MUHAMMED YUSUF MERMER – CMP2003 TERM PROJECT

PROJECT REPORT

This report contains "Reading from File", "Why Hash?", "What has been used for hash functions?", "Why min Heapify?" and "Final Results" as sections. Looking at "Why Hash?" part is especially recommended.

Reading From File:

```
ifstream otherFile(R"(C:\Users\myusu\Downloads\PublicationsDataSet.txt)",
ifstream::in);
	string line;

	while (otherFile) {//different lines
		std::getline(otherFile, line);
```

Each document's metadata information written on different lines, so getline function used to get data line by line to string. And until we extract last line (means extracting last document's metadata) loop will continue to work

```
int cur = line.find("unigramCount", 482); //start to search from 482 because the
nearest unigramCount starts from that point
		//+14
		int curLoc = cur + 15; //pass words of unigramCount and "{"
```

Program must not count words before the "unigramCount" word, so at first, program needs to find unigramCount in string. And to be faster, with the help of another program (which is also coded by me) looked where unigramCount started in each documents' metadata and saw that the one that closest to the beginning of the line starts from 482nd character, so the searching skips first 481 characters and starts from 482nd character to find unigramCount. After that, because implementation uses another loop for taking pairs (word and iteration time), program passes characters of "unigramCount" and "{".

Before the loop, to understand loop better please look "Definition of word" part.

Definition of word:

Any contiguous character from a-z with at most contains one " ' " between them considered as a word. So, no other characters must be between a-z characters. If they do, the program will not count them as a word. If these other characters are at the end or beginning, simply program will not count them. But also another condition considered: program needs to split words when it sees "-".

transport-related	➔	transport related
transport-/related	➔	(not considered as word)
non-data-related	➔	non data related
transport-related23.	➔	transport related
transport-rel&ated	➔	(not considered as word)

```
while (!isalpha(line[curLoc]) && (line[curLoc] != '\"' || line[curLoc + 1] != ':' ||
!isdigit(line[curLoc + 2]))) // go until find a-z
                              curLoc++;
bool r = Y.dream1(line, str1, str2, str3, curLoc);
```

As a stopping condition for word, program needs to see double quotation mark, colon, and an arbitrary digit in the sequential order. Used 3 elements in order, because there are some exceptions such as colon and another double quotation marks in between double quotation marks.

Program reads the first pair (to find word) until it finds alphabetic character. If cannot find, it will be empty (look at dream1 function). If it finds an alphabetic character:

```
bool dream1(string& line, string& str1, string& str2, string& str3, int& curLoc) {
//first word determiner until first "-"
              while ((isalpha(line[curLoc]) || line[curLoc] == '\'') && (line[curLoc]
!= '\"' || line[curLoc + 1] != ':' || !isdigit(line[curLoc + 2])))
              {
                     str1 += tolower(line[curLoc]);
                     curLoc++;


              }
              if ((line[curLoc] != '\"' || line[curLoc + 1] != ':' ||
!isdigit(line[curLoc + 2]))) {
                     if (line[curLoc] == '-') {
                            if (isalpha(line[curLoc + 1])) {
                                   curLoc++; return dream2(line, str2, str3, curLoc);
                            }
                            else {
                                   return alphaChecker(line, curLoc);
                            }
                     }
                     else {
                            return alphaChecker(line, curLoc);
                     }
              }
              return 1;
}
```

If being word condition satisfied it will return 1 and if not satisfied return 0 (look at if(!r) statement). Everything has been taken as references because changes on the formal parameters must also affect to actual parameters, and also line variable contains a long string, so copying a whole line will be time costly.

Everything converted to lower-case and added to a string until there isn't any alphabetic character left. Then; if first pair finished, return. If not finished; in the current location, there must be different character, if this character is not "-", then look whether there is any other alphabetic character after current location (alphaChecker function). If there is an alphabetic later, return 0; if not, returns 1. If this character is "-", look whether next character is alphabetic. If not use alphaChecker function again. If yes, then return to dream2 function which looks like the dream1 function.

```
if (!r) str1 = str2 = str3 = "";
```

If dream returns 0, then being word condition is not satisfied so program needs to make strings ="".

```
while (line[curLoc] != ',' && line[curLoc] != '}') { //take all digits
                              num *= 10;
                              num += line[curLoc] - '0';
                              curLoc++;
```

```
}
```

Takes string digits and convert it to int, then puts to num.

