This document presents an overview of a word analysis application designed to efficiently analyze a text file, such as providing comprehensive statistics regarding the words used in the text file.

Contents

Notes:

1- When the application prompts you to enter a word to find its frequency or line number, ensure there are no spaces before or after the word.

2- When processing large files with the application, the console's output may be so extensive that not all content is visible in the terminal. However, this does not imply that the application is unable to locate the content that is not displayed.

3- Using the application just needs placing the text file in the "Solution\Assessment1\bin\Debug\Resources" directory, and then changing the name of the text file in the "filePath" variable of the "Program" class to the name of the text file.

Implementation Details 1: Choice for the Project; Comparison between Data Structures (Other Optional Solutions for the Project)

In this project, a Binary Search Tree (BST) was selected as the primary data structure. Each node within the BST is utilized to store data and comprises five fields for this purpose. The fundamental characteristic of a BST is that, for any given node, words that precede others in lexicographic order are inserted into the left child node, while subsequent words are inserted into the right child node. So, there is no need to do extra work to order words in ascending or descending order.

- Time complexity of BST

  The time complexity of a binary search is contingent upon the structure of the tree, specifically its height, as operations necessitate traversing the tree's height in the worst-case scenario. In an ideally balanced binary search tree (BST), the height of the tree is logarithmic in relation to the number of nodes (n). Consequently, the time complexity of a balanced tree is approximately $O(\log_2(n))$. In the worst-case scenario, where the tree's height equals n (the number of nodes) due to the absence of branching, the time complexity becomes O(n).

Based on what have been discussed about time complexity of binary search tree, BSTs are search efficient. They allow to find a word in O(log n) time.

As in our application, insertion and searching are the main tasks, the time complexity of these operations in our data structure becomes important.

Other choices for data structures can be:

1- Static data structures are those with a fixed size, such as arrays. When addressing a problem with this approach, two arrays are needed: one for storing the unique values and another for storing the frequency of each word. The size of the first array determines the number of unique words.
   To sort the words in order, an iteration must compare each word to all others in the array. If a smaller value is found, the two values are exchanged. This process is repeated for all values from the first to the last index. The time complexity of this sorting method is approximately $O(n^2)$, which suggests the need to consider alternative solutions.
   Similarly, to find the longest word, each value's length must be compared to every other length, tracking the largest until all values have been evaluated. The main issue with using static data structures like arrays is their time inefficiency; in most scenarios, examining all values in the array is necessary to resolve the problem at hand.

2- Dictionary: Using a dictionary data structure, each key represents a unique word, and the associated value indicates its occurrence. Built-in methods such as “Count” can determine the number of unique words. For printing the words in an ordered fashion, methods like “OrderByDescending” are useful before displaying the keys (words). To identify the longest word, one can iterate through all the keys while keeping track of the longest word encountered thus far. To maintain access to the line numbers where words appear, a second dictionary can be defined, in which the keys are the words, and the values are lists of line numbers. The primary advantage of utilizing a dictionary is that it eliminates the need for searching through all the values, as it employs hash tables. However, storing hash tables can consume a significant amount of memory when dealing with large data sets.

3- Linked List: Designing a linked list involves defining nodes that contain four fields: the word, line numbers, frequency, and a link to the next node. The linked list class is responsible for managing these nodes and establishing connections between them. To print the words in order, one would iterate through the entire linked list, storing the words in a separate list before sorting them. Due to the linear increase in the number of comparisons as the list expands, the overall time complexity becomes $O(n^2)$. Furthermore, searching for a specific word necessitates traversing the linked list sequentially until the desired word is found, resulting in a time complexity of $O(n)$. This makes linked lists unsuitable for frequent lookups. Considering both time and space complexities, linked lists do not appear to be the best choice for applications, particularly when dealing with large text files.

The distinction between arrays, dictionaries, LinkedLists, and Binary Search Trees lies in the fact that arrays and dictionaries do not store the frequency and line number of words as fields within each node. Consequently, Binary Search Trees and LinkedLists tend to consume more memory, resulting in higher space complexity compared to arrays or dictionaries.

Here is a table summarizing all options:

Data Structure	**Pros**	**Cons**
Static data structure		Low Time Efficiency, steep learning curve
Dictionary	Fast lookups	No natural ordering, more memory usage
LinkedList	Good for sequential access	Slow searches, high memory usage
Binary Tree Search	Ordered data storage, efficient search	Unbalanced trees can degrade performance

Table 1: An overview of data structures and their properties

Implementation Details 2: Process of Implementation with BST

The application was developed using C# within the .NET framework. Below is a detailed overview of the development process:

In addition to the "Program" class, which contains the "Main" method that initiates the project, two additional classes are defined: "Node" and "BinarySearchTree." The "Node" class serves as the fundamental building block of the "BinarySearchTree" class.

The "Node" class is for creating node objects that store the unique words from the text file. Each distinct word in the file is assigned to a corresponding node in the tree, meaning the total number of unique words matches the number of nodes in the tree. Each node contains five fields:

- _word: A responsible string that represents the word stored at this node.
- frequency: An integer that indicates how many times the word appears.
- lineNumbers: A list of integers that captures the line numbers where the word is found in the text file.
- left and right: References to the left and right child nodes of the binary search tree (BST).

Each node in the tree has a property method that allows for easy access and modification.

The "BinarySearchTree" class is designed to manage operations on an arbitrary binary search tree. The methods included in this class are represented in features:

Feature 1: Store All Unique Words

The process involves the "InsertWordsfromFile" method and a supporting "Insert" method. Initially, words are extracted from lines of text, and specific conditions are evaluated to determine which words should be retained. Once these conditions are met, the chosen words are inserted into a Binary Search Tree (BST) through the helper "Insert" method.

Feature 2: Display the Number of Unique Words

The "CountUniqueWords" method is employed in this scenario. It utilizes recursion to navigate through the entire tree, incrementing the count each time a node is encountered. Ultimately, the final count will represent either the total number of nodes or the number of unique words within the file.

Feature 3: Store the Number of Occurrences of Each Word

The occurrences of each word are tracked by a dedicated field for each node. This takes place during the implementation of feature 1. When a word is imported, the system checks if an identical word already exists in the tree. If it does, the frequency of that word is incremented.

Feature 4: Display all the words (and the number of occurrences) – any order.

The "DisplayAllWords" method utilizes two recursive calls to print the elements in an "InOrder" fashion. First, it reaches the leftmost node, which is printed along with its occurrence. Then, it processes the current node of the leftmost node, followed by the right child of the current node. This process continues until the entire tree has been traversed.

Feature 5: Display all the words (and the number of occurrences) in-order (either ascending or descending alphabetical order).

The "DisplayAllWordsDescending" method closely resembles the previous method, with a minor adjustment in the order of the recursive calls. It first reaches and prints the rightmost node, followed by the current node, and finally the left nodes. This approach creates a reverse in-order traversal, resulting in the words being printed in descending order.

Feature 6: Output the Longest Word and the Number of Occurrences

The "DisplayLongestWord" method traverses the entire tree from the left to the right nodes, starting from the root node. As the traversal progresses, it updates the "longest" node variable with the values of "leftlongest" and "rightlongest." This process continues until the end of the recursion, at which point the final result of the longest node is returned.

Feature 7: Output the most frequent word stored and the number of occurrences.

The methodology employed in this instance mirrors that utilized in feature 6; however, the focus shifts from analyzing the length of words to evaluating their frequency. This comparative analysis of word frequency enables a deeper understanding of linguistic patterns and their implications within the given context.

Feature 8: Given a specific word, output the line numbers where the word was found.

In the “DisplayWordLineNumbers” method, the current root node is compared with the specified node. Based on the comparison result, the data is split in half. This recursive process continues until the comparison yields 0, indicating that the two words being compared are equal and the target word has been located. At this point, the found node is returned, and the recursion concludes. This approach differs from other methods that do not include a "return" statement in their recursive calls, as the result is returned immediately here, rather than being deferred to the end of the recursion.

Feature 9: Give a specific word, output the word’s frequency.

The method used here is similar to that employed in feature 8, with a key difference: we output the frequency of the specified word instead of its line numbers.

In the “Program” class, a helper method called “InsertWordsFromFile” is defined to read words from a text file and pass them to an instance of the “SearchBinaryTree” class.

The “Main” method contains all the operations related to the nine features.

Notes:

- Beside each one of the methods, there are helper methods to make recursion easier and organized.
- All fields within the classes are designated as "private," while each method is marked as "public" to establish an abstract data structure. This approach allows the structure to perform its tasks without permitting any alterations to the underlying details.

The code underwent a comprehensive refactoring to enhance its readability and maintainability. The method names were meticulously refined to facilitate improved comprehension of the application’s functionality.

Analysis of the most complex method

The method “DisplayWordLineNumbers” is expected to have the highest complexity since it requires traversing the tree until the user-specified word is located. To simplify the recursion process, we define a helper method called “DisplayWordLineNumbersRec,” which is responsible for outputting the node being searched for. If this node is not found, the console will display the message “word not found.”

The method “DisplayWordLineNumbersRec” is designed to search for a specified word within a tree structure, starting from the root node. It compares the root word with the user's input. If the root word is larger, the search continues in the left subtree; if it's smaller, the search moves to the right subtree. In the case of searching in the left subtree, this effectively eliminates the right subtree from consideration, thereby halving the search space with each recursion. The process continues until a match is found, indicated by a comparison result of zero. Once the word is located, the information is relayed to the “DisplayWordLineNumbers” function, and the node's “LineNumbers” property provides the count of lines where the word appears. The method returns results at each recursion level to ensure it stops upon finding the target node, rather than continuing to the leaf nodes of the tree.

The time complexity of the method is O(1 + m), where "m" represents the number of letters being compared between the two words. This is because the method must compare each character in both strings until a difference is found. In the worst-case scenario, the comparison will continue until the end of the shorter string is reached.

In the if statements, one condition is evaluated each time recursion occurs, resulting in behavior akin to that of a binary search tree. In this scenario, the best and worst case time complexities are O(log n) and O(n), respectively. Given that we encounter roughly the same number of letters, m, in each instance, the overall time complexity can be summarized as: $O(1+m) \times O(n) \approx O(mn)$.