Text Analysis tool

6G7V0008 Algorithms & Data Structures

Ruchi Rathod - 24801180

Introduction

The text analysis tool, developed in C#, processes a provided text file while giving multiple analyses through operations on these words. The core logic revolves around the selected data structure, which is Linked list, that efficiently works on operations such as storing unique words, tracking their frequencies, identifying the longest and most frequent words, and searching for specific words and their line occurrences. Below is an in-depth explanation of how this application functions and the logic behind each feature. This report also includes a comparative analysis of static data structures, dictionaries, linked lists, and binary search trees as potential alternatives, and provides reasoning for the selection of the linked list in this project.

Overview of the System Design

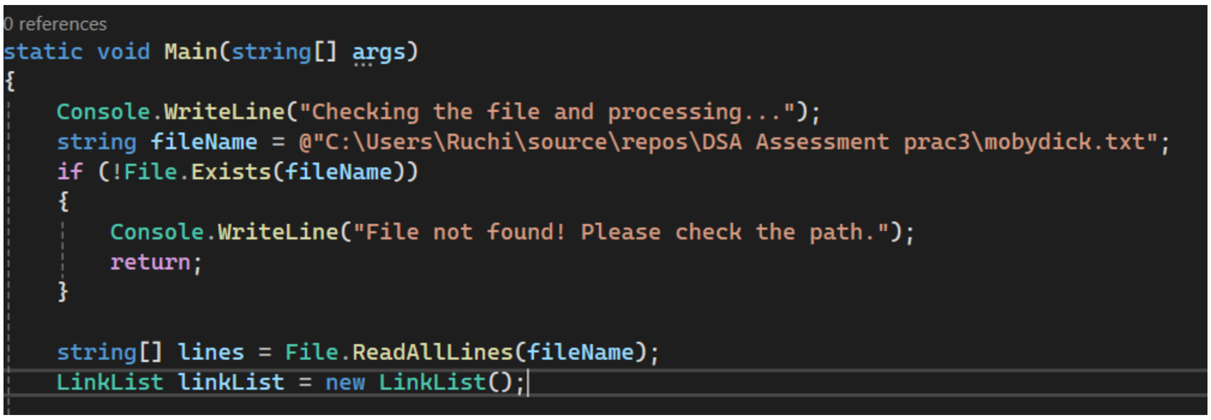
The application is designed to explore and gain insights into text data by providing multiple features for word analysis. The system which is created for this project is divided into three main parts:

1. **Program.cs**: This file is the main part of the application that runs everything. It starts the program, reads the text file, and shows a menu(using switch case) so the user can choose what they want to do. Based on the user's choice, it calls the right parts of the code to do the work and shows the results.
2. **Link.cs**: Defines the structure of the Link class, each link represents a word in the text and stores the word along with its frequency and the line numbers where it appears. The class is designed to be part of a linked list that holds all the words in the text.
3. **LinkList.cs**: This file is responsible for organizing and managing the list of words in the application. It keeps track of each word, allows you to add new words to the list, shows all the words when needed, and helps you sort them. Additionally, it can find the most frequent word or the longest word in the list.

Explanation of the program

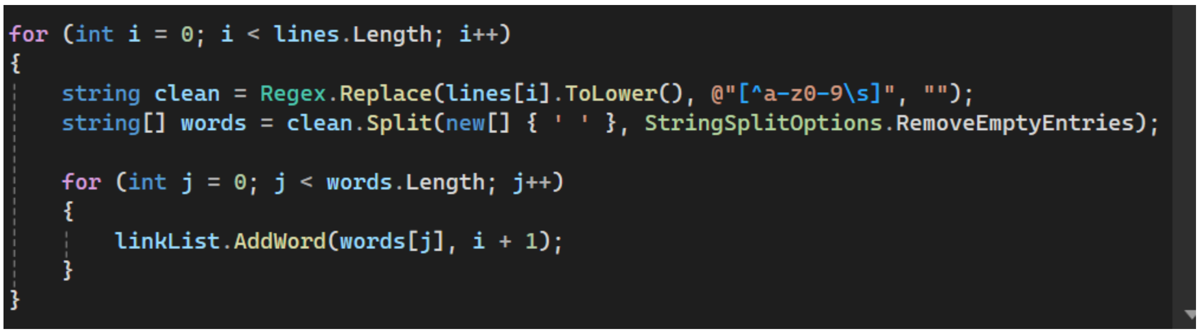
Program.cs

Reading the Text File: First step is reading the content of a text file into an array of strings. Each string represents one line of the text file. If the file does not exist, an error message is displayed, and the program exits early.



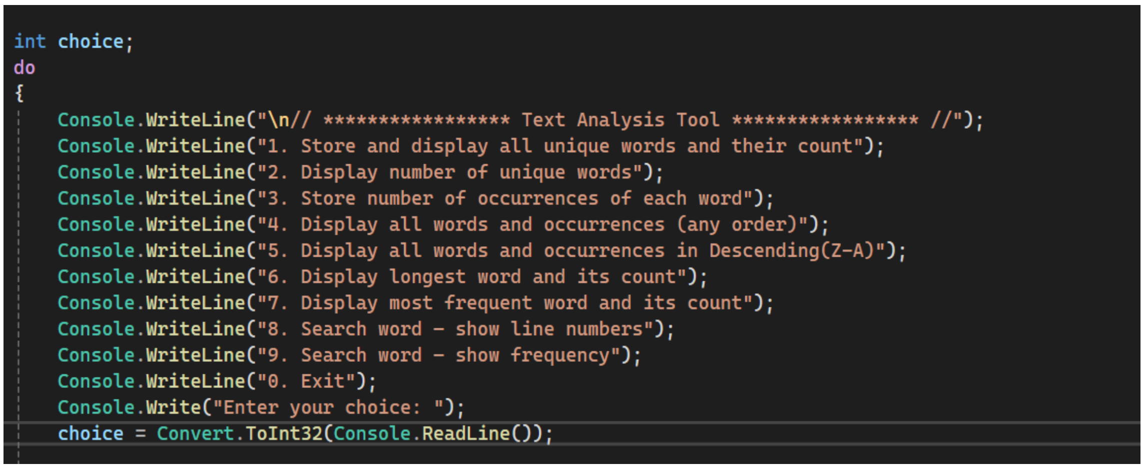
```
0 references
static void Main(string[] args)
{
    Console.WriteLine("Checking the file and processing...");
    string fileName = @"C:\Users\Ruchi\source\repos\DSA Assessment prac3\mobydick.txt";
    if (!File.Exists(fileName))
    {
        Console.WriteLine("File not found! Please check the path.");
        return;
    }

    string[] lines = File.ReadAllLines(fileName);
    LinkList linkList = new LinkList();
```

Processing the Words: The program then processes the text by looping through each line, cleaning the line to remove any special characters, and converting all characters to lowercase. It uses regular expressions to clean each line. This ensures that word comparisons are case-insensitive and that only the words themselves are stored. After cleaning the line, each line then is split into individual words. For each word, the program calls the AddWord method from the LinkList class to store it in the linked list. If the word is already present, the program adds the current line number to the list of line numbers where the word appears.

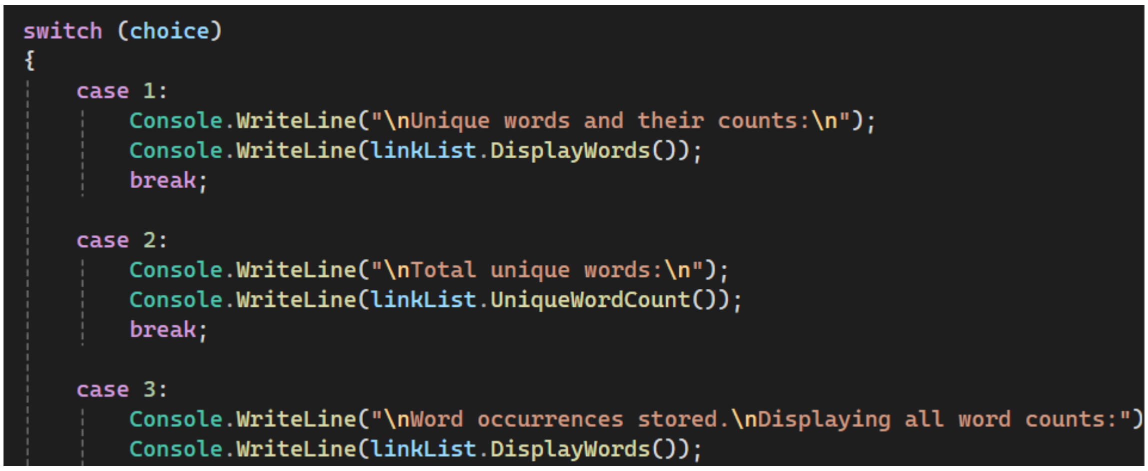


```
for (int i = 0; i < lines.Length; i++)
{
    string clean = Regex.Replace(lines[i].ToLower(), @"[^a-z0-9\s]", "");
    string[] words = clean.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

    for (int j = 0; j < words.Length; j++)
    {
        linkList.AddWord(words[j], i + 1);
    }
}
```

User Interaction: For this part of the program, switch case function has been used for the interactive menu. where users can choose different options to perform word analysis. The options include displaying unique words, counting word occurrences, sorting words, searching for a word's frequency, and more. The program continuously prompts for input until the user chooses to exit.



```
int choice;
do
{
    Console.WriteLine("\n// **************** Text Analysis Tool **************** //");
    Console.WriteLine("1. Store and display all unique words and their count");
    Console.WriteLine("2. Display number of unique words");
    Console.WriteLine("3. Store number of occurrences of each word");
    Console.WriteLine("4. Display all words and occurrences (any order)");
    Console.WriteLine("5. Display all words and occurrences in Descending(Z-A)");
    Console.WriteLine("6. Display longest word and its count");
    Console.WriteLine("7. Display most frequent word and its count");
    Console.WriteLine("8. Search word - show line numbers");
    Console.WriteLine("9. Search word - show frequency");
    Console.WriteLine("0. Exit");
    Console.Write("Enter your choice: ");
    choice = Convert.ToInt32(Console.ReadLine());
```



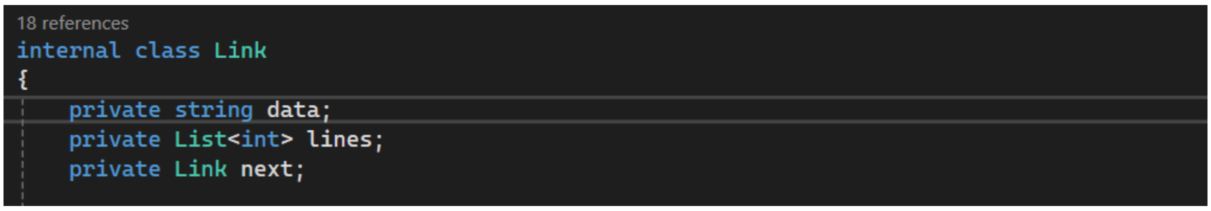
```
switch (choice)
{
    case 1:
        Console.WriteLine("\nUnique words and their counts:\n");
        Console.WriteLine(linkList.DisplayWords());
        break;

    case 2:
        Console.WriteLine("\nTotal unique words:\n");
        Console.WriteLine(linkList.UniqueWordCount());
        break;

    case 3:
        Console.WriteLine("\nWord occurrences stored.\nDisplaying all word counts:")
        Console.WriteLine(linkList.DisplayWords());
```

Link.cs

The class Link represents a single node in a linked list. Each node stores a word, keeps track of the line numbers where the word appears, and holds a reference to the next node. This structure helps in organizing and accessing words one after another in sequence.

<u>internal class Link</u>

This defines a class Link. It is marked as internal, meaning it can only be accessed within the same project. The class represents a single node in a linked list, used to store one word from the text, the lines it appears on, and a reference to the next word in the list.

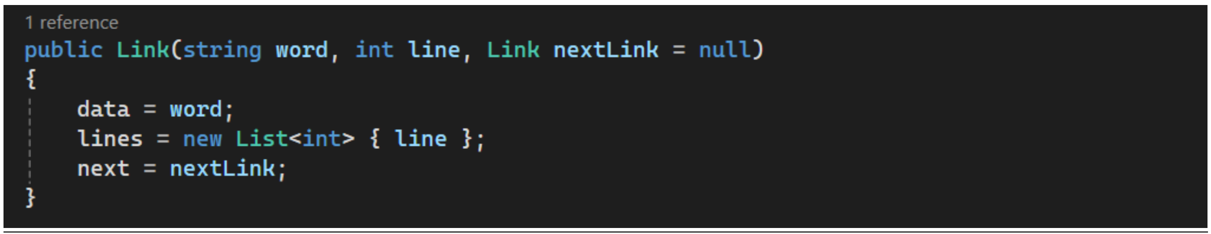


```
18 references
internal class Link
{
    private string data;
    private List<int> lines;
    private Link next;
```

public Link()

This is the constructor. It's the function that runs when a new Link object is created.

It takes three parameters, the word to store in this link, the line number where the word was found and the next Link object this one should point to (this is optional and defaults to null if not provided).

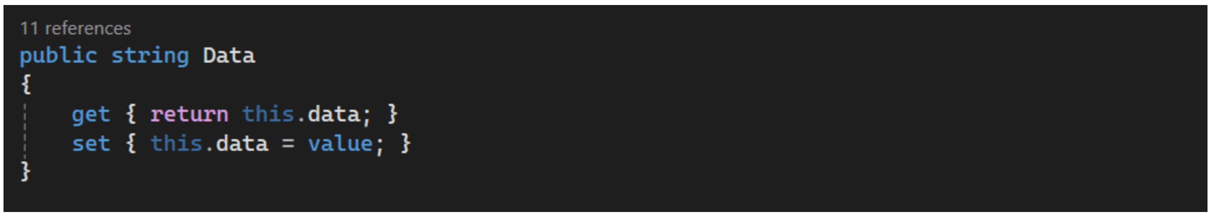


```
1 reference
public Link(string word, int line, Link nextLink = null)
{
    data = word;
    lines = new List<int> { line };
    next = nextLink;
}
```

Word is stored in data, lines have a new list of integers created and immediately adding one number to it, new List<int>() creates a new, empty list that can hold numbers (in this case, line numbers). { line } adds the value of line (which was passed to the constructor) as the first item in that new list. whereas the next variable is set to point to the next link (or to null if it's the last one in the chain).

private string data

This is a property called Data that holds the actual word stored in the node. Get set allows to read(get) and update(set) the word stored in this node.



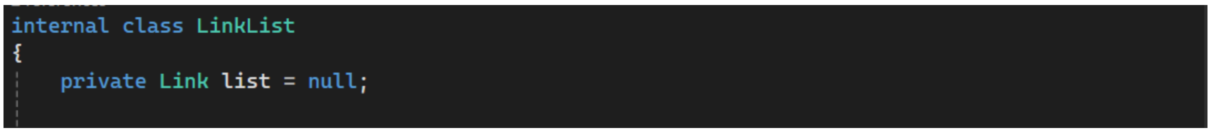
```
11 references
public string Data
{
    get { return this.data; }
    set { this.data = value; }
}
```

Rest of the properties does the same present in this class List.

LinkList

private Link list = null;

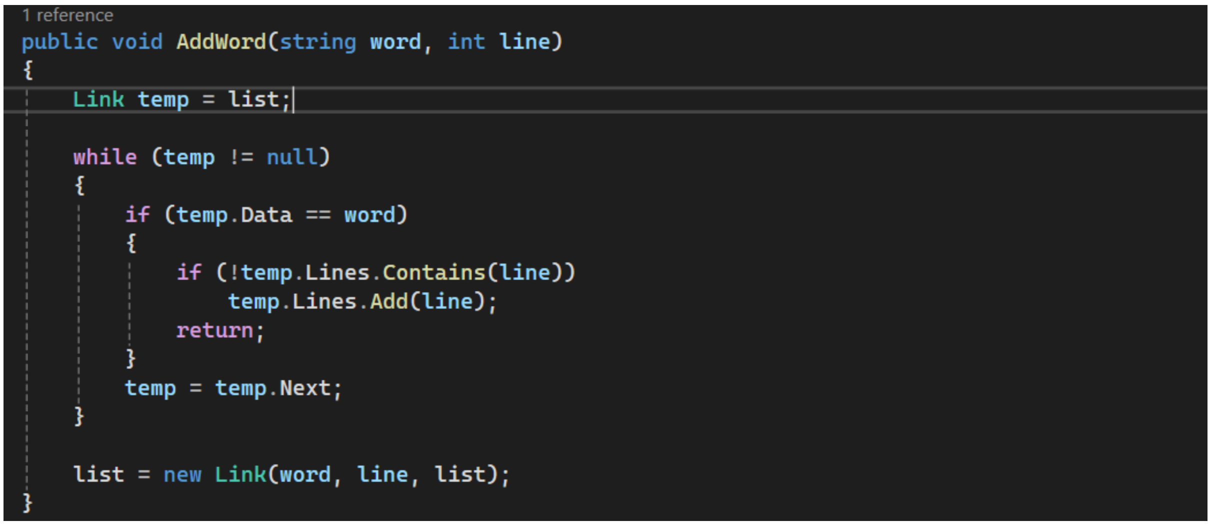
This is where your list of words starts. At first, it is empty (null). Every word you add will become a new "link" in this chain.



```
internal class LinkList
{
    private Link list = null;
```

AddWord Method

The AddWord method adds a word to the linked list, checking if the word already exists. If the word is found in the list, it adds the current line number to the lines list of that word, provided it is not already there. If the word is not found, a new Link object is created and added at the front of the linked list.



```
1 reference
public void AddWord(string word, int line)
{
    Link temp = list;

    while (temp != null)
    {
        if (temp.Data == word)
        {
            if (!temp.Lines.Contains(line))
                temp.Lines.Add(line);
            return;
        }
        temp = temp.Next;
    }

    list = new Link(word, line, list);
}
```

Link temp = list; - Started by pointing temp to the start of the linked list. If this is the first word, list is null, so nothing will be searched.

while (temp != null) - Here, the loop goes through each node in the list until it reaches the end that is null.

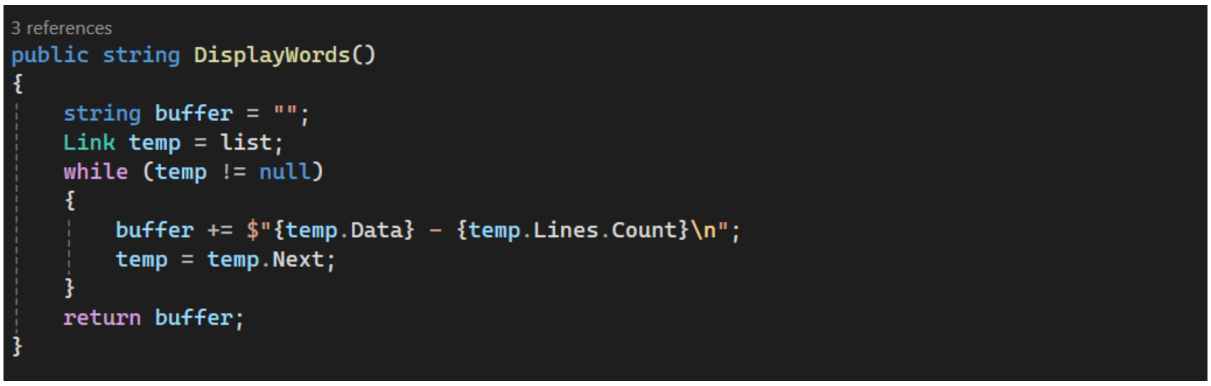
```
if (temp.Data == word)
{
    if (!temp.Lines.Contains(line))
        temp.Lines.Add(line);
    return;
}
temp = temp.Next;
```

This loop first checks if the current node matches the word which we are adding. If not in the list already then add it, if the word is in the list we skip, to avoid duplicates.

list = new Link(word, line, list); - add the word and line number if not already exist. It links the new word to the existing list, placing it at the very front so it becomes the new starting point.

DisplayWords Methods

Display all words and their occurrence counts: The DisplayWords method is used to traverse the entire linked list and display each word along with its frequency. When a file is processed, each word and its corresponding line number are stored as a node (Link) in the linked list. The DisplayWords method is used to look through this entire list from start to end and print out each word along with how many lines it appears on.

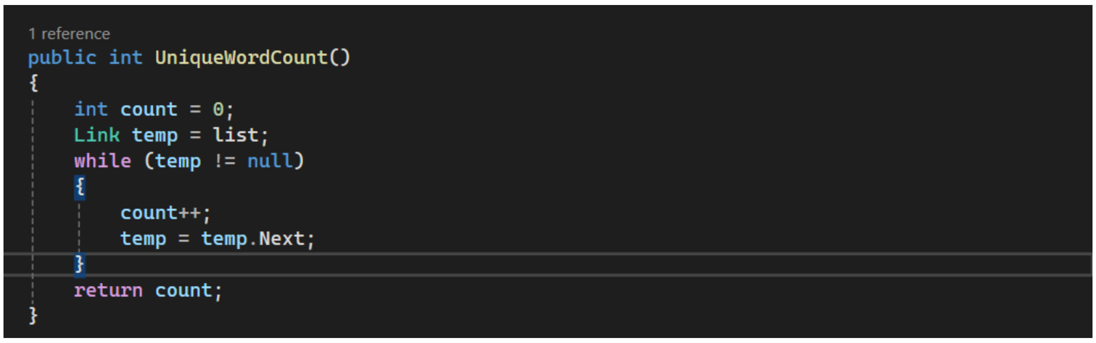


```
3 references
public string DisplayWords()
{
    string buffer = "";
    Link temp = list;
    while (temp != null)
    {
        buffer += $"{temp.Data} - {temp.Lines.Count}\n";
        temp = temp.Next;
    }
    return buffer;
}
```

```
public string DisplayWords()
 {
     string buffer = "";          → This will store the final result to return
     Link temp = list;            → Start from the beginning (head) of the linked
list
     while (temp != null)         → Loop through the whole list
     {
         buffer += $"{temp.Data} - {temp.Lines.Count}\n";
→ Add word and how many lines it appears on
         temp = temp.Next;        → Move to the next node in the list
     }
     return buffer;               → Return the complete string
 }
```

UniqueWordCount Method

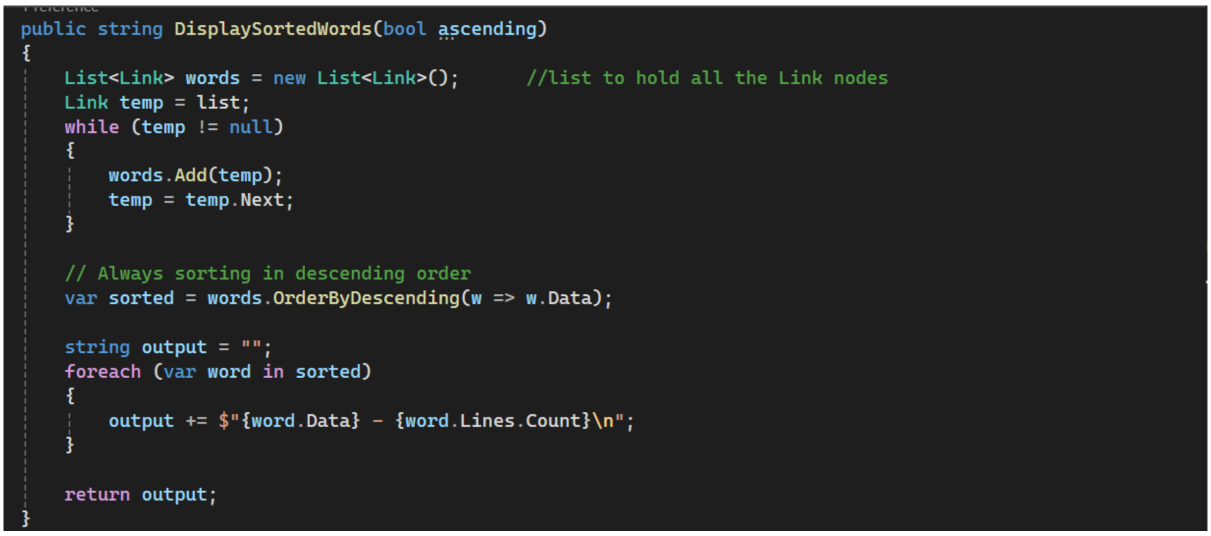
This method counts the total number of unique words by traversing the linked list and incrementing a counter for each node. This is how it works, it goes through the entire list one word at a time, increases the count for each word found, returns the final count



```
1 reference
public int UniqueWordCount()
{
    int count = 0;
    Link temp = list;
    while (temp != null)
    {
        count++;
        temp = temp.Next;
    }
    return count;
}
```

DisplaySortedWords Method

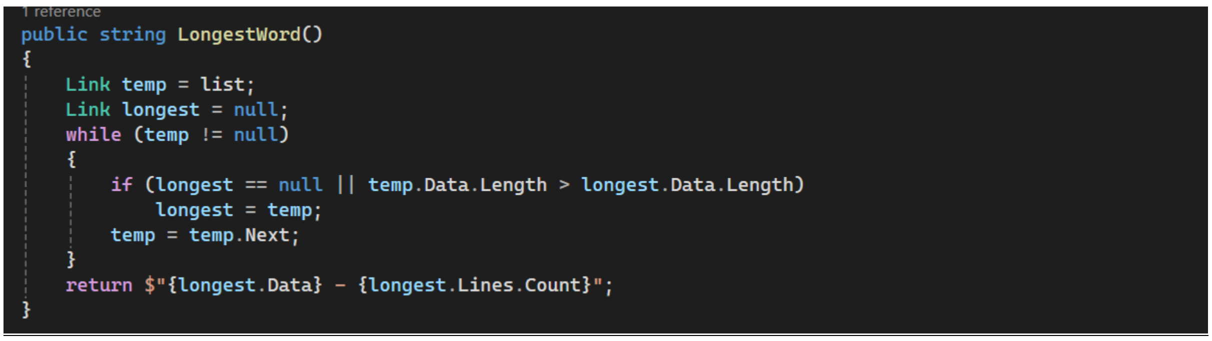
This method is taking all the unique words stored in the linked list and displays them in descending (Z-A) alphabetical order, along with how many different lines each word appeared on.



```
1 reference
public string DisplaySortedWords(bool ascending)
{
    List<Link> words = new List<Link>();      //list to hold all the Link nodes
    Link temp = list;
    while (temp != null)
    {
        words.Add(temp);
        temp = temp.Next;
    }

    // Always sorting in descending order
    var sorted = words.OrderByDescending(w => w.Data);

    string output = "";
    foreach (var word in sorted)
    {
        output += $"{word.Data} - {word.Lines.Count}\n";
    }

    return output;
}
```

Started with creating a new List<Link> called words. This is a temporary list that will be used to hold all the nodes from our linked list. Then a temp pointer to traverse the linked list from the head. This loop goes through each node in the linked list, each node (temp) is added to the words list and then we move to the next node using temp = temp.Next. This step is needed because we can't sort a LinkedList directly so we copy the data into a structure that can be sorted. Then we sort the words list in descending alphabetical order, the "OrderByDescending" method arranges the list based on the word itself (w.Data), from Z to A.

LongestWord Method: This method is used to find the word that has the maximum number of characters (the longest word) from the linked list of words. Program walks through the entire linked list, keep track of the word with the most characters, and finally return it along with the number of lines it appeared on.

