, and keep track of it's DLL. It's `add` function is used for adding an instance of **Node** to its DLL via Node pointer. It's also possible to add with just *word* and *frequency* values, where Entries will create and initialize a new Node on it's own and add to its DLL. The `findFreq` method acts as a "get" method. It is used to retrieve the corresponding *frequency* to a *word*. Since words are unique but frequencies are not (i.e. different words can have the same number of frequencies), the Entries class only has a get method for the frequency of a word and does not have a get method for the word of a frequency. In other words, it would not make sense for the Entries class to search its DLL by frequency, since there may be multiple records of the same frequency. The method `updateFreq` is used to change the frequency of a record in the list.

4.2.1 Hpp file

The below is the `.hpp` file for the class.

```cpp
#ifndef ENTRIES_HPP
#define ENTRIES_HPP

#include <string>

#include "Node.hpp"

class Entries{
public:
  Entries();
  Entries(std::string word, int freq);
  Entries(const Entries& other);
  Entries& operator=(const Entries& other);
  ~Entries();

  bool add(std::string w, int f);
  bool add(Node* newbie);
  int findFreq(std::string word);
  void updateFreq(std::string word, int newFreq);
  bool remove(std::string w);

  Node* top();
  Node* bottom();

  std::string toString() const;

private:
  int num_entries;

  Node* head;
  Node* tail;
};

#endif

```

4.2.2 Default constructor

```cpp
Entries::Entries()
```

Initializes an instance of Entries with *head* pointer and *tail* pointer set to `nullptr`, *num_entries* `int` value set to `0`.

4.2.3 Parametrized constructor

```
Entries::Entries(std::string word, int freq)
```

Initializes an instance of Entries. Creates one new instance (on heap) of Node. Sets the instance of Entries's *head* and *tail* pointers to the new node. This represents an entry in the database that exists in the hash table array with no overfill entries yet.

4.2.4 Copy constructor

```
Entries(const Entries& other);
```

Throws `std::runtime_error`: "Not Implemented".

4.2.5 Assignment operator

```
Entries& operator=(const Entries& other);
```

Throws `std::runtime_error`: "Not Implemented".

4.2.6 Destructor

```
Entries::~Entries()
```

Traverses down the DLL that the instance of Entries holds deleting each heap-allocated instance of Node. Sets *head* and *tail* pointers to `nullptr` at the end to prevent any attempts to re-access data where memory was freed.

4.2.7 `add` - create new Node

```
bool Entries::add(std::string w, int f)
```

Initializes a new heap-allocated instance of Node (via Node's parametrized constructor) using given `std::string w` and `int f`. Adds new node to the end of DLL (i.e. *tail* pointer now points to this node). Increments `num_entries` by 1.

4.2.8 `add` - with given Node

```
bool Entries::add(Node* newbie)
```

Adds given `Node* newbie` to the end of DLL (i.e. *tail* pointer now points to this node).

4.2.9 `findFreq`

```
int Entries::findFreq(std::string word)
```

Traverses DLL searching for node that holds *word* value that matches `std::string word`. If no nodes are found, the function returns `-1`. Since no entries with a non-positive, non-zero *freq* value can exist in Entries, the application recognizes a `-1` value return from this function to mean that no node was found.

4.2.10 `updateFreq`

```
void Entries::updateFreq(std::string word, int newFreq)
```

Checks to see if `int newFreq` is zero or positive and if it is not either, the function terminates. Traverses DLL searching for node that holds *word* value that matches `std::string word`. If no nodes are found, the function does nothing (changes/modifies nothing). If a node is found, the function updates that entry's *freq* value with `int newFreq`.

4.2.11 `remove`

```
bool Entries::remove(std::string w)
```

Traverses DLL searching for node that holds *word* value that matches `std::string w`. If a node is found, the function removes the node from DLL and returns `true`. Otherwise, it returns `false`.

4.2.12 `top`

```
Node* Entries::top()
```

Returns *head* Node pointer.

4.2.13 `bottom`

```
Node* Entries::bottom()
```

Returns *tail* Node pointer.

