UiT Norges arktiske universitet

Institutt for Informatikk

Search application

Assignment 2

Erling Heimstad Willassen

INF-1101, Spring 2025

Content

1 Introduction

The student was tasked with implementing a Search Application that return document names from a query input of words and operations. The query indicates one or more words that are either in or not in the documents which to be returned. The implementation will return a list of documents with a point score based on the relevancy to the query input. This report will describe in detail the design and implementation of said search application.

2 Technical Background

2.1 Inverted index

Inverted index is a type of database where the index are the terms (words, numbers) that are contents in other objects mapped into a table [1]. In contrast, forward index is when the objects are the mapped key while the terms are the stored content in the index. With inverted index, the mapped terms will contain all the objects/documents of interest to that specific term. This makes it efficient for search engines to fetch all the documents relevant to a particular term. With forward indexing, a search engine would have to iterate through all the documents in the database to find the key of interest and would therefore be highly inefficient in fetching terms.

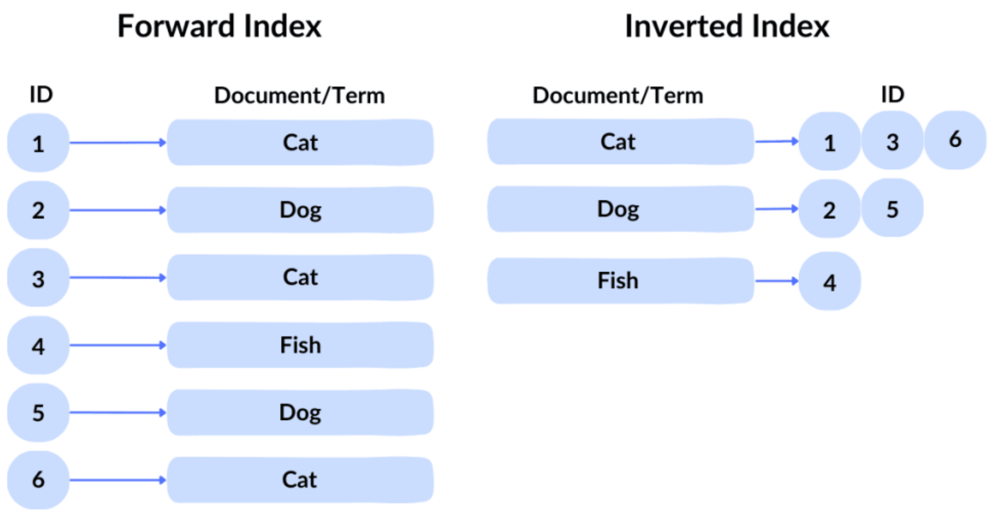
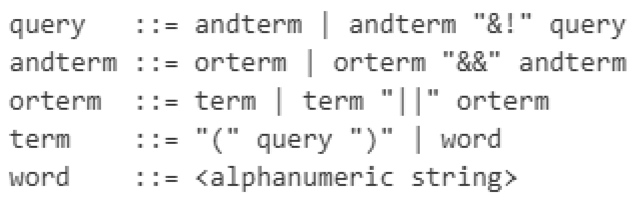


Figure 1: An example of how inverted index structure is compared to a forward index[2].

2.2 Backus-Naur Form

Backus-Naur Form (BNF) is a way to define how grammar rules should be represented in a programming language code [3]. The BNF will define how grammar rules should be represented logically in the code through syntax. Defining the syntax rules with BNF makes it easier to describe the logical structure of a syntax for others to read and understand, instead of describing in words that could deem difficult to understand and convey the structure.



```
query   ::= andterm | andterm "&!" query
andterm ::= orterm | orterm "&&" andterm
orterm  ::= term | term "||" orterm
term    ::= "(" query ")" | word
word    ::= <alphanumeric string>
```

Figure 2: An example of Backus-Naur form of syntax logics [4].

2.3 Recursive Descent Parser

Recursive Descent Parser (RDP) is a top-down parser that handles input from a set of recursive functions, where each function is determined by grammar rules. A top-down parser is a methodology that starts with the left most symbol and breaks the input down by each grammar rule set in the RDP. Having a recursive functions for each grammar makes it easy to implement and handle inputs.

2.4 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a data tree structure that represents operations done in a chronological order [5]. In the AST, the root nodes of each subtree represent a Boolean operator that dictates an operation on its two child nodes. The leaf nodes (nodes with no children) represent the terms that an operation should be performed on. By traversing through the structure from the root node of the AST, the leaf nodes will be performed an operation on based on what the Boolean operator represent. The result will be sent upwards to the next Boolean operator until all operations are met, and the resulting term operations are returned to system.

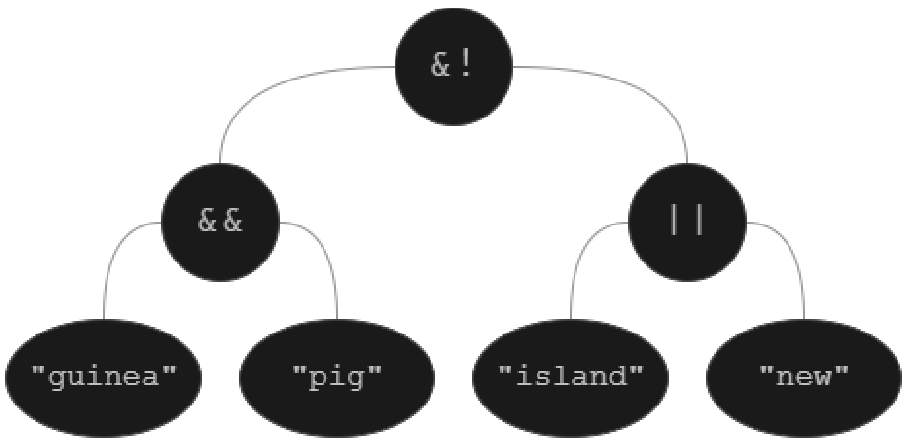


Figure 3: Example of an AST from a query with operators and words [4].

3 Design

The design of the search application will have each term or word as a key to a hashmap as the inverted index. The element in the bucket for each key in the hashmap will contain an ADT that contains all the documents to that specific term. For each document in the ADT, an own struct that holds a string for document name and an integer for word score will be implemented. This way, when fetching documents for a term the score (frequency of word in document) is also easily fetched.

When acquiring a query, each token in the query needs to be parsed and implemented into an AST. While the assignment recommends a recursive descent parser function for each grammar rule, in this design a one recursive function is made for all terms. This function handles all terms and recursively initiate itself for further development. The function will then create an AST based on the terms and operators in the query. Each word or operator node in the AST will have identifier to as what type of term it is for later use in set operations.

The operators are the terms that dictate which operations should be performed on the words in the query. In this assignment, the operators for union, difference, and intersection operations will be represented as the character string of “||“, “ &!“, and “&&“, respectively, from the query. These symbols will indicate what operations will be performed on the tokens from the query. Grouping of queries will be performed by having parenthesis with the terms inside the brackets.

After creating the AST, the tree will be traversed with recursion until the leaf nodes containing the words are found. There the words are exposed to a set operation based on the parent node operator. After set operation, the resulting set are returned where the next set operation is performed until the whole tree is traversed recursively.

The returned set with documents based on the query are then given points based on the frequency of the words in query to each document. Each document is then put in a linked-list which is then returned to the search application.

4 Implementation

For the implementation of the design for the search application some adjustments had to be made for having a functional query searcher. For the ADT as the key element in the hashmap for the inverted index, a linked-list was first thought to be implemented as to store document names for that specific term. However, this would give problems deeper down in the application when trying to do set operations on the terms in the query. Therefore, using an ADT of a red-black tree in the pre-code following the assignment was deemed as the best solution. This ADT contained methods that allowed for set operations, which means that the one did not have to make new functions for set operations on linked lists thus saving time. Having a red-black tree also makes search algorithm in the ADT much faster than by having linked-lists.

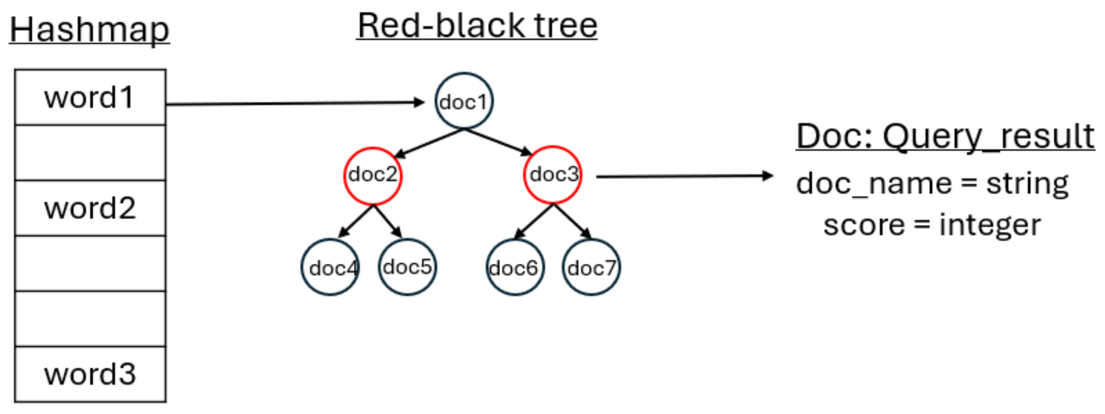


Figure 4: Illustrator on how the implementation of word to document names with frequency are stored.

When inserting query term to a hashmap with document name, a structure called query_result was used to contain that information. The structure query_result was a part of the pre-code that was supposed to be used to return the resulting documents after a query operation containing document name and score of the resulting operation. However, the structure deemed suitable to also store information for document name and frequency of that word in the document to the hashmap as well. This way, the structure had to two purposes in the code by both storing document name and frequency of word in hashmap and returning documents to the search engine, saving code space and time in coding. Although, searching through documents that were already in the hashmap for a specific term was a bit more difficult by using query_result structure. The pre-code function for searching did not work well with nodes that has query_result as items. A new function called cmp_doc_query() was made specifically to search for query_result objects in the hashmap ADT to avoid this problem. By using this function, words that are occurring several times in a document could be fetched and the frequency of the word could be easily updated.

For the AST, a new ADT had to be made to facilitate correct set operation on the terms from a query. This new ADT was called parser_tree which has just a pointer to a root node as a variable. The nodes contain four variables for pointer to token_type, item (string for term), and left and right pointers to next nodes further down in the tree structure. Token_type is another struct that contains Boolean values for AND, OR, ANDNOT, and WORD for term confirmation for that node. By having these Boolean values, it was easier to make set operation algorithms by checking to see if the node is a word or not, and if the algorithm had to traverse further down the tree to the leaf nodes where the word terms are located.

The parsing function was created as a one-stop function to handle all terms in a query. The parsing function is defined into two parts. The first main part of the function checks if the query term is a "(", indicating a query grouping of terms. This will recursively initiate the function again where the next term will be checked. If the next term is not "(", the second main part of the function will be initiated. This part will control all terms inside a group query and create nodes to a subtree. After a word child node and a term parent node is created, the function recursively initiates itself again in case there is another group query inside the first group query, where the process will be repeated, or a word-term, where the node is returned as the right child node of the subtree. The second part is now done and returned to the first part of the function. Here, it will check if the next term is ")", meaning the full query is done, or if there is more to the query. And the process is repeated once again.

