Name-Surname: Burak ÇETİN

Number: 201711015

Course: Data Structures - CENG 218

Data Structures Course (CENG 218) Homework 5

(Document Indexing System)

1. Introduction:

Document indexing in its simplest definition is organizing and storing documents for later retrieval based on words they contain. This system will allow users to enter a single word, then the system will return a list of documents containing this word.

In this project, I designed a document retrieval system that will respond to single-word queries (determining documents containing the given word). And also, I experimented on how long indexing of all the documents and search operations take on with LinkedList and Binary Search Tree using the contents of the files. I compared the performance differences arising from the usage of LinkedList and BST while indexing the documents by words in their contents.

With this technique called indexing, we are getting rid of constructing a program that reads a keyword from a user, and then the program will go over (scan) all files and list the names of the files containing this keyword. Developing a solution like this to this problem would be the most expensive method in terms of efficiency and time. So, we are using the indexing approach.

2. Design of the Program:

This system is a simple menu-driven program that will first start the indexing by reading file location. Once the index is created, the system will indicate the end of the index, created with an appropriate message, and then the menu will appear that will enable the user to benefit from the system.

Users are interested in finding documents containing a particular keyword. And the program that I developed will let users search for this keyword among all documents, of which there are approximately 10,000 text files.

There are certain searching rules while reading text files (or documents);

- All words that are inputted by the user will be lowercased. So, the user can search only non-alpha and lowercase characters [a-z].

- When the user inputted a keyword which is a string of alpha characters terminated by a non-alpha character (white space is not alpha). The alpha characters are defined to be [a-z].
    - For example; the sequence of characters for the word "*apple+78&'^+orange*" would be '*apple*' and '*orange*'.

- There is also a stopwords.txt file. This file consists of filtered words, such as 'a', 'an', 'the' etc. "stop words" usually refer to the most common words in a language. So, these words will be trimmed and will not be searched when the user runs the program.

3. Code and Results:

Before getting into the results of the indexing process, I want to briefly explain the code blocks that I wrote. First of all, I declared an array of strings named stopArr which has the size of 174 for saving stoping words, which I mentioned in part 2 (Design of the program), into a string array.

After that, I printed the menu on to screen thus users can have a choice of their process within it. The menu will have appeared after each operation is done until the user chooses to exit from the program.

If the user chooses to search a keyword, depending on whether it has in the documents or not, the required output is printed on the screen:

- Providing the keyword is exist in the documents, the program will be starting to search the inputted keyword, which means finding a list of documents containing this word, and will print the including text numbers to the screen.
- But if the keyword does not exist, "NOT FOUND!" text will be printed on the screen.
- A final option for keyword indexing choice occurs when the user enters a stoping word. In this case, the program will not accept the keyword, and print a warning will be suppressed that indicates, "stoping word has been entered", on the screen.

I didn't make other menu options. Because, in those options, no situation should be observed within the scope of the project. So, I only observe the searching within LinkedList and BST, other options are excluded.

a. Indexing on LinkedList:

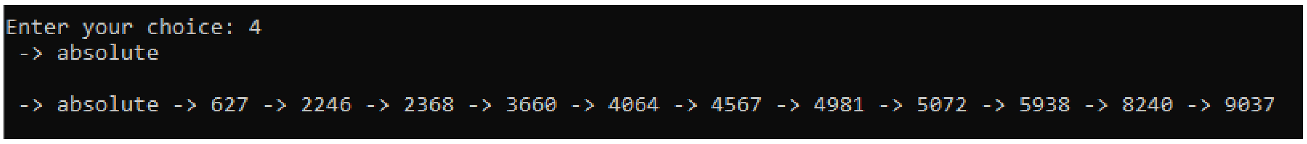
Theoretically, the time complexity of searching on a linked list is O(n). On the other hand, inserting an item to the front of the linked list is O(1).

Searching on Linked List $\longrightarrow$ **O(n)**

Inserting an item to the front of the Linked List $\longrightarrow$ **O(1)**

In the practice of indexing on LinkedList with my code. I get approximately 4.5 seconds (O(n) cost) to end the search process on LinkedList. And the insertion process took about the same amount of cost as theoretically suggested with big O notation, which is O(1).