4.2.14 `toString`

```
std::string Entries::toString() const
```

Returns `std::string` representation of the instance of Entries's DLL. An example of a return value for this function would be:

```
        Grosseltern<->Lancef<->FALDA<->recognitionem

```

4.3 Hash class

4.3.1 Hpp file

The below is the `.hpp` file for the class.

```
#ifndef HASH_HPP
#define HASH_HPP

#include <string>

#include "Entries.hpp"

class Hash{
public:
  Hash();
  Hash(int buckets);
  Hash(const Hash& other);
  Hash& operator=(const Hash& other);
  ~Hash();

  void put(std::string word, int freq);
  int get(std::string word);
  bool remove(std::string word);

  float getLoadFactor();

  void print();

private:
  Entries** table;

  bool setNewFreq(std::string word, int newFreq);

  unsigned int hasher(std::string word);

  void resize();
  bool isPrime(int number);
  int nextPrime(int number);

  int num_words;
  int num_buckets;
  float threshold;
};

#endif
```

4.3.2 Default constructor

```
Entries::Entries()
```

Initializes an instance of Hash. Initializes private variables: sets `num_words` to `0`, sets `num_buckets` to `199` (default Hash *table* size), sets `threshold` to `0.75f`. Initializes and allocates memory for an array (of size `num_buckets` of pointers to instances of Entries. Iterates over private variable *table* array, setting each index value to `nullptr`.

4.3.3 Parametrized constructor

```
Hash::Hash(int buckets)
```

Initializes an instance of Hash of give size `int buckets`. Initializes private variables: sets `num_words` to `0`, sets `num_buckets` to `int buckets`, sets `threshold` to `0.75f`. Initializes and allocates memory for an array (of size `num_buckets` of pointers to instances of Entries. Iterates over private variable *table* array, setting each index value to `nullptr`.

4.3.4 Copy constructor

```
Hash(const Hash& other);
```

Deep copies information from other.

4.3.5 Assignment operator

```
Hash& operator=(const Hash& other);
```

Deep copies information from other.

4.3.6 Destructor

```
Hash::~Hash()
```

Iterates over array calling Entries destructor at each index if the index is not empty (pointing to `nullptr`. Then free memory allocated for *table* array.

4.3.7 `put`

```
void put(std::string word, int freq)
```

Checks to see if the load factor of the table (`num_words / num_buckets`) has passed the *threshold*. If it has, function calls private function `resize` to resize and rehash table. Checks to see if the word already exists in the table. If it already exists, the record's *freq* will be updated with `int freq` and terminate. If it doesn't already exist, function hashes `std::string word` and uses the hash code value it generates to insert the entry into the table. Increments `num_words` by `1`.

4.3.8 `get`

```
void get(std::string word)
```

Hashes `std::string word` and uses the hash code value it generates to check if there exists entries at that index in table. If there does not exist entries at that index, the function terminates and returns `-1`. Otherwise, if there exists an instance of Entries at that index, the function calls the `findFreq` function in Entries.

4.3.9 `remove`

```
bool remove(std::string word)
```

Hashes `std::string word` and uses the hash code value to check if there exists an instance of Entries in *table* at that index. If there does not exist an instance, the function terminates and returns false. Otherwise, if there does exist an instance, the function calls that instance of Entries's `remove` function. If Entries's `remove` function returns `true`, `num_words` decrements by `1` and the function returns true. Otherwise, the function returns false.

4.3.10 `getLoadFactor`

```
float getLoadFactor()
```

Divides `num_words` by `num_buckets` and returns `float` value result.

4.3.11 `toString`

```
std::string Hash::toString()
```

Returns `std::string` representation of the instance of hash table (with overchain entries chained included). An example of a return value for this function would be:

```

        6683005: MANCIA<->Moredock
        6683006: nullptr
        6683007: nullptr
        6683008: slopeof<->Publithed
        6683009: nullptr
        6683010: Pushpa<->Seabrooke's
        6683011: orster<->lpeak
        6683012: nullptr
        6683013: nullptr
        6683014: nullptr
        6683015: nullptr
        6683016: MANCIL
        6683017: Lancin
        6683018: Lancio<->Publithen
        6683019: nullptr
        6683020: nullptr
        6683021: nullptr
        6683022: Lancis<->Lane's<->Publither
        6683023: Lancit
        6683024: MOPSUESTIA

```

4 References

[1] *Modulo operator with negative values [duplicate].*
https://stackoverflow.com/questions/7594508/modulo-operator-with-negative-values

[2] *when to resize a hash table?*
https://stackoverflow.com/questions/4959466/when-to-resize-a-hash-table

[3] *CS 3137 Class Notes.*
http://www.cs.columbia.edu/~allen/S14/NOTES/hash.pdf

[4] *What is a polynomial hash function for a string?*
https://www.quora.com/What-is-a-polynomial-hash-function-for-a-string

[5] *function: pow.*
http://www.cplusplus.com/reference/cmath/pow/

[6] *Best way to resize a hash table*
https://stackoverflow.com/questions/22437416/best-way-to-resize-a-hash-table

[7] *Why are hash table expansions usually done by doubling the size?*
https://stackoverflow.com/questions/2369467/why-are-hash-table-expansions-usually-done-by-doubling-the-size

[8] *CS240 – Lecture Notes: Hashing*
https://www.cpp.edu/~ftang/courses/CS240/lectures/hashing.htm

[9] *Doubly linked list*
https://en.wikipedia.org/wiki/Doubly_linked_list

[10] *c# - Mod of negative number is melting my brain*
`https://stackoverflow.com/questions/1082917/mod-of-negative-number-is-melting-my-brain`

[11] *Hash table. Dynamic Resizing*
`http://www.algolist.net/Data_structures/Hash_table/Dynamic_resizing`

[12] *Can I use 2 or more delimiters in C++ function getline? [duplicate]*
`https://stackoverflow.com/questions/37957080/can-i-use-2-or-more-delimiters-in-c-function-getline`

[13] *std::basic_stringstream*
`https://en.cppreference.com/w/cpp/io/basic_stringstream`

[14] *std::vector::operator[]*
`http://www.cplusplus.com/reference/vector/vector/operator[]/`

[15] *Finding the type of an object in C++*
`https://stackoverflow.com/questions/351845/finding-the-type-of-an-object-in-c`

[16] *How to convert string to in and int to string in C++*
`https://www.systutorials.com/131/convert-string-to-int-and-reverse/`

[17] *Reading delimited files in C++ [duplicate]*
`https://stackoverflow.com/questions/1075712/reading-delimited-files-in-c`

[18] *Read tab separated file into array in c++*
`https://stackoverflow.com/questions/19989727/read-tabs-separated-file-into-arrays-in-c`

[19] *How can I read and parse CSV files in C++?*
`https://stackoverflow.com/questions/1120140/how-can-i-read-and-parse-csv-files-in-c`

[20] *read comma delimited text file into an array*
`http://www.cplusplus.com/forum/general/17771/`

[21] *Tokenizing a string in C++*
`https://www.geeksforgeeks.org/tokenizing-a-string-cpp/`

[22] *Parse (split) a string in C++ using string delimiter (standard C++)*
`https://stackoverflow.com/questions/14265581/parse-split-a-string-in-c-using-string-delimiter-standard-c`

[23] *C++ how to use fstream to read tab-delimited file with spaces*
`https://stackoverflow.com/questions/40337177/c-how-to-use-fstream-to-read-tab-delimited-file-with-spaces`

[24] *std::getline*
`https://en.cppreference.com/w/cpp/string/basic_string/getline`

[25] *Fastest way to check if a file exist using standard C++/C++11/C?*
`https://stackoverflow.com/questions/12774207/fastest-way-to-check-if-a-file-exist-using-standard-c-c11-c`

[26] *Linux Time Command*
https://linuxize.com/post/linux-time-command/

[27] *Function to check if string contains a number*
https://stackoverflow.com/questions/9642292/function-to-check-if-string-contains-a-number

[28] *How Can I Check if a string represents an positive non-zero int?*
https://stackoverflow.com/questions/4625362/how-can-i-check-if-a-string-represents-an-positive-non-zero-int

[29] *Project 5: Word Frequency Database*
https://www.cs.tufts.edu/comp/15/r/p5.pdf