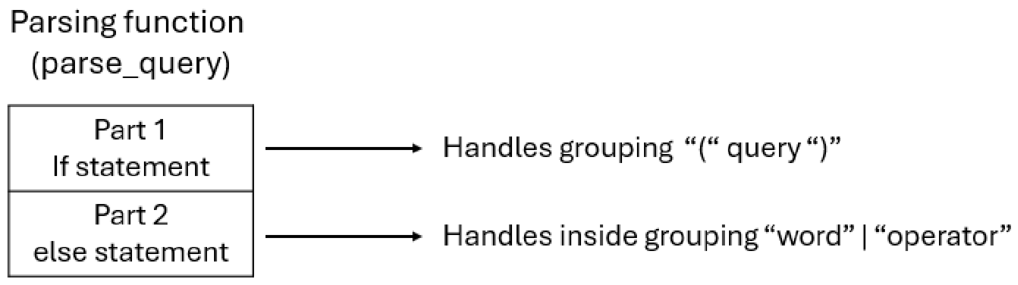
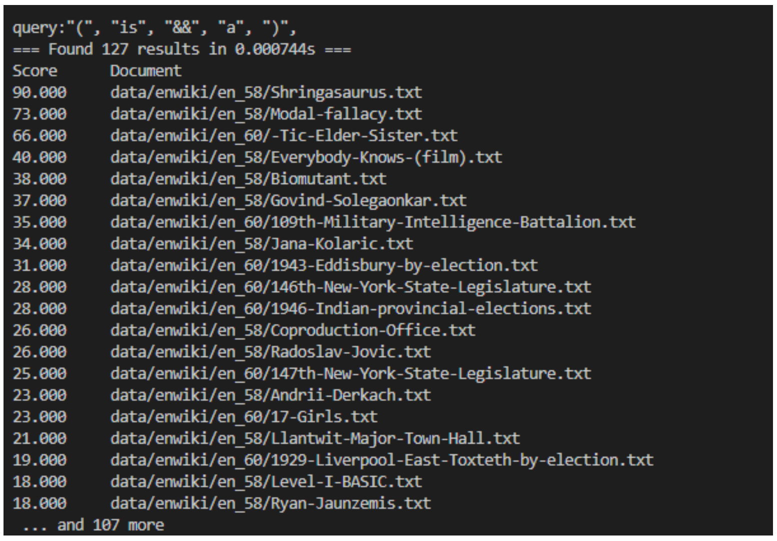

Parsing function
(parse_query)
Part 1
If statement
Part 2
else statement
Handles grouping "(" query ")"
Handles inside grouping "word" | "operator"

Figure 5: A compressed version on how the parsing function divides their task.

After performing set operations from the query, a result set was made in the structure of a red-black tree. This tree contains all the documents that are the result from the set operations from the query. To determine which documents are the most relevant for the query, a scoring system were created by summing the frequency of all word-terms to each document. This scoring system makes so that if many combined words are prevalent in the document, the sum of all words are added to the score of the document. This was done by first creating a linked-list of all the word-terms in the query. Then the documents for each word from the hashmap was compared to the documents from the resulting set. The documents that matched, got the frequency for that specific word added to the document score variable in the resulting set. After traversing through all the words, all the documents would have their score variable updated. The documents with the updated scores in resulting set were then put in a linked-list which and sorted after their score, and returned to the search application system. After this, the search application presents the results, and the query parsing are done.