Then if the strings are not empty, send them to generalHashPutter function.

Program continues to do these operations until "}" (the end of the line)

Why Hash?:

Hashes have O(1) time complexity, so it is very fast especially when processing big data. In this program, they are used for storing stopping words and words from the Publications Data Sets.

For speed efficiency, linked list style close addressing unordered maps are not that much efficient, so both stop words and counting words are stored as probing sequence. But because they are used in different ways, their probing sequences are also different. For counting words, used Hopscotch Hashing.

Hopscotch Hashing:

Hopscotch hashing is an efficient algorithm originally proposed by Maurice Herlihy, Nir Shavit and Moran Tzafrir. It guarantees that with the small number of lookups, we can do insertion, search, and deletion operations. How does it work?:

Home bucket is a location where the item is intended to be put if that location is empty. Neighbourhood used for the distance between the home bucket of the item and furthest location that item can be put from the home bucket. Each home bucket has a H neighbourhood.

Let's assume H=3, then:

0	1	2	3	4	5	6
x	y		a	b		

If we want to add new item, whose home bucket is 0, then start from 0 and try to find place until 0(home bucket) + 3(number of neighbourhood) =3. Location 2 is empty so put there.

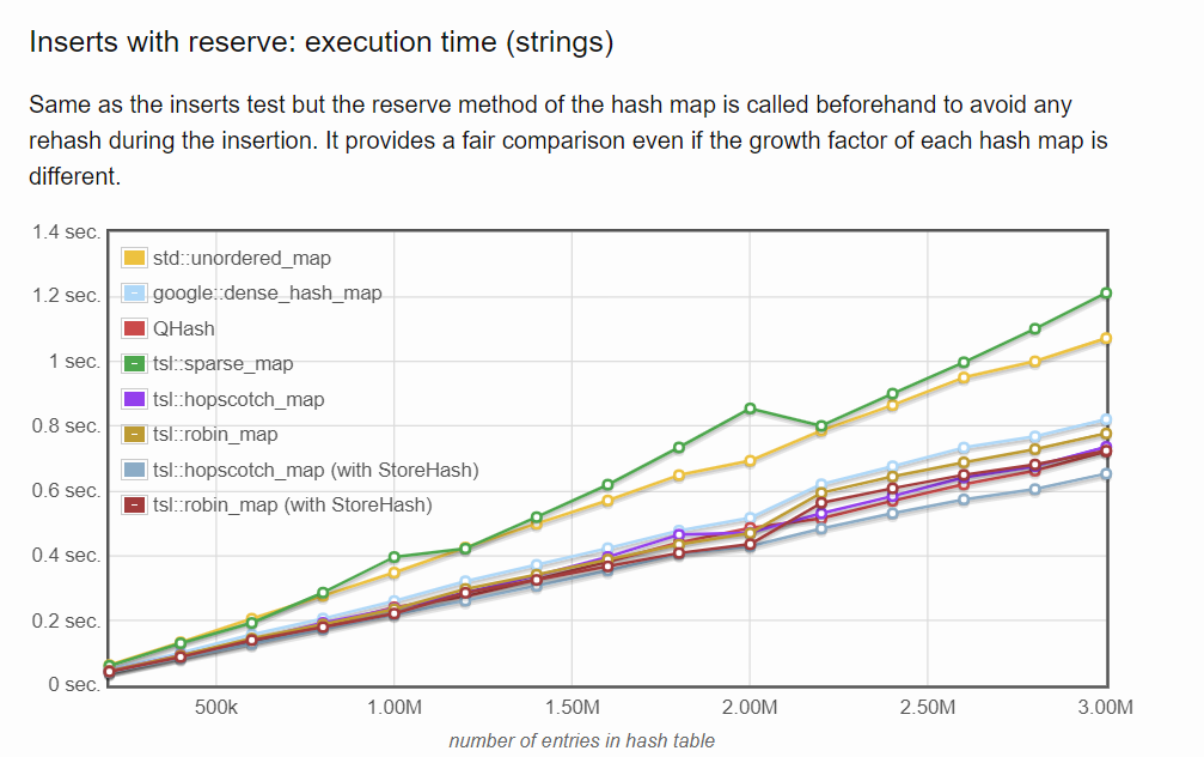
0	1	2	3	4	5	6
x	y	z	a	b		

Again, our new item's home bucket is 0, but for this time its all neighbourhoods are full. Therefore, after understanding that, find the nearest empty place (which is 5). Then start to look from home bucket 0 and find the first item that can swap with location 5. For this example, let's assume 0 is home bucket of items x, y, z and 3 is home bucket of items a and b. So, after passing x, y and z; program will see that item a can swap from 3 to 5. Therefore, program swaps values of locations 3 and 5.

0	1	2	3	4	5	6
x	y	z		b	a	

The new item can be inserted to location 3.

It is quite efficient for counting words, also because while program is searching for a word or an empty place when it finds an empty place first, this means that word hasn't been used yet and you can put the word to that location.



Given graph is from https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html

But after insertion is finished, if we want to search for a specific item, we need to search it on H items, which will be costly. Program is not doing an extra search for counting words, however, in the hash of stopping words, after insertion operations, program does lots of searches to understand whether the given word is in the stop words or not. And this may result in inefficiency.

When the word doesn't exist on stopping words' hash, that hash shouldn't be iterated too many times to search a word (for hopscotch, constantly H times). On the other hand, at the link list style close addressing, program can be maximumly iterated as the number of collisions that home bucket has; and this may prevent unnecessary searches. However, also using linked lists with close addressing is not efficient when it compared to array probing because arrays are faster in passing to the new element. So come up with mixed of these two:

Hash for Stopping Words:

The hash of each element contains two parts: the word and the next item's place. Initially, all next item's place values are -1, which means there isn't any other collusion.

In insertion operation, after program finds the location of the new item that want to be inserted, if that location is empty, then puts word to there; if not empty, then, the program enters to loop until the next item's place value becomes -1. After this value becomes -1, program stores the latest location (stored in storeIndex) and looks location=location+7 iteratively (7 preferred because 7 is a prime number, so the sum would be much unique) until it finds an empty place. When the program finds an empty place, puts the word over there and changes storeIndex location's next place value to this brand-new location. This might be a little bit complicated, but it is for the worst case of the insertion, which did not occur much.

In search operation, program starts from the home bucket and if the word is in the current location is not the same word that we are searching for, looks next item's place value and goes to that location, then looks at whether the searched word is in that location. If not, does the same operations until the next item's place value becomes -1. If it becomes -1, then it means program shouldn't continue to search. So as in the close addressing of the linked list, it will only iterate as the number of collisions

What has been used for hash functions?:

In practice, multiplication with prime number gives more unique results when the program looks remainder by a specific number. And bitwise operators' calculations are also faster compared to other types of mathematical operators'.

Why min Heapify?:

In this program, min Heapify shows the word which has the smallest frequency among the other 14 words without sorting array completely. Therefore, in every iteration of insertion operation, after the hash operations; first, program checks whether the word is already in heap or not, if yes then program adds only new counted iteration time comes from the Publications Data Set to the location of the word in the heap and does heapify operation staring from this word's location. If the word is not on the heap, then after insertion operation, program looks at whether the frequency of the word is bigger than the frequency of the smallest frequented word in the min heapify array. If the current word is bigger, then change places of smallest one and current word and do heapify operation.

The array of heapify stores 15 elements, so heapify operations at each step will be equal to a constant number. And these operations will be done totally n times, so time complexity would be approximately O(n). However, if program does heapify at the end of hashing, then the total time complexity would be approximately O(n*log n). Therefore, sorting at every iteration of the insertion is faster.

Final results: