Algorithms and Data Structures Assessment

Farima Torabzadeh – 24846939

May 2025

1. Overview

The project designed a Text Analysis tool that can process a text file to extract specific information about the words it contains. The algorithm includes identifying unique words, the longest and the most frequent word, and displaying all the words and their frequencies. It also can return a list of unique line numbers that a word had appeared in it.

Several data structures can be considered in the process of designing this tool, but the first three listed below are not suitable for handling the full functionality of the application.

- **Static Array:** although is the simplest structure, its predefined size is a limitation in dynamic applications. Searching and updating entries would be costly with complexity of O (n) and maintaining order is also another challenge.
- **Dictionary:** while it offers fast lookups and updates with complexity of O (1), traversing in them requires additional sorting steps.
- **Linked List:** although it provides dynamic sizing and straightforward insertion, has poor performance with complexity of O(n) and no considerations for component's order.
- **Binary Search Tree (BST):** selected due to its ability to efficiently insert, search, and support in-order traversal through its hierarchical data structure.

2. Methodology

Below is the description of each class in the application and the steps taken for their specific tasks.

2.1. `WordInfo` class

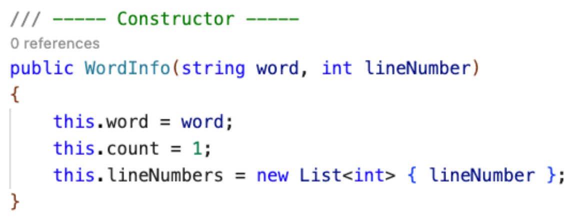
Serves as a container that stores all the statistical information related to each word.

It includes the variables below:

- `word`: the word string itself
- `count`: number of times the word appears
- `List<int> lineNumbers:` a list displaying the unique line numbers where the word was found. If a word appears multiple times on the same line, the first appearance is only recorded to avoid redundancy.

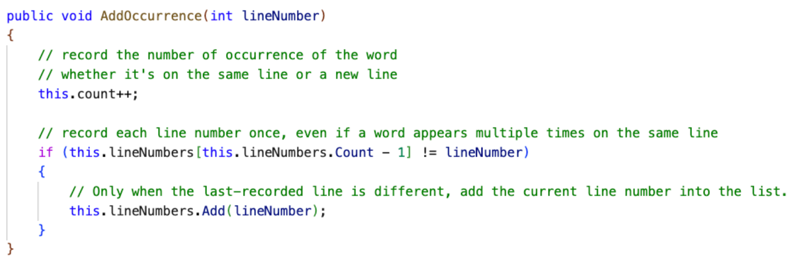
In addition to its constructor has this method:

- `AddOccurrence:` this method has two tasks
  - Adding to the word count
  - Recording the new line number in case it hasn't been added to the list `lineNumbers`



```
/// ----- Constructor -----
0 references
public WordInfo(string word, int lineNumber)
{
    this.word = word;
    this.count = 1;
    this.lineNumbers = new List<int> { lineNumber };
}
```

code snippet1. WordInfo constructor



```
public void AddOccurrence(int lineNumber)
{
    // record the number of occurrence of the word
    // whether it's on the same line or a new line
    this.count++;

    // record each line number once, even if a word appears multiple times on the same line
    if (this.lineNumbers[this.lineNumbers.Count - 1] != lineNumber)
    {
        // Only when the last-recorded line is different, add the current line number into the list.
        this.lineNumbers.Add(lineNumber);
    }
}
```

code snippet2. AddOccurrence method

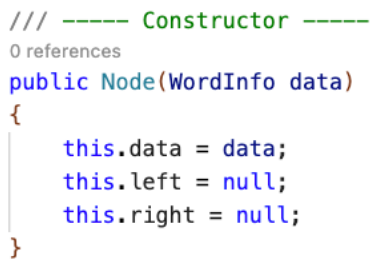
2.2. Node class

Represents nodes of the binary search tree which are its building blocks. Each node in the tree is actually a unique word which is extracted from the input.

It stores the object of WordInfo class, named data and includes the variables below:

- left: reference to the left child node, which stores the words that are alphabetically come before the current word
- right: reference to the right child node, which stores the words that are alphabetically come before the current word

Defining this class enables us to insert new words in their correct position based on their alphabetical order, and in-order traversing the tree.



```
/// ----- Constructor -----
0 references
public Node(WordInfo data)
{
    this.data = data;
    this.left = null;
    this.right = null;
}
```

code snippet3. Node constructor

In word insertion process, the tree compares the word alphabetically with the current node's word, there are 3 situations:

- same word, it updates the existing information in the WordInfo
- alphabetically less, it moves to the left subtree.
- alphabetically greater, it moves to the right subtree.

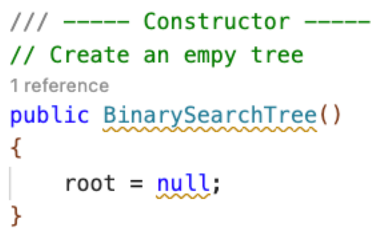
2.3. BinarySearchTree class

As the core data structure of the application, it is used to store all the unique words that appeared in the text file. Each node in the tree holds a WordInfo object, which represents a unique word and its related information. Storing the words in such structure enables us to avoid storing duplicate values, retrieve the words based on alphabetical order and without the need to sort them manually in a data structure like a list.

Primary variable, root, holds a reference to the topmost node in the tree, and all the insertion and traversal processes begin from this point.

Several methods are provided in this class:

- Insert: walks through the tree and decides to update the word in case it already exists in the tree or create a new node if the word is new.
- PreOrder, InOrder, PostOrder: display different traversal methods that return the lists of all words in different orders.



```
/// ----- Constructor -----
// Create an empy tree
1 reference
public BinarySearchTree()
{
    root = null;
}
```

code snippet4. BinarySearchTree constructor

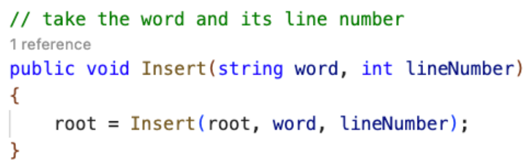
Two `Insert` functions are defined in this class.

- `public void Insert`, is called from other classes and provides the interface to start the recursive insertion process. It internally calls the private recursive `Insert` method, which performs the actual process of locating the words in their correct position in the tree.

- `private Node Insert`, has three inputs that are passed during recursion to navigate through the tree and find out where to insert the word:
  - `Node node`: the current node that is being inspected, which can be any position
  - `string word`: the new word that is being inserted or updated
  - `int lineNumber`: the line number that the word appeared in the text file

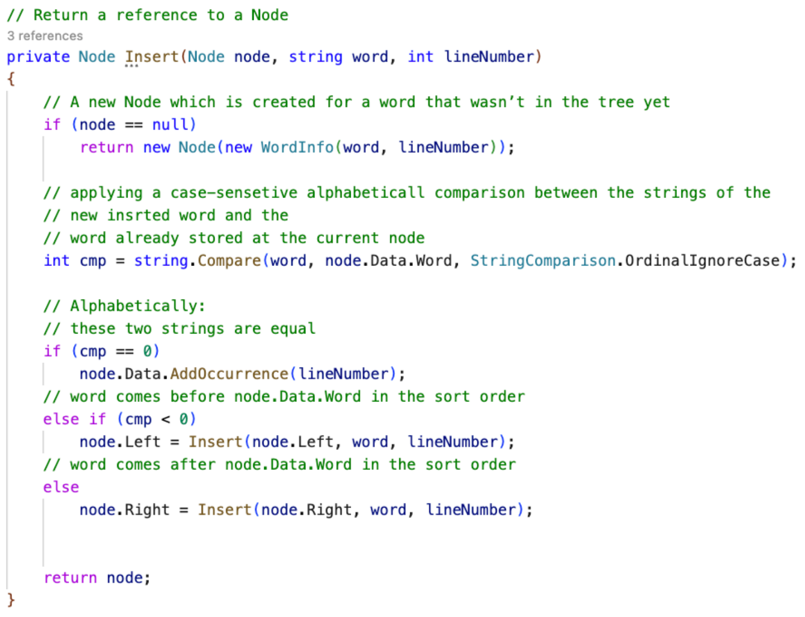
  The word that is going to be inserted can go into three processes:
  - If it is a new word, the method creates a new `Node` and returns it so it can be linked into the tree.
  - If the word already exists, it updates the existing `WordInfo` and returns the same node.
  - If the word belongs in the left or right subtree, it performs the insertion recursively and returns the updated child node, so the link in the parent node's `.Left` and `.Right` can be maintained.

  This function eventually returns a `Node`, which might be newly created or updated, leading the tree to maintain its correct structure.



```
// take the word and its line number
1 reference
public void Insert(string word, int lineNumber)
{
    root = Insert(root, word, lineNumber);
}
```

code snippet4. Public `Insert` method

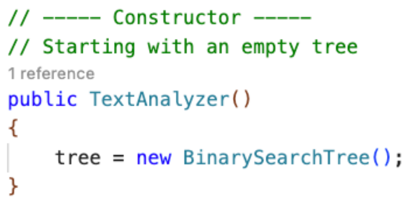


```
// Return a reference to a Node
3 references
private Node Insert(Node node, string word, int lineNumber)
{
    // A new Node which is created for a word that wasn't in the tree yet
    if (node == null)
        return new Node(new WordInfo(word, lineNumber));

    // applying a case-sensetive alphabeticall comparison between the strings of the
    // new insrted word and the
    // word already stored at the current node
    int cmp = string.Compare(word, node.Data.Word, StringComparison.OrdinalIgnoreCase);

    // Alphabetically:
    // these two strings are equal
    if (cmp == 0)
        node.Data.AddOccurrence(lineNumber);
    // word comes before node.Data.Word in the sort order
    else if (cmp < 0)
        node.Left = Insert(node.Left, word, lineNumber);
    // word comes after node.Data.Word in the sort order
    else
        node.Right = Insert(node.Right, word, lineNumber);


    return node;
}
```

code snippet5. Private `Insert` method

2.4. `TextAnalyzer` class

Serves as the coordinator of the application and handles the reading process of the text file. It extracts the words from each line and

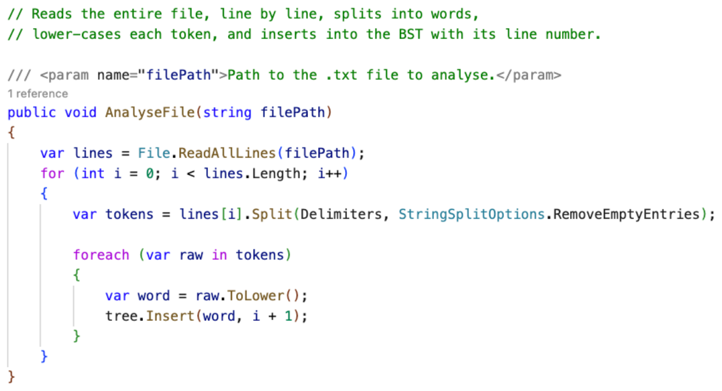


```
// ----- Constructor -----
// Starting with an empty tree
1 reference
public TextAnalyzer()
{
    tree = new BinarySearchTree();
}
```

code snippet4. **`TextAnalyzer` constructor**

Several methods are defined in this class besides the constructor:

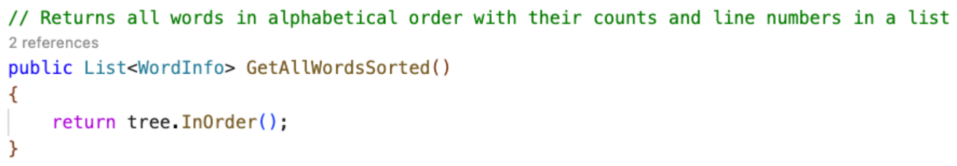
- `AnalyseFile`**:** reads the text file by using the `ReadAllLines` method, processes it line by line, extracts words and utilises the `Insert` method to insert the words into the tree and generally transforms the raw data into the structured data stored in the tree.



```
// Reads the entire file, line by line, splits into words,
// lower-cases each token, and inserts into the BST with its line number.

/// <param name="filePath">Path to the .txt file to analyse.</param>
1 reference
public void AnalyseFile(string filePath)
{
    var lines = File.ReadAllLines(filePath);
    for (int i = 0; i < lines.Length; i++)
    {
        var tokens = lines[i].Split(Delimiters, StringSplitOptions.RemoveEmptyEntries);

        foreach (var raw in tokens)
        {
            var word = raw.ToLower();
            tree.Insert(word, i + 1);
        }
    }
}
```

code snippet5. `AnalyseFile` method

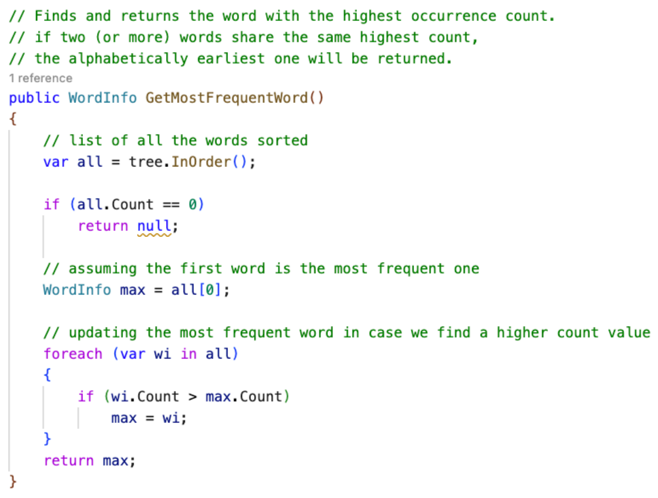
- `GetAllWordsSorted`**:** returns a list of all words which is sorted alphabetically by calling the `InOrder` method and performing an in-order traversal of the binary search tree.



```
// Returns all words in alphabetical order with their counts and line numbers in a list
2 references
public List<WordInfo> GetAllWordsSorted()
{
    return tree.InOrder();
}
```

