INF-1101 Data structure and Algorithms

Exam Report

1. Introduction

In this exam, the assignment was to implement an index abstract data structure (ADT). This index should support indexing off words from documents and search after specific words and sentences. The final product should look like a typical *"ctrl + f"* service. My approach to create an index was to use a retrieval data structure for the searching of words, hashing data structure for storing the positions of words, and linked list to store different information about the documents added to the index.

2. Implementation

The program consists of 2 important structures, the index structure, and the search result structure. There is also an autocomplete function.

2.1 Index structure

The index acts like a library, containing all the information about the documents in an organized manner. The index is later used by the search result handling, and it is therefore important to make it easy and effective to access the necessary information. The structure consists of:

- A trie to hold every unique word.
- A hash map to hold the position of every single word.
- A linked list that holds information about the documents added to the index.
- An integer to count how many documents that is in the index.

The hash map is a combination of a chained and open map. The reasoning is that we want to hold every position of a specific word, and since we get the same hash value for the same word, we simply chain the positions for that word. If two different words happen to get the same table, than we just move the latest word to a new table and chain from there. The linked list that holds information about the documents uses a structure. This structure includes the content, name, and size of said document. Here is an illustration that visualizes how the index structure is implemented.

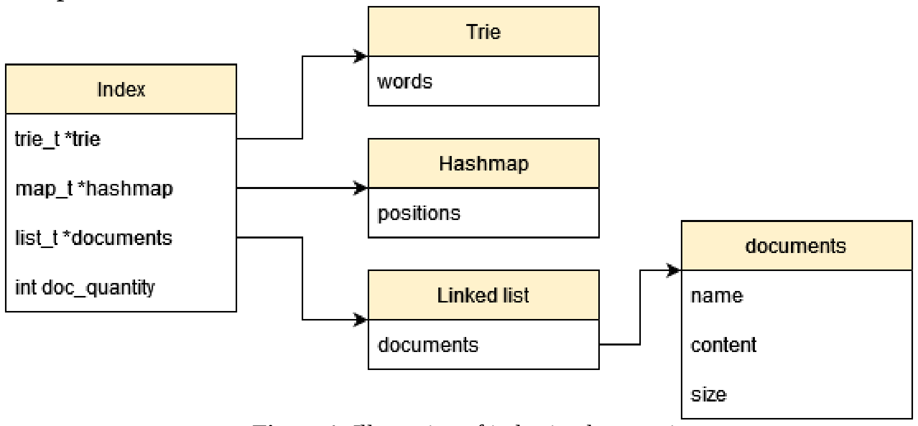


Figure 1: Illustration of index implementation.

The function that adds documents to the index, is the only function that directly affects the index. Other than that, we pass the information from the index structure to the search result structure.

2.2 Search result structure

When creating a search result, it takes a query as an argument and gets the positions for that query in every document. It acts like an iterator that will iterate through the positions and will tell which document the current hit is located. The structure itself consists of:

- A list iterator for the documents added to the index.
- A linked list that contains the positions and length for the search results after a query.
- An iterator that will iterate over said list.
- An integer to tell the size of the current document.

This is a visualization of how the structure gets its variables.

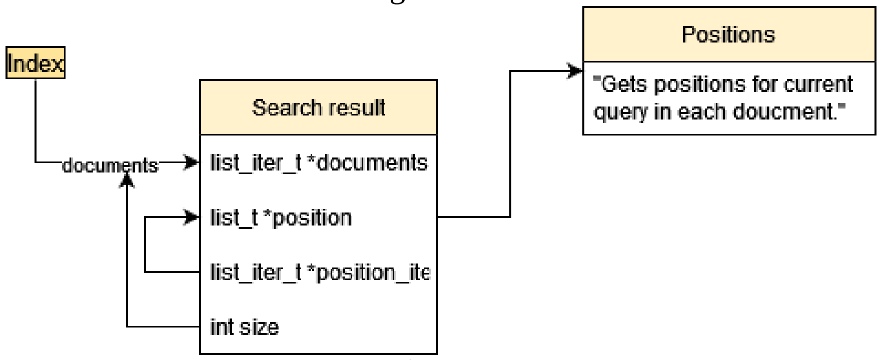


Figure 2: Illustration of search result implementation.

There are three functions that is used by this structure:

- Getting a single search hit for the current query and telling when to switch documents.
- Getting the content for the current document that the search hit is inside of.
- Getting the size of the current document.

This is a visualization of how the search result structure is used by its functions.

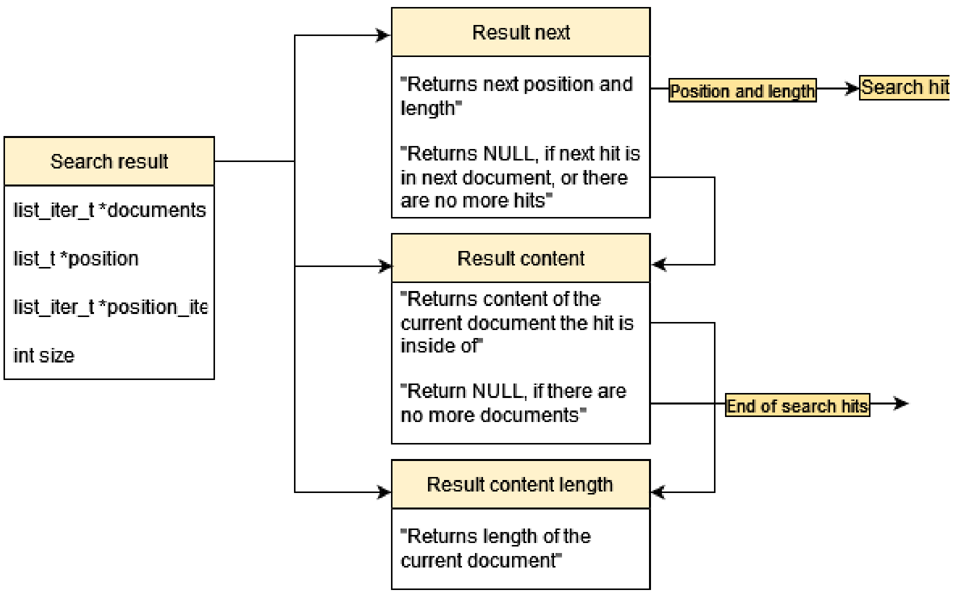

Search result
list_iter_t *documents
list_t *position
list_iter_t *position_ite
int size
Result next
"Returns next position and length"
"Returns NULL, if next hit is in next document, or there are no more hits"
Position and length
Search hit
Result content
"Returns content of the current document the hit is inside of"
"Return NULL, if there are no more documents"
End of search hits
Result content length
"Returns length of the current document"

Figure 3: Illustration of function logic by search result structure.

2.2a Position list

When the search result structure is created, it creates a linked list that contains the position and length for the search hits. This list is indirectly used by every function and is somewhat convoluted. It can also hold the position for every possible search, meaning it is possible to search for single words and sentences. There are three steps to create such a list. Let us use the search “Tragedy of Hamlet” as example.

First step, we call upon the hash map and obtain the table for each word. This means we have three different tables, each with the positions in every document for the unique word.

The second step is to filter the tables by which document the positions belong to. Let us say “Tragedy” appears first in the first document, but the second time is in the next document. We would then want to separate these two positions, into two different lists. By doing this step we will obtain lists equal to the number of documents, and each list will have the positions that belongs to that document.

The Third and final step is to find out if the positions appear in sequence. To do so we sort the lists by rising order, since our list only contain positions, and if they appear in sequence, they will now be right next to each other. Now we simply move through the list and calculate the difference between three positions at the time, since we searched for three words in our example.

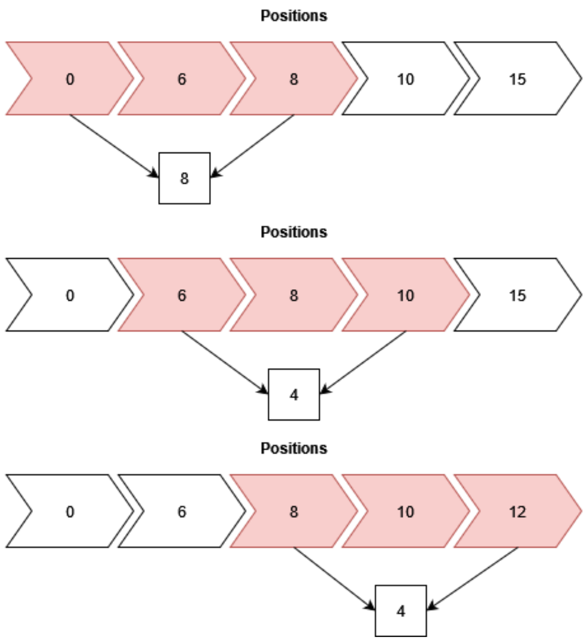


Figure 4: example of words that appear in sequence.