5 Result

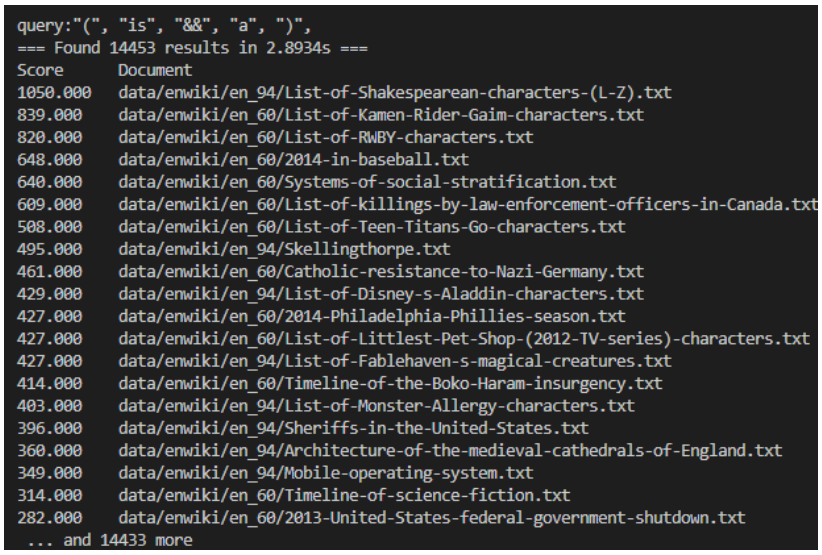
The search application algorithm works as intended. The algorithm can handle error handling fine as long as the same number of front and back parentheses are used in the query. One can also write without parentheses and still work, although some unexpected results may occur. The search algorithm works fast with no noticeable time delay for all sized queries.



```
query:"(", "is", "&&", "a", ")",
=== Found 127 results in 0.000744s ===
Score      Document
90.000     data/enwiki/en_58/Shringasaurus.txt
73.000     data/enwiki/en_58/Modal-fallacy.txt
66.000     data/enwiki/en_60/-Tic-Elder-Sister.txt
40.000     data/enwiki/en_58/Everybody-Knows-(film).txt
38.000     data/enwiki/en_58/Biomutant.txt
37.000     data/enwiki/en_58/Govind-Solegaonkar.txt
35.000     data/enwiki/en_60/109th-Military-Intelligence-Battalion.txt
34.000     data/enwiki/en_58/Jana-Kolaric.txt
31.000     data/enwiki/en_60/1943-Eddisbury-by-election.txt
28.000     data/enwiki/en_60/146th-New-York-State-Legislature.txt
28.000     data/enwiki/en_60/1946-Indian-provincial-elections.txt
26.000     data/enwiki/en_58/Coproduction-Office.txt
26.000     data/enwiki/en_58/Radoslav-Jovic.txt
25.000     data/enwiki/en_60/147th-New-York-State-Legislature.txt
23.000     data/enwiki/en_58/Andrii-Derkach.txt
23.000     data/enwiki/en_60/17-Girls.txt
21.000     data/enwiki/en_58/Llantwit-Major-Town-Hall.txt
19.000     data/enwiki/en_60/1929-Liverpool-East-Toxteth-by-election.txt
18.000     data/enwiki/en_58/Level-I-BASIC.txt
18.000     data/enwiki/en_58/Ryan-Jaunzemis.txt
 ... and 107 more
```

Figure 6: Two of the most used words in a query with a small dataset (400 documents).

With larger datasets of >10 000 documents, the search application did become noticeable slower with words that are the most frequent in documents such as “a” and “is” compared to the system with small dataset. See the time results for the larger dataset (2.8934s) compared to the one with a small dataset (0.00744s).



```
query:"(", "is", "&&", "a", ")",
=== Found 14453 results in 2.8934s ===
Score      Document
1050.000   data/enwiki/en_94/List-of-Shakespearean-characters-(L-Z).txt
839.000    data/enwiki/en_60/List-of-Kamen-Rider-Gaim-characters.txt
820.000    data/enwiki/en_60/List-of-RWBY-characters.txt
648.000    data/enwiki/en_60/2014-in-baseball.txt
640.000    data/enwiki/en_60/Systems-of-social-stratification.txt
609.000    data/enwiki/en_60/List-of-killings-by-law-enforcement-officers-in-Canada.txt
508.000    data/enwiki/en_60/List-of-Teen-Titans-Go-characters.txt
495.000    data/enwiki/en_94/Skellingthorpe.txt
461.000    data/enwiki/en_60/Catholic-resistance-to-Nazi-Germany.txt
429.000    data/enwiki/en_94/List-of-Disney-s-Aladdin-characters.txt
427.000    data/enwiki/en_60/2014-Philadelphia-Phillies-season.txt
427.000    data/enwiki/en_60/List-of-Littlest-Pet-Shop-(2012-TV-series)-characters.txt
427.000    data/enwiki/en_94/List-of-Fablehaven-s-magical-creatures.txt
414.000    data/enwiki/en_60/Timeline-of-the-Boko-Haram-insurgency.txt
403.000    data/enwiki/en_94/List-of-Monster-Allergy-characters.txt
396.000    data/enwiki/en_94/Sheriffs-in-the-United-States.txt
360.000    data/enwiki/en_94/Architecture-of-the-medieval-cathedrals-of-England.txt
349.000    data/enwiki/en_94/Mobile-operating-system.txt
314.000    data/enwiki/en_60/Timeline-of-science-fiction.txt
282.000    data/enwiki/en_60/2013-United-States-federal-government-shutdown.txt
 ... and 14433 more
```

Figure 7: Two of the most used words in a query with a big dataset (>10 000 documents)

6 Discussion

The search application was a success where the design and implementation worked as intended. By writing a query with right syntax into the search application, the right documents corresponding to the query was shown in the terminal. The time usage was unnoticeable from shorter to larger queries, indicating that the system is optimized for most query size.