code snippet6. `GetAllWordsSorted` method

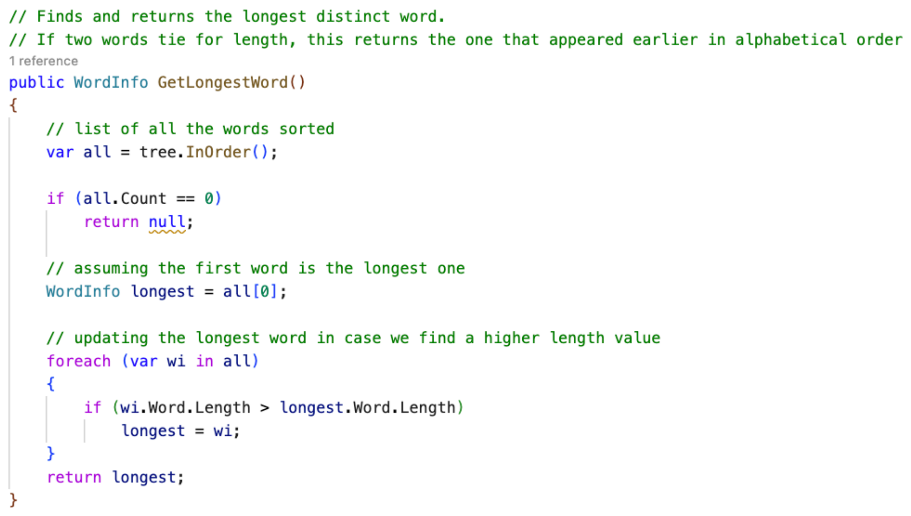
- `GetMostFrequentWord`**:** by using the output of the `GetAllWordsSorted` method and getting a list of all words, iterates through the list and finds the word with the highest number of occurrences.



```
// Finds and returns the word with the highest occurrence count.
// if two (or more) words share the same highest count,
// the alphabetically earliest one will be returned.
1 reference
public WordInfo GetMostFrequentWord()
{
    // list of all the words sorted
    var all = tree.InOrder();

    if (all.Count == 0)
        return null;

    // assuming the first word is the most frequent one
    WordInfo max = all[0];

    // updating the most frequent word in case we find a higher count value
    foreach (var wi in all)
    {
        if (wi.Count > max.Count)
            max = wi;
    }
    return max;
}
```

code snippet6. `GetMostFrequentWord` method

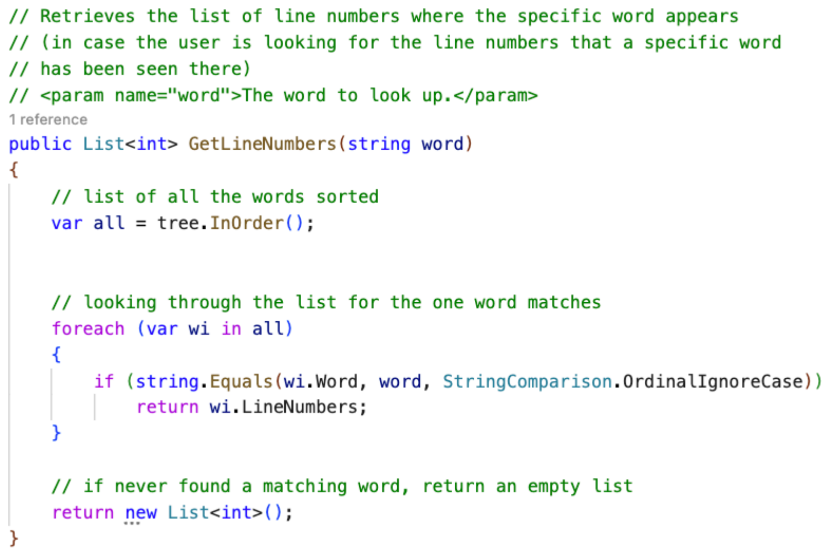
- `GetLongestWord:` by using the output of the `GetAllWordsSorted` method and get a list of all words, iterates through the list and compare the word lengths to find the word with the greatest character length.



```
// Finds and returns the longest distinct word.
// If two words tie for length, this returns the one that appeared earlier in alphabetical order
1 reference
public WordInfo GetLongestWord()
{
    // list of all the words sorted
    var all = tree.InOrder();

    if (all.Count == 0)
        return null;

    // assuming the first word is the longest one
    WordInfo longest = all[0];

    // updating the longest word in case we find a higher length value
    foreach (var wi in all)
    {
        if (wi.Word.Length > longest.Word.Length)
            longest = wi;
    }
    return longest;
}
```