If we look at this illustration, we see that the difference between the first positions is 8, meaning they do not appear in sequence. The difference between the next positions is 4, meaning they do appear in sequence. Note that our program counts spaces and such as unique words. The last positions again do appear in sequence sine the difference is 4.

The last and final check in this step is to find out if the words that appear in sequence match the query. In our example we search for “Tragedy of Hamlet”, but there are two cases that happens back-to-back, this means that one of these cases does not match the query. Therefore, it is necessary to compare the positions with the query. If one of them match, we then add the position of the first word and the difference (being the length), to the final and complete list. We also add -1 to the final list to tell the rest of the program when to switch to the next document.

2.2b Search result handling

The search results are handled by iterating over the linked list that the structure creates. This linked list will contain the positions and lengths of search hits, it will also contain -1 to tell when to switch to the next document.

2.3 Autocomplete

The autocomplete calls upon the retrieval data structure in the index, it will then use the input form the search bar to traverse the trie until the end of the input. From there it will travers the shortest possible path by looking for a node that have a child. If a node has any value, it will return the key for that node. This means that the autocomplete finds the shortest complete word in alphabetical order.

3. Discussion

A perfect search application uses the shortest possible time to find a word. The trade-off to do so is usually less space efficiency. To find out if a program uses optimal time, you could do a Big O notation analysis.

3.1 Index structure

As mentioned, the index structure uses a hash map, trie, and linked list ADT. They have different purposes and accomplish their job in about the best possible way.

The hash map is used for storing positions, the key to access these positions is the word you wish to search for. This map is combination of a chained and open map, which will result in O(1) search time on average and O(n) in absolute worst case, where n is number of tables. [1] Since the map chains elements, there is almost no need to resize the map, which drastically decreases time spent inserting elements into the map. It neither uses much memory storage since elements are hashed into an array, which will result with less unused tables in the array. An optimalisation is to use a singly linked list instead of a doubly linked list to chain elements. The result is less memory storage.

The trie is used for storing every unique word and used to create an autocomplete functionality. The search time is brought to the optimal limit O(M) where M is the length of the string, however the downside is that it requires huge amount of storage. [2] To improve this issue you could implement another version of a trie, called Radix-trie. Which merges child nodes with its parents, if the child node is the only node of the parent. The result is that we reduce number of nodes, and therefor we reduce memory storage. [3]

The linked list which holds the information about documents is conceptually the worst ADT in the index structure. Compared to an array, a linked list requires extra storage, random access is not allowed, and it is not cache friendly, which in some cases makes it a lot slower to iterate through depending on the data it holds. [4] But in our case, a linked list works just fine. Its dynamic which means we could easily add a new document to the index without much effort, and there is no need for random access, since we want to iterate through it start to finish and access all its elements. This will result in the linked list to scale O(n) in time complexity, where n is number of nodes.

3.2 Search result structure

Linked list is the only ADT in the search result structure, and it gets passed information from the index. The structure mainly spends time creating a list that contains the position and length of search hits, as described in the implementation section. The process for creating this list involves creating and deallocating many other linked lists, which is a slow process and uses a lot of memory.

A simple optimalisation is to use singly linked lists, as they need less memory. But to improve the performance drastically, you could skip step two in the process entirely. Recall that step two was separating words from different documents to their own lists. A solution could be adding an array as the value to the hash map, that contains the position as usual and another integer that tells which document it belongs to. Now we can use this integer to instantly separate the positions correctly. Another solution could be having different hash maps for each document. Both approaches would require more memory usage by the index structure, which is a favorable trade-off since we care about time efficiency rather than space efficiency.

Recall in step three, that we sort the lists by rising order, which is very important since we can easily check if positions appear in sequence. But if the case is that the user only searches for a single word, this makes the list automatically being sorted in rising order. Which means we could skip sorting the list entirely. If the user searches for more than a single word, then we can predict how the list will look like. Let us say we search for “Tragedy of Hamlet” again, then the first values in the list will be all be “Tragedy”, then “of”, and lastly “Hamlet”. Here is a visualization.