When the user wants to quit, the program will show a summary chart of the linked lists.



```
Enter your choice: 4
 -> absolute

 -> absolute -> 627 -> 2246 -> 2368 -> 3660 -> 4064 -> 4567 -> 4981 -> 5072 -> 5938 -> 8240 -> 9037
```

Figure 1: Console View of the Containing Files

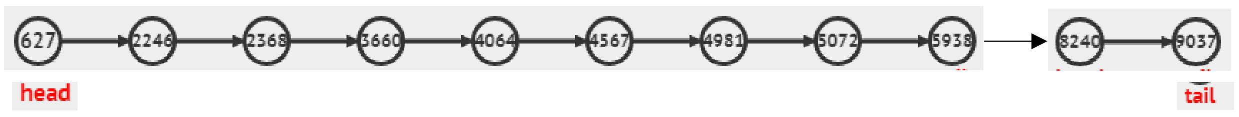


Figure 2: LinkedList View of the Containing Files

b. Indexing on Binary Search Tree (BST):

Theoretically, the time complexity of searching on a binary search tree is O(log n). And, adding an item into the binary search tree is also O(log n).

Searching on BST $\longrightarrow$ **O(log n)**

Adding an item into the BST $\longrightarrow$ **O(log n)**

In the practice of indexing on BST with my code. I get approximately 6.5 seconds (O(log n) cost) to end the search process on BST. And the insertion process took about the same amount of cost as theoretically suggested with big O notation, which is O(log n).

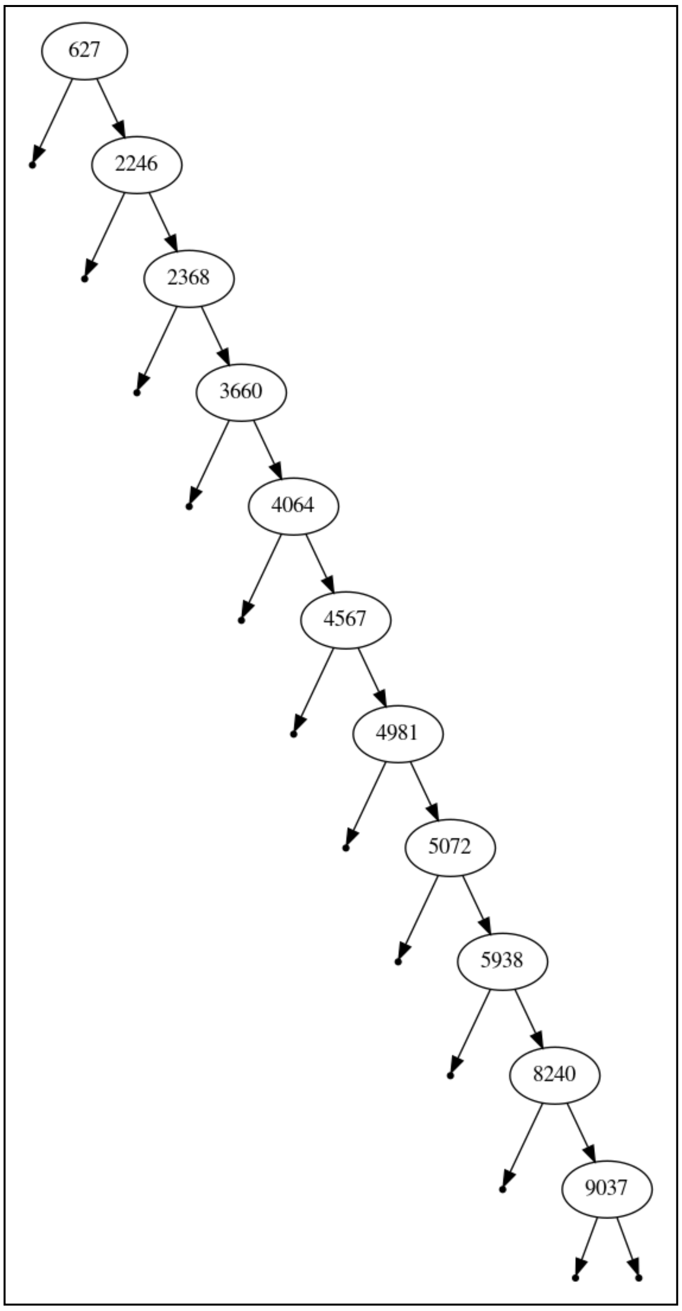


Figure 3: Binary Search Tree View of the Containing Files