The system worked perfectly in the set-up for the student. However, this set-up might not be the best test subject for the system. Due to the size of the compressed document database, not all documents were extracted for time reasons. This means that the test database for documents for the system was small, and therefore the system did not have much information to work with. This could make the system seem to act efficiently and fast even with queries that are complex and long even if the system is not optimized correctly for said queries. However, even if this might be true, the effect for larger datasets could be negligible. Since fetching documents from words happens with a hashmap and a red-black tree, the time complexity for search is O(1) and O(log n) [6], respectively. This means searching for documents from a large dataset would most likely not be too time consuming than from a smaller dataset. Although, the number of operations compared to the size of the dataset would be seen due to the O(log n) nature of red-back tree, the number of operations would diminish with the size of dataset as well. This would most likely mean that the search application system would work with even larger datasets as well. The test on >10 000 documents did show this to be true as seen in figure 7, although the words such as “is” and “a” in the documents did have a big increase in time use.

For the parser function, a one-stop function for all terms from the Backus-Naur form were created instead of one function for each term. The reason for this decision was part challenge and part having full overview of the parsing in one function. By having one function to do the whole parsing sequence, it was easier to understand the flow of tokens from the query would be represented into the AST. By having five different functions, the flow would not be as obvious on how tokens would be affected throughout the parsing process. With having one function, the readability of code would also be easier for others to understand the process of the parser instead of having to follow a thin red recursive line between functions.

The design and implementation of the inverted index was deemed to be the best structure for a search application system. By having the words as the key for a hashmap, a search complexity of O(1) was achieved when trying to acquire the documents for said word. This design would yield fast query search due to the low complexion of hashmap. For the design of storing documents, a red-black tree was used. The reason for the black-red tree ADT was no more than that the code was already implemented in the pre-code. The red-black tree also had set operation such as difference, union and intersection functions that would even further save time to code the search application. In theory, most other trees or even hashmaps could be work perfectly fine in this scenario as long as set operation would be possible. However, by having an ADT with O(n) complexion, such as a linked-list, could be too ineffective and slow for words with big datasets of documents.

The implementation of the ADT worked perfectly as intended in the design. By having recursively functions that traverse down the ADT in both left and right nodes until leaf/word nodes are met, one could easily do set operations on the words and returning the resulting set upwards to the next set of operations.

7 Conclusion

The design and implementation of the search application worked as intended. By having a hashmap for words found in documents and a red-black tree ADT to store document name and frequency, the search application achieved great speed and efficiency to acquire the document of relevance from a query. By having a parsing function that puts words/terms in the leaf nodes, the system acquired an efficient AST to perform set operations on and acquire the document of interest.

8 Sources

[1] «Inverted index», *Wikipedia*. 5. mars 2025. Åpnet: 28. april 2025. [Online]. Tilgjengelig på: https://en.wikipedia.org/w/index.php?title=Inverted_index&oldid=1278940424

[2] N. V. Otten, «How To Implement Inverted Indexing [Top 10 Tools & Future Trends]», Spot Intelligence. Åpnet: 30. april 2025. [Online]. Tilgjengelig på: https://spotintelligence.com/2023/10/30/inverted-indexing/

[3] «Backus–Naur form», *Wikipedia*. 15. mars 2025. Åpnet: 28. april 2025. [Online]. Tilgjengelig på: https://en.wikipedia.org/w/index.php?title=Backus%E2%80%93Naur_form&oldid=1280640335

[4] «Search Application, INF-1101, Obligatory Assignment 2». Universitetet i Tromsø, Spring of 2025.

[5] «Abstract syntax tree», *Wikipedia*. 14. mars 2025. Åpnet: 28. april 2025. [Online]. Tilgjengelig på: https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1280483463

[6] «Red–black tree», *Wikipedia*. 27. april 2025. Åpnet: 30. april 2025. [Online]. Tilgjengelig på: https://en.wikipedia.org/w/index.php?title=Red%E2%80%93black_tree&oldid=1287651957