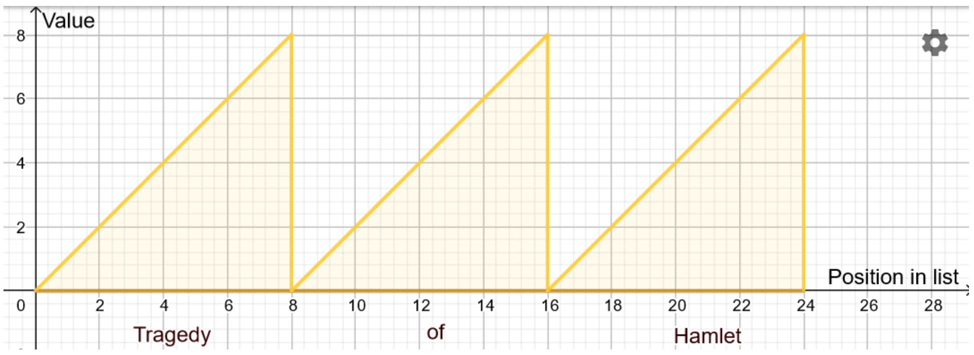


Figure 5: Visualization of values inside position list.

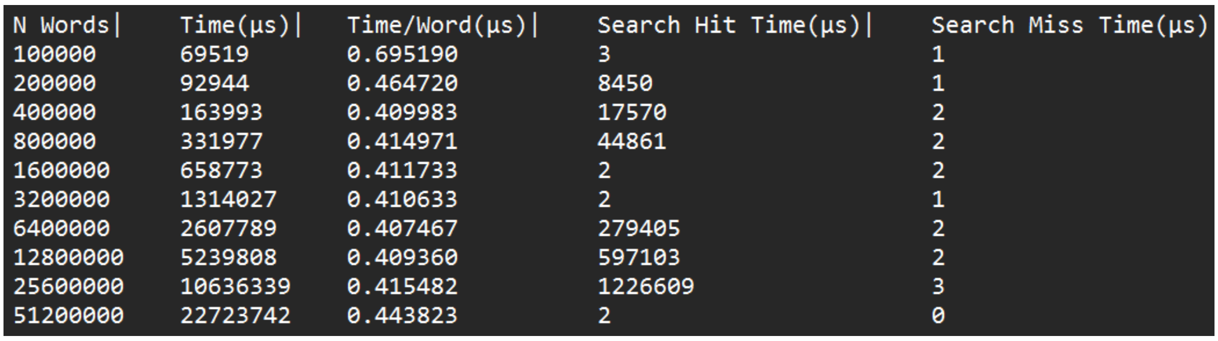
As we can see, the value rises for “Tragedy” as we move further into the list, since the word is found multiple times in the document, until it suddenly drops. This is where the values of “of” starts, and the pattern repeats itself. This makes merge sort a very effective algorithm, since it scales O(n log n) in all 3 cases (best, average, worst) in time efficiency, where n is number of elements. [5] And the list will always have this pattern regardless of how many words the user searches for.

3.3 Results

The results we get from the ADTs in the index structure is that it scales well with just adding a lot of words and not many documents, since the hash map scales O(1) on average, and the linked list scales O(n). We get a worse result when adding a lot of documents since the list scales O(n).

Search time simply scale O(n) since most of the handling uses linked list. There is not really any difference in adding many words or documents. It will simply scale with the amount added.

If we run the benchmark test, which stress test the index by adding a substantially number of words, we get this result.