code snippet7. `GetLongestWord` method

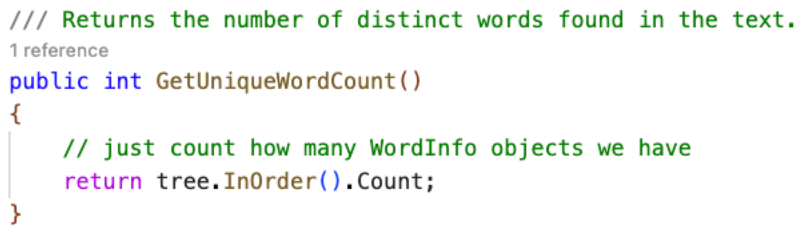
- `GetLineNumbers:` returns a list of line numbers that the given word appears by calling the `InOrder` method and looking for the word that matches the input.



```
// Retrieves the list of line numbers where the specific word appears
// (in case the user is looking for the line numbers that a specific word
// has been seen there)
// <param name="word">The word to look up.</param>
1 reference
public List<int> GetLineNumbers(string word)
{
    // list of all the words sorted
    var all = tree.InOrder();


    // looking through the list for the one word matches
    foreach (var wi in all)
    {
        if (string.Equals(wi.Word, word, StringComparison.OrdinalIgnoreCase))
            return wi.LineNumbers;
    }

    // if never found a matching word, return an empty list
    return new List<int>();
}
```

code snippet8. `GetLineNumbers` method

- `GetUniqueWordCount:` returns the total number of unique words found in the text by calling the `tree.InOrder().Count;`



```
/// Returns the number of distinct words found in the text.
1 reference
public int GetUniqueWordCount()
{
    // just count how many WordInfo objects we have
    return tree.InOrder().Count;
}
```

code snippet9. `GetUniqueWordCount` method

- `GetAllWordsUnsorted:` returns a list of all words in the order they were first inserted into the tree by calling the `PreOrder` method**.**

```
/// Returns all words in “insertion” (pre-order) order.
1 reference
public List<WordInfo> GetAllWordsUnsorted()
{
    // tree.PreOrder() returns a List<WordInfo> in the order nodes were first inserted
    return tree.PreOrder();
}
```

code snippet10. `GetAllWordsUnsorted` method

2.5. `Program` class

The main role of this class is to guide the user through the analysis process of a text file. The application begins by asking the user to provide the path of the text file, then hands the file to the `TextAnalyzer` class. After the file has been processed, it presents a designed menu system as the interface that the user can explore various information.

Behind the simple menu displayed to the user is a switch statement that maps each entered numbered option to a specific task. When the user presses a number, the program calls the corresponding method from the `TextAnalyzer` class and displays the result.

```
// Prompt for the input file path at startup
Console.Write("Enter full path to text file: ");
string filePath = Console.ReadLine()                  // read raw input
                      .Trim()
                      .Trim('"', '\'');           // remove surrounding " or '

if (!File.Exists(filePath))
{
    Console.WriteLine($"Error: File not found: {filePath}");
    return;
}
```

code snippet11. File path input

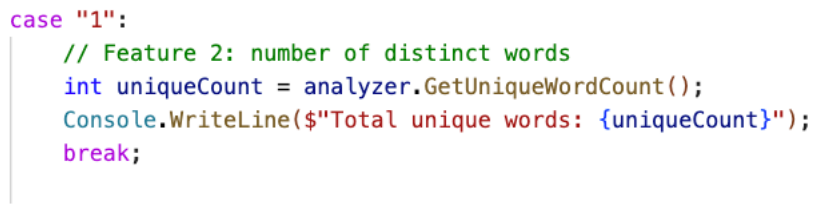
To start the analysis part, a new instance of the `TextAnalyzer` class is created, and the input file will be passed to it for the analysis.

```
// Analyse immediately
var analyzer = new TextAnalyzer();
analyzer.AnalyseFile(filePath);
Console.WriteLine("File loaded and analysed successfully.\nPress Enter to continue...");
Console.ReadLine();
```

code snippet12. File path input validation

In this section, each menu option is broken down by presenting its corresponding case in the code, followed by an explanation of its functionality and how it is linked to the relevant methods in the `TextAnalyzer` class. When the user enters numbers 1-8 from the console menu, the switch block matches it with the relevant case.

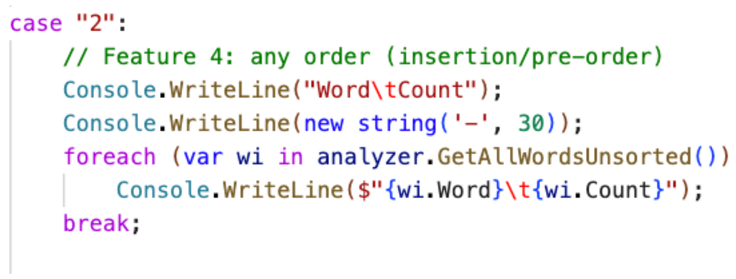
- **Case 1.** Calls the method `GetUniqueWordCount` which belongs to the `TextAnalyzer` object `analyzer`**.** When the `GetUniqueWordCount` is called, it performs an in-order traversal of the tree via `tree.InOrder().Count which returns a List<WordInfo>`. In this list, each component is a `WordInfo` and by using the `.Count` property, we can extract the number of unique words in it.



```
case "1":
    // Feature 2: number of distinct words
    int uniqueCount = analyzer.GetUniqueWordCount();
    Console.WriteLine($"Total unique words: {uniqueCount}");
    break;
```

code snippet13. Number of unique words

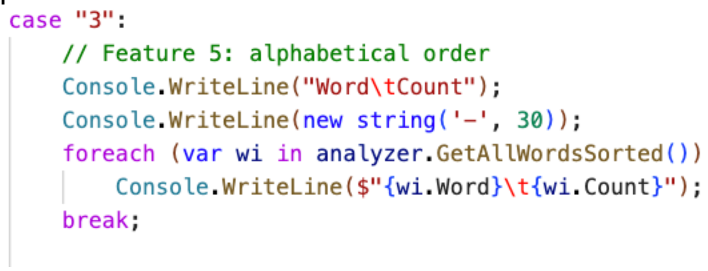
- **Case 2.** Calls the method `GetAllWordsUnsorted` from the `TextAnalyzer` instance. This method is used to retrieve a list of all the words inserted into the binary search tree. Internally `GetAllWordsUnsorted`, performs a pre-order traversal of the binary tree. The returned list is a `List<WordInfo>`. The loop iterates through this list and prints each word and how many times it appeared by using `Word` and `Count` methods defined in the `WordInfo` class.



```
case "2":
    // Feature 4: any order (insertion/pre-order)
    Console.WriteLine("Word\tCount");
    Console.WriteLine(new string('-', 30));
    foreach (var wi in analyzer.GetAllWordsUnsorted())
        Console.WriteLine($"{wi.Word}\t{wi.Count}");
    break;
```

code snippet14. All words and their frequency (in insertion order)

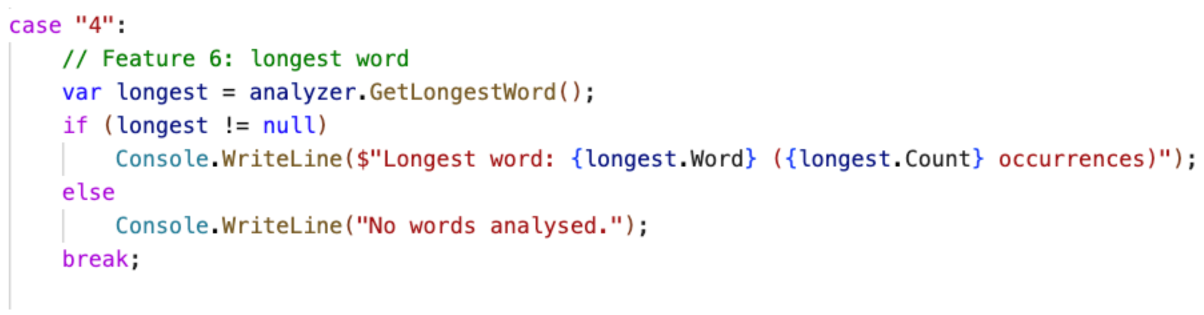
- **Case 3.** The same process is followed as in Case 2, but the methods called differ. `GetAllWordsSorted` uses in-order traversal for alphabetical output, while Case 2 uses pre-order traversal for insertion order.



```
case "3":
    // Feature 5: alphabetical order
    Console.WriteLine("Word\tCount");
    Console.WriteLine(new string('-', 30));
    foreach (var wi in analyzer.GetAllWordsSorted())
        Console.WriteLine($"{wi.Word}\t{wi.Count}");
    break;
```

code snippet15. All words and their frequency (in alphabetical order)

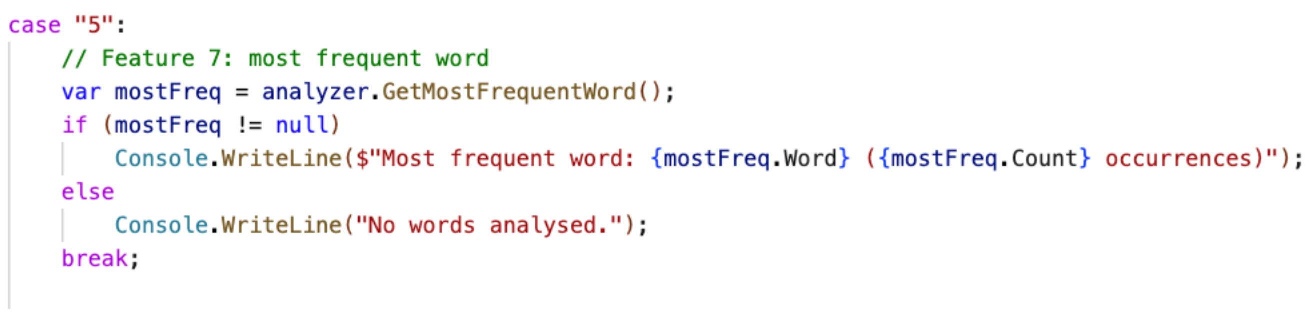
- **Case 4.** Calls the GetLongestWord method from the TextAnalyzer instance. Inside the TextAnalyzer, tree.InOrder() is called to get all words in alphabetical order. This method iterates through the list, compares word lengths by using the Length property, and the longest word returned as a WordInfo object. The result is displayed in the console, showing both the word and how many times it appeared.



```
case "4":
    // Feature 6: longest word
    var longest = analyzer.GetLongestWord();
    if (longest != null)
        Console.WriteLine($"Longest word: {longest.Word} ({longest.Count} occurrences)");
    else
        Console.WriteLine("No words analysed.");
    break;
```

code snippet16. The longest word and its frequency

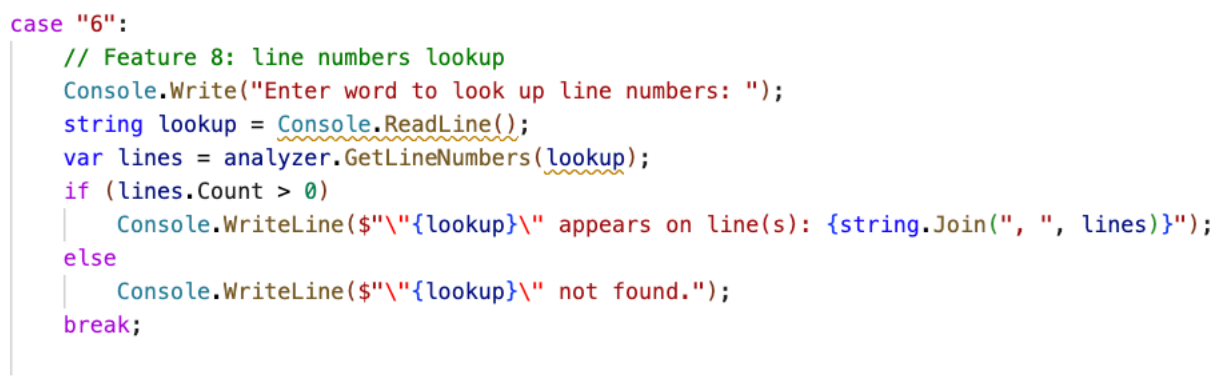
- **Case 5.** Follows the same structure as Case 4, but in the final stage, it accesses the Count method of each WordInfo object to compare word frequencies.



```
case "5":
    // Feature 7: most frequent word
    var mostFreq = analyzer.GetMostFrequentWord();
    if (mostFreq != null)
        Console.WriteLine($"Most frequent word: {mostFreq.Word} ({mostFreq.Count} occurrences)");
    else
        Console.WriteLine("No words analysed.");
    break;
```

code snippet17. The most frequent word and its frequency

- **Case 6.** When the user selects this option, the program prompts them to enter a specific word. The input is passed to the GetLineNumbers method in the TextAnalyzer class. The method then searches the tree for a WordInfo object that matches with the word the user entered as input and returns a list of line numbers that the word was appeared there.



```
case "6":
    // Feature 8: line numbers lookup
    Console.Write("Enter word to look up line numbers: ");
    string lookup = Console.ReadLine();
    var lines = analyzer.GetLineNumbers(lookup);
    if (lines.Count > 0)
        Console.WriteLine($"\"{lookup}\" appears on line(s): {string.Join(", ", lines)}");
    else
        Console.WriteLine($"\"{lookup}\" not found.");
    break;
```

code snippet18. Line numbers a given word by user is appeared

- **Case 7.** Prompts the user to enter a word, similar to Case 6. The program calls `GetAllWordsSorted` and then applies `FirstOrDefault` to search for the first `WordInfo` which its word matches the input word. If a match is found, the program prints its frequency; otherwise, it shows a “not found message”.

```
case "7":
    // Feature 9: frequency lookup
    Console.Write("Enter word to look up frequency: ");
    string lookupFreq = Console.ReadLine();
    var info = analyzer.GetAllWordsSorted()
                        .FirstOrDefault(wi =>
                            string.Equals(wi.Word, lookupFreq, StringComparison.OrdinalIgnoreCase));
    if (info != null)
        Console.WriteLine($"\"{lookupFreq}\" occurs {info.Count} time(s).");
    else
        Console.WriteLine($"\"{lookupFreq}\" not found.");
    break;

default:
    Console.WriteLine("Invalid choice. Please select a number between 1 and 8.");
    break;
```

code snippet19. Total frequency of a given word by user

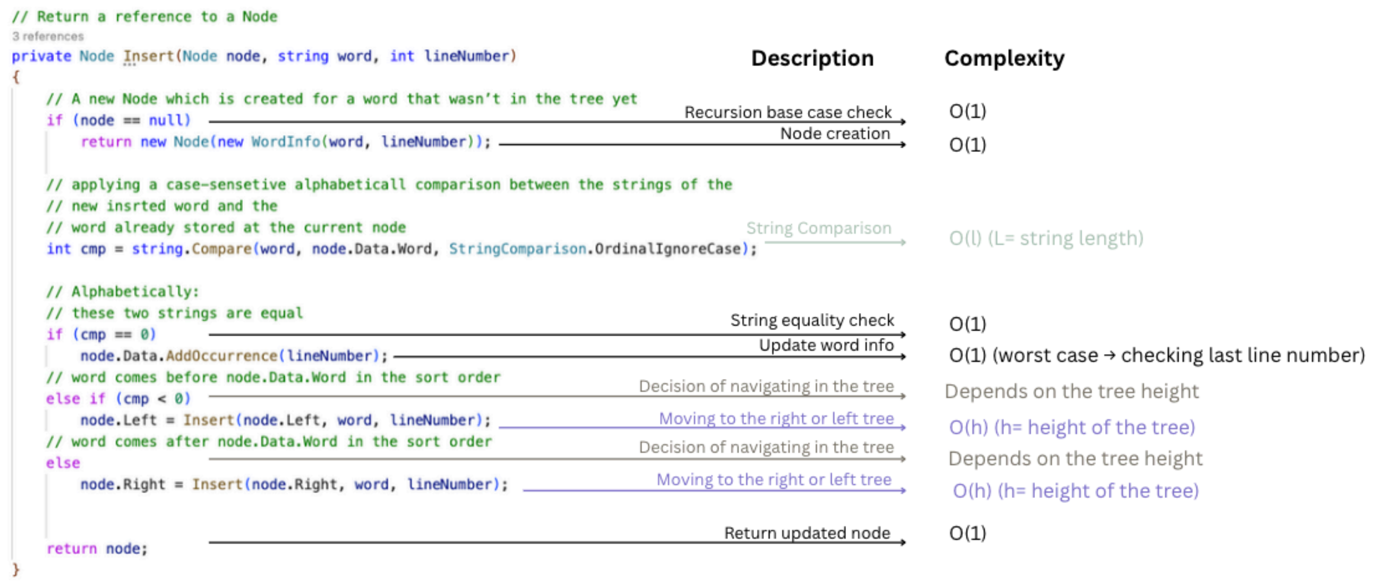
- **Case 8.** The program breaks out of the main menu loop, which terminates the menu system and moves on to show the goodbye message.

```
if (choice == "8")
    break;
```

code snippet20. Exit

3. Computational Complexity Analysis

`Insert` method has a central role in the tree construction process and the highest algorithmic complexity in the application due to its recursive structure.



```
// Return a reference to a Node
3 references
private Node Insert(Node node, string word, int lineNumber)
{
    // A new Node which is created for a word that wasn't in the tree yet
    if (node == null)
        return new Node(new WordInfo(word, lineNumber));

    // applying a case-sensetive alphabeticall comparison between the strings of the
    // new insrted word and the
    // word already stored at the current node
    int cmp = string.Compare(word, node.Data.Word, StringComparison.OrdinalIgnoreCase);

    // Alphabetically:
    // these two strings are equal
    if (cmp == 0)
        node.Data.AddOccurrence(lineNumber);
    // word comes before node.Data.Word in the sort order
    else if (cmp < 0)
        node.Left = Insert(node.Left, word, lineNumber);
    // word comes after node.Data.Word in the sort order
    else
        node.Right = Insert(node.Right, word, lineNumber);

    return node;
}
```

Code	Description	Complexity
`if (node == null)`	Recursion base case check	O(1)
`return new Node(new WordInfo(word, lineNumber));`	Node creation	O(1)
`int cmp = string.Compare(word, node.Data.Word, StringComparison.OrdinalIgnoreCase);`	String Comparison	O(l) (L= string length)
`if (cmp == 0)`	String equality check	O(1)
`node.Data.AddOccurrence(lineNumber);`	Update word info	O(1) (worst case → checking last line number)
`else if (cmp < 0)`	Decision of navigating in the tree	Depends on the tree height
`node.Left = Insert(node.Left, word, lineNumber);`	Moving to the right or left tree	O(h) (h= height of the tree)
`else`	Decision of navigating in the tree	Depends on the tree height
`node.Right = Insert(node.Right, word, lineNumber);`	Moving to the right or left tree	O(h) (h= height of the tree)
`return node;`	Return updated node	O(1)

Fig1. Line by line complexity breakdown

Regarding the analysis above, overall time complexity of the `Insert` method depends on two main factors of tree height and word string length.

- **Best-case:**
  In a **balanced binary tree**, the tree height = O (Log n), where n is the number of nodes. Also, the absolute difference between the heights of the left and right subtrees at any node should be no more than 1.

  In our case, inserting a new word or updating an existing word needs traversing a path from the root to a leaf or intermediate node (O (Log n)). Since each step involves comparing strings of average length, the overall complexity becomes O (l * Log n)

- **Worst-case:**
  In contrast, an **unbalanced tree** occurs when nodes are inserted in a sorted or nearly sorted order, causing the tree to turn into a linear structure with height O (n). This leads to a worst-case complexity of O (l * n).