CMP2003 DATA STRUCTURES AND ALGORITHMS PROJECT

Hilmi Mert Acar 2004285;

Birgül Öztürk 1802819;

Berkcan Ürüm 1901891;

Our project is about reading and counting unique words used in documents and finding top 10 frequent words used in these documents.

PublicationsDataSet.txt file was given to us which includes 1500 different publications. There was also another file stopwords.txt which included many common words that were not to be counted.

We mainly used an open addressing hashtable with quadratic probing for collisions to store the words for our project this hashtable is dynamic so it can grow if needed.

We also used a LinkedList for storing the data before using it.

The execution time of our code is between 650-700 milliseconds, depending on the processor.

In PublicationsDataSet.txt, in every line the data we are interested in starts after "unigramCount": so, using this we for every line we extracted the data and put it inside of our LinkedList.

For dissecting the data we created two size_t variables named wordSTARTSearch and wordENDSearch starting from 0 searched for where the combination of ":number," is in the line and equaled wordENDSearch to that position and equaled wordSTARTSearch to the old value of wordENDSearch. Thus if we start writing from wordSTARTSearch to wordENDSearch we get something like:

"others

":1,"air

":1,"networks,

Etc.

Every word's corresponding number starts 3 positions in front of it. And before every word there is the previous word's number except of the first word of every line. Knowing these we extracted the words and their numbers. After this we deleted the non-alphabetical and non-number characters from the start and the end of our word. If our word wasn't wholly made up of non-alphabetical and non-number characters, we then checked if they were fit to be added to our hashtable using the .shouldAddWord() function. Our criteria were if there were any non-alphabetical characters inside the word except only one ' . This function is in our hashtable.h file.

Next, we removed every word in the stopfile.txt from our hashtable and using the fastertop10 function we find the 10 most used words in the PublicationsDataSet.txt and write them down.

We also write down the time elapsed since the start of the code.

Our hashtable and linkedList are .h files. Our linked list can store any type of variables. It has 7 functions. .insert(i), .remove(deletedata),.removeAtPosition(), .at(), .getCurrentAndMove(), .getCurrent(), .resetCurrent(). Not all these functions are used but they all work. Using .getCurrentAndMove instead of .at was more efficient for our specific case since we need use the lines iteratively. This linkedList implementation has 2 extra variables than a basic linkedList implementation. These are a pointer current pointing to a node, and currentPosition keeping track of where this node is.

Using .getCurrentAndMove() we get the value of the current node and the current moves one position forwards.

Using .getCurrent() we get the value of the current node, but the current doesn't move.

Using .resetCurrent() sets the current to the first node.

Using .at(i) gets the ith item in the linkedList but if the i is iteratively increasing using current is faster.

Using .remove(deletedata) removes the first node where node->data equals deletedata.

Using .removeAtPosition(i) removes the ith node.

Using .insert(data) inserts data on the back of the linkedList.

Before our hashtable implementation we create a structure named wordCount. This struct is made up of a string and an integer. Our hashtable is made up of a pointer named table pointing to an array of wordCount. This arrays length is asked when the hashtable is called and stored in size. The length keeps track of how many spots on the array is taken. At first all spots on our table are filled with "" and -1.

In our insert function we first check if length+1 is %90 of size. If so we double the size of our hashtable and reinsert every item inside our hashtable.

Next our insert function changes the word being inserted all into lowercase letters and calculates the words hashindex. To calculate our hashindex for every character in our word we add its ascii value squared times that characters position in the word plus 3.

After that we check the spot corresponding to hashindex%size. If the string value there is the same as the word we are inserting we add the number value to that spot. If the spot is empty we place our string and value into that position. If it is neither we increase our probingItteration and check (hashindex+(probingItteration)^2)%size. We do up until this until we can insert the item or probingItteration is the same as size.

Our remove function checks every position in our hashtable and if the word we are trying to remove is found it is removed and stops.

Our top10 function creates a copy of our table and a wordcount array named sortedList with 10 positions. It looks for the biggest number in the copy of the table and inserts that number and its corresponding word into the top of sortedList. We then delete this word and its corresponding number and do this a total of 10 times. Our function returns the sortedList.

Our fastertop10 function works almost the same but it doesn't create a copy of the table and instead just delete the values from there. The table isn't important since we won't be adding anything onto it after this point.