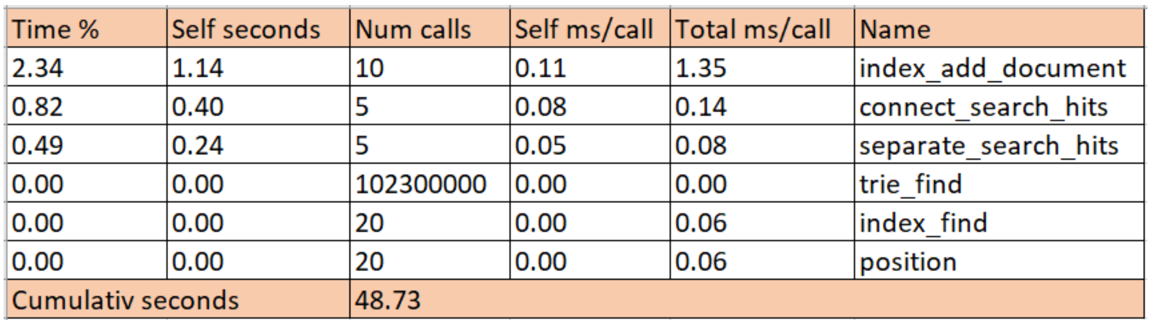


```
N Words|    Time(µs)|   Time/Word(µs)|    Search Hit Time(µs)|     Search Miss Time(µs)
100000      69519       0.695190          3                        1
200000      92944       0.464720          8450                     1
400000      163993      0.409983          17570                    2
800000      331977      0.414971          44861                    2
1600000     658773      0.411733          2                        2
3200000     1314027     0.410633          2                        1
6400000     2607789     0.407467          279405                   2
12800000    5239808     0.409360          597103                   2
25600000    10636339    0.415482          1226609                  3
51200000    22723742    0.443823          2                        0
```

Figure 6: Benchmark result.

By understanding this result, shows that the time it takes creating the index doubles when number of words added is also doubled, which makes sense since the hash map scales O(1). There is approximately no difference in insertion time per word regardless of how many words is added, which is as expected. Search hit time have a lot of variation, which does not really make that much sense. The only explanation is that in some of the cases, the benchmark test searches for either a word at the start or the end of the document. This will result in short search time if the word is at the start, or longer time if the word is at the end. Search miss is instant, since the user searches for a word that does not exist in the hash map, which means that the word points to an empty table in the map.

Using a profiler to gather the time spent by the program during the benchmark test and filtering the result by functions that I have implemented, we get this result.



Time %	Self seconds	Num calls	Self ms/call	Total ms/call	Name
2.34	1.14	10	0.11	1.35	index_add_document
0.82	0.40	5	0.08	0.14	connect_search_hits
0.49	0.24	5	0.05	0.08	separate_search_hits
0.00	0.00	102300000	0.00	0.00	trie_find
0.00	0.00	20	0.00	0.06	index_find
0.00	0.00	20	0.00	0.06	position
Cumulativ seconds		48.73			

Figure 7: Important functions to look at during the benchmark test.

The result is sorted by which function uses the most amount of time in the program. The function that adds words from a document into the index used a total of 1.14 seconds out of 48.73 seconds of total run time, which calculates to 2.34%. Apart from that we see that step two and three in creating the search hit list (connect- and separate search hits) uses some time and connecting the search hits is the slower function. As discussed earlier, there are some optimalisations that can be implemented. “Trie find” is used in the autocomplete functionality, it is called a ridiculous amount of times but don’t use any time at all, which seems suspicions but I do not have an explanation for this. The last two functions are used to create the search result hit list. These functions alone do not use any time, but the last cell in the table *represent the average number of milliseconds spent in this function and its descendants per call.* [7] Which adds up to 0.06 seconds.

4. Conclusion

My program uses a hashing-, retrieval-, and linked list data structure to create an index which supports storing of words and positions inside a document. The index also supports searching for single words and full sentences, and an autocomplete functionality that will suggest a word when the user types in the search bar. The results from running a benchmark test which stress testes the index by inserting a vast number of words, shows that time used creating the index scales linearly with number of words added. There are some optimalisations to be done, but the program runs relatively smoothly.

References

[7] *archive.* (2000, August 18). Retrieved from GNU gprof: http://web.archive.org/web/20141129061523/http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html#SEC2

[1] GeeksforGeeks. (2021, July 19). *GeeksforGeeks.* Retrieved from Hashing | Set 1 (Introduction): https://www.geeksforgeeks.org/hashing-set-1-introduction/

[2] GeeksforGeeks. (2022, March 29). *GeeksforGeeks.* Retrieved from Trie | (Insert and Search): https://www.geeksforgeeks.org/trie-insert-and-search/

[4] GeeksforGeeks. (2022, March 31). *GeeksforGeeks.* Retrieved from Linked List | Set 1 (Introduction): https://www.geeksforgeeks.org/linked-list-set-1-introduction/

[5] GeeksforGeeks. (2022, April 22). *GeeksforGeeks.* Retrieved from Merge Sort: https://www.geeksforgeeks.org/merge-sort/

[3] Wikipedia. (2022, March 15). *Wikipedia.* Retrieved from Radix tree: https://en.wikipedia.org/wiki/Radix_tree