INF-1101: Introduction to programming and computer behavior

Exam

Mai 11th, 2022

1 Introduction

In this exam we were tasked with creating a search engine that would find the placement of words in your document, but we had to give it a few more features. My index was made with the use of a hash table and linkedlists since you can easily store and lookup values with the use of words as keys.

Features

- Word placement
- Scan through multiple documents and store words
- Iterate through multiple documents and show words

  #### Bonus features
  - Show how many words you've found on current document
  - Show which word you're currently on

1.1 Requirements

These are functions that I had to create for the search engine to work.

- Index_create
- Index_destroy
- Inde_add_document
- Index_find
- Result_get_content
- Result_get_content_length
- Result_next

And these are needed for the autocomplete to work

- Trie_find (not needed for then engine to
- Autocomplete (not needed for the engine to work)

2 Design

The ADT's I've used in my project are hash table, linkedlist and trie. Inside the hash table you'll find many linkedlists that hold placements, to access these lists you'll need a key. You could use "milk" as a key and get access to a linkedlist that contained "10", "54", "180" these numbers tell us where the word milk Is located within our document.

The autocomplete is made with a trie, I input every unique word into the trie and simply search and iterate through it to get a suggestion of what I'm typing could be.

Structs was also a big part of storing data and keeping everything tidy such that accessing whatever document and the data within it was no problem.

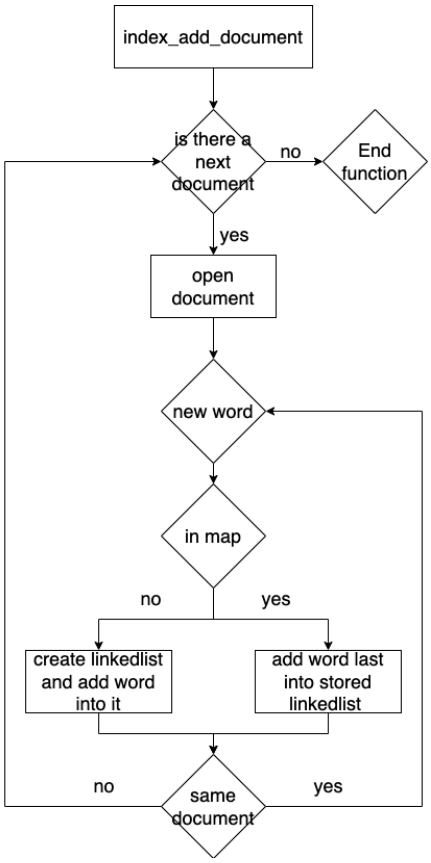


Figure 1: Flowchart of index_add_document

When the "index_add_document" function is called we store everything we need inside a document struct, we then enter a while loop that iterates trough the list "words" which a list made by tokenizing the document.

In this while loop I firstly check if the new word is contained within our map, this is done with the "map_haskey" if it's a new word we simply open the map with the word as the key and create a linkedlist, then we push our placement onto that list and then tell the map that the value is our linkedlist.

This means that we have now stored a linkedlist in the map, with our word as the key where the value in our linkedlist is the words placement.

When we hit the same word the next time we'll just open the map and put the placement inside the linkedlist there.

When I've gone through the whole document I simply add the pointer to the document struct into a linkedlist that is contained in the index struct. By doing this I can access and loop trough all my document data where I have access to my main index struct.

This also makes it easier to read the code and be able to understand and access it.

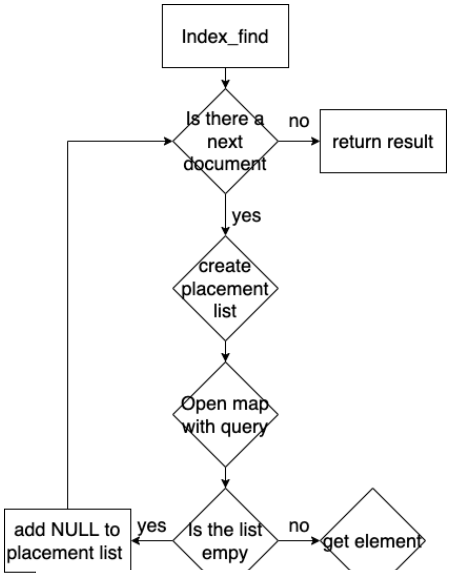


Figure 2: Flowchart of "index_find"

```
docu_iter = list_createiter(idx->document_data);
// Iterate through documents
while (list_hasnext(docu_iter))
{
    idx->result->document = list_next(docu_iter);
```

Figure 3: document iteration

When we want to find a word and its placements, we simply iterate through the list which contains the document pointers and check their hash table to see if they contain our word.

If the word is in our hash table we pull out the list that was assigned to the word and start to iterate through it, while we iterate through it we store the placement found into a list we made called "word_placements" when we're done iterating and have saved all the placements, we then save this new list in our index inside our result part.

Index->result->document->word_placement.

We can easily do this because of how we're iterating through the documents and by the way the structs are setup.

```
SEARCH RESULTS FOR /frog.txt
Hello i'm a frog.

HUMAN WORD: 4 COMPUTER WORD: 6
```

Figure 4: example of word placements

There is a little problem when it comes to placements for us and placements for the computer.

in this sentence "Hello I'm a frog." we would think that "frog" is placed 4. but the computer would say that it's placed 6. This is because the computer counts things we wouldn't think of as a word, like a space or even when there is a new line. To combat this I had another variable that would subtract from the word count every time it hit those things. The reason we need the computer version of the placements is for the highlighter to show the right word.

The autocomplete was simple to add, all I did was first iterate through what I had typed, then I iterated from 0 to 26 till I found the first node that wasn't NULL, that meant I had hit a new letter that I then jumped into and did the same there. This was done until the letter we were on had a key. The reason I check for a key is because when I put a word into the trie I give it a key value, which means that the last letter of every word will have a key while the letters before it won't. That means if we insert "frogman" into the trie, all the keys will be null except "n". This meant that when we hit a letter with a key we've also hit a full word. I then returned that word as a suggestion. This method goes by the alphabet and shortest word, because when it iterates from 0 to 26 it means we're first checking a then b then c and so on, and since we're stopping at the first key we find it means we'll hit the first word possible.

3 Discussion

I want to start by talking about why I've used a hash table and linkedlists, what I could have done and then end the discussion talking about time and space complexity between hash tables and tries.

When I first read about what we were tasked with in this exam I sat there thinking about how was I supposed to store all these words and efficiently look them up? That's when I thought about the "phonebook ADT" hash table, simply because when I want to add something to the ADT I'm able to choose a key and assign a value to that key, and when I want to look something up I just use a key and get the value, which fits perfectly with what I was tasked, since I can use a word as a key and assign the placement of the word as the value.

The problem I came across then was that this wouldn't work if I get the same word twice since the new word is just going to overlap the old one. The solution to that problem was adding some sort of list instead of the value and then adding the value onto that list. The list type I chose was linkedlist since it's I can easily look through it, add elements, create it and it allocates memory dynamically. Arrays would need a lot of extra data to keep track of, like the size of each array, allocate memory to each one and then dynamically allocate it later.

I will only compare the trie and hash table data types since I do think that I would also end up using linkedlists if I were to implement a trie instead of a hash table.

Hash table	Best case	Average case	Worst case
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Trie	Best case	Average case	Worst case
Searching	O(1)	O(n)	O(n)
Insertion	O(n)	O(n)	O(n)
Deletion	O(n)	O(n)	O(n)

Figure 5: Time complexity for trie and hash table [2&3]

When it comes to a search engine you want it to be fast, this is where you use big O notation to check best, average and worst case before you've even started. You want it to be fast because it has to feel responsive, if you pressed "ctrl + f" in word and had to wait 1 second when you try to search up a word it wouldn't feel all the responsive and great, it's a small thing but you notice it.

On figure 5 we can already see that the hash table is faster than the trie when it comes to searching and this will always be the case, the only time the trie will manages to search with best case is if it's looking for "a" since this is stored as the first node within the trie.

Some might think that the hashing of a key would slow down the hash table, but the hashing is so insignificant that it's not included.

Why is the hashing insignificant? Well when we say O(1) it means that the time spent is always the same no matter how big the table is. For the hashing the time spent is always the same no matter how big the word is.

The time spent hashing 1 key is 0,006 µs while the time spent searching for a word is 14 µs, here we can see that the hashing has close to no impact on our search through the map.

The worst-case O(n) is if we have collisions, which in this case wont matter since we have a linkedlist at the end, if the word frog and man has the same key it will still be O(1) since they're both pushed into the linkedlist, now this is a big problem if we end up with different words with the same keys, but you can combat this by making your table bigger as more and more keys are taken, and make the hashing function calculate its value with the size of the table.

The real difference is when we insert data

The hash tables insert and search functions basically operate the same, since they just open the map and either insert or extract data. So we follow the same logic that we used when talking about the search time spent. Inserting words does not care about the map size.

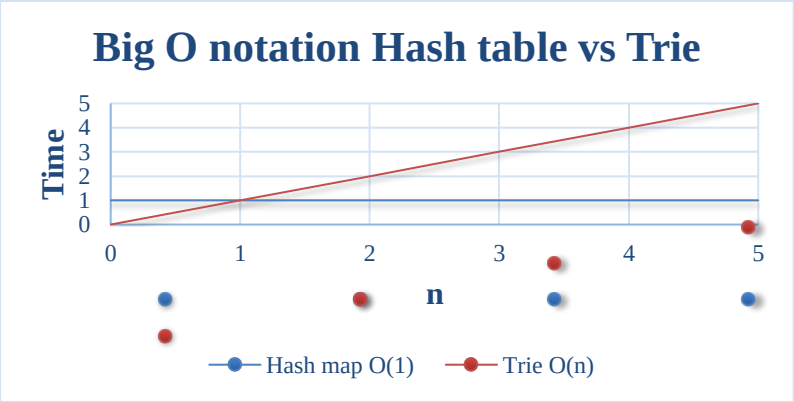


Figure 6: graph of time complexity for hash table and trie (insert & delete)

The time spent by a trie is going to be like a rollercoaster in the beginning, this is because n is the average word length within a trie, so the time spent will jump up and down with a higher margin at the start, but it will steady out the more words we get into the trie. In figure 6 you'll see that the only way for a trie to insert or delete faster is if the word average length is less than 1, which in reality will never happen.

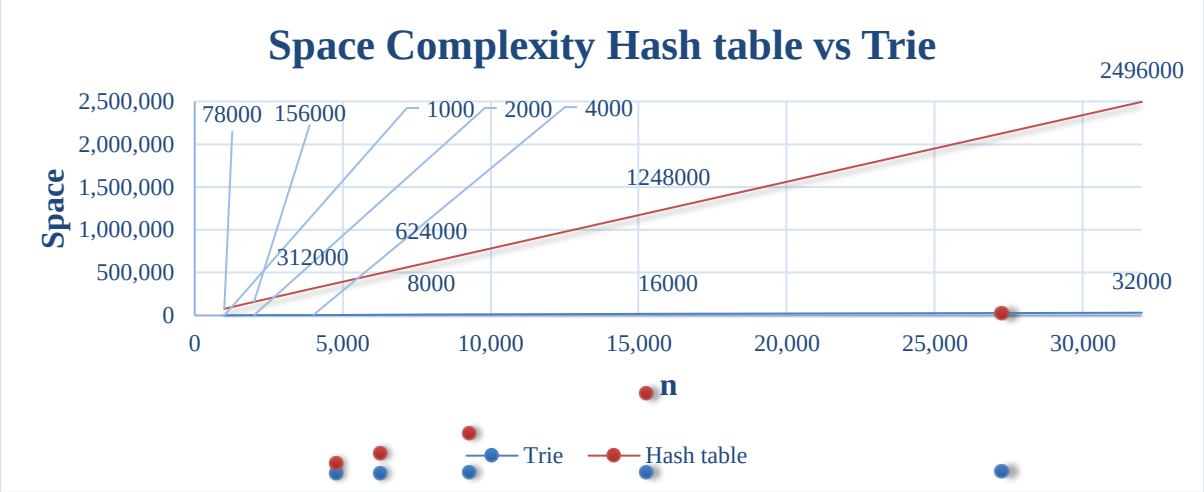
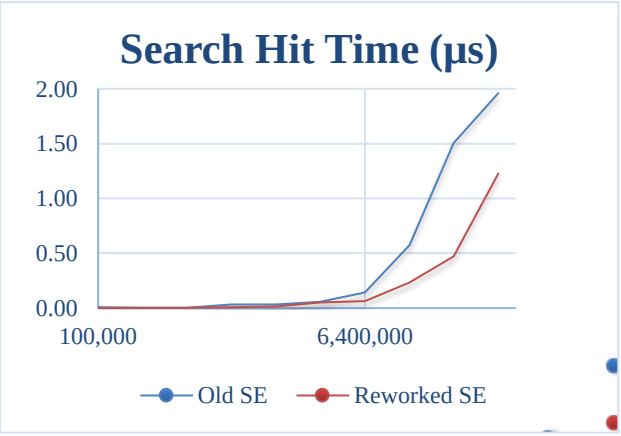


Figure 7: graph of space complexity between trie and hash table

Then we have space complexity, where the hash table simply has a big O notation of (n) this means that we only take the space that we need and nothing else. If we have 10 words in our big O would be (10)

When it comes to a tries space complexity it gets a little more complicated, first off a trie trades space for time, by this I mean that It uses way more space than needed so it can act faster. The big O notation for a trie looks like this (alphabet size * average key length * n) this is because for every letter we store, we have 26 places where there could be another letter, lets try to put 10 words into this trie, where the average key length is 4. (26 * 4 * 10) now the big O suddenly becomes (1040) that is a lot of space taken just for 10 words.

These are the reasons I chose to use a hash table, it's just better in every way since it takes less space while also being faster than the trie.



I also rewrote my whole search engine since the code was messy and had a lot of unnecessary parts to it that slowed it down.

Here on figure 8 we can see how much of an improvement I made on the rework, this is mainly because I used the structs more efficiently which then made it easier and faster to access data.

Figure 8: comparison between old SE and new

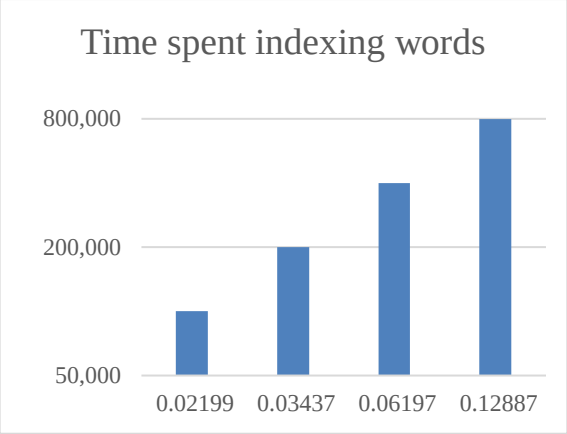


Figure 9: graph of time spent indexing. (hash table)

This is a graph of how much time my index spends on indexing words. Now this is something we would already know when looking at the time complexity for inserting into a hash table which is O(1). By that I mean that since the time it spends adding each word is always the same, the total time spent will increase linearly with the number of words, so if we double the number of words added we also double the amount spent indexing them. This is why big O notation can be so useful for us, since we can look at the O notation and pick the best ADT for our project without having to implement multiple ADTs and then check out which one is best. We don't need to have data to know how our code will work, we can just check the big O notation.

4 Conclusion

Almost everything works as I've intended it to work, there are no random crashes as far as I know since I've squashed the ones that was found. You can look up any word that you want and when you've gone through them all you're able to just go back and search up a new one. I wish I had more time to implement multiple word searches but that wasn't an option based on how much time I have left after procrastinating.

The only thing that does not work as intended after I implemented it is the autocomplete, I want to be able to write "hard" and then have it suggest "hardly" or write "drag" and it'll say "dragons" but this does not work because it will not suggest a new word if it already suggested a word and hasn't gotten any new input to refresh its suggestions, I do think this is a problem with how the autocomplete is implemented in the UI but I'm not sure.

Other than the autocomplete and not being able to search for multiple words I'm happy with how this turned out, I was not expecting to have my code running this smoothly with all these features working.

5 References

1. https://quick-adviser.com/what-is-the-time-complexity-for-inserting-an-item-into-a-hash-map/

2. https://iq.opengenus.org/time-complexity-of-trie/

3. https://iq.opengenus.org/time-complexity-of-hash-table/

4. https://iq.opengenus.org/time-complexity-of-linked-list/