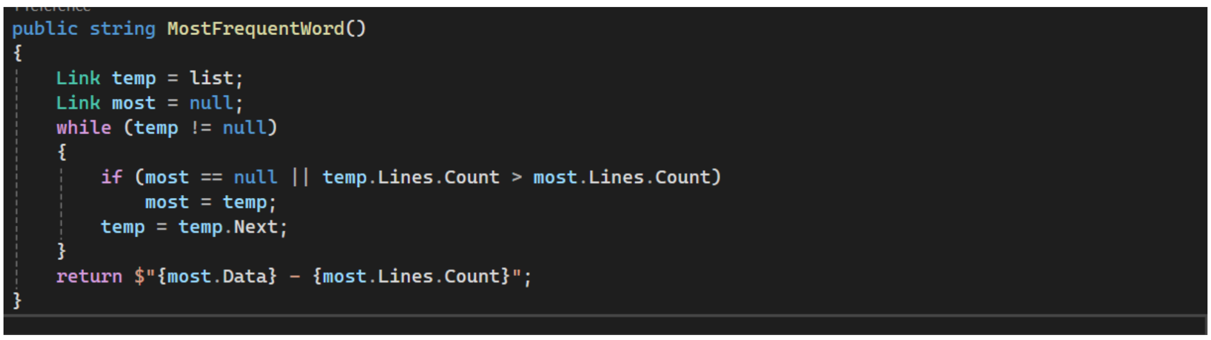


```
1 reference
public string LongestWord()
{
    Link temp = list;
    Link longest = null;
    while (temp != null)
    {
        if (longest == null || temp.Data.Length > longest.Data.Length)
            longest = temp;
        temp = temp.Next;
    }
    return $"{longest.Data} - {longest.Lines.Count}";
}
```

Inside the loop, we compare lengths of the word, for every word, If we have not found any word yet, we set longest = temp, Otherwise, we compare the length of the current word (temp.Data.Length) with the current longest word (longest.Data.Length).

If the current word is longer, we update longest to point to this new word. After checking all words, we return a string showing, the longest word (longest.Data), how many lines that word appeared on (longest.Lines.Count).

MostFrequentWord Method: The MostFrequentWord() method is used to find the word that appears the most number of times list.

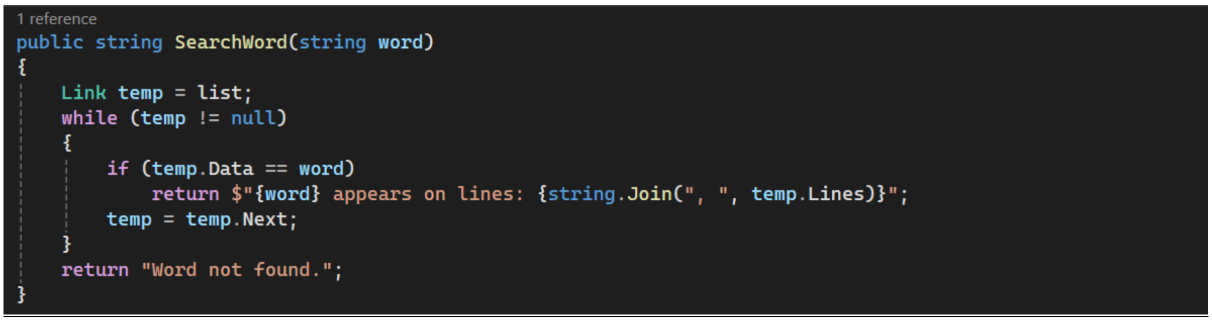


```
public string MostFrequentWord()
{
    Link temp = list;
    Link most = null;
    while (temp != null)
    {
        if (most == null || temp.Lines.Count > most.Lines.Count)
            most = temp;
        temp = temp.Next;
    }
    return $"{most.Data} - {most.Lines.Count}";
}
```

The logic here goes same as the longest word method, the only difference is the program compares temp.Lines.Count i.e. how many lines the word appeared on. And longest word checks temp.Data.Length i.e. length of the word.

Searchword Method:

This method searches the linked list to find if a particular word exists. If it finds the word, it tells you on which lines the word appears. If it doesn't find the word, it says "Word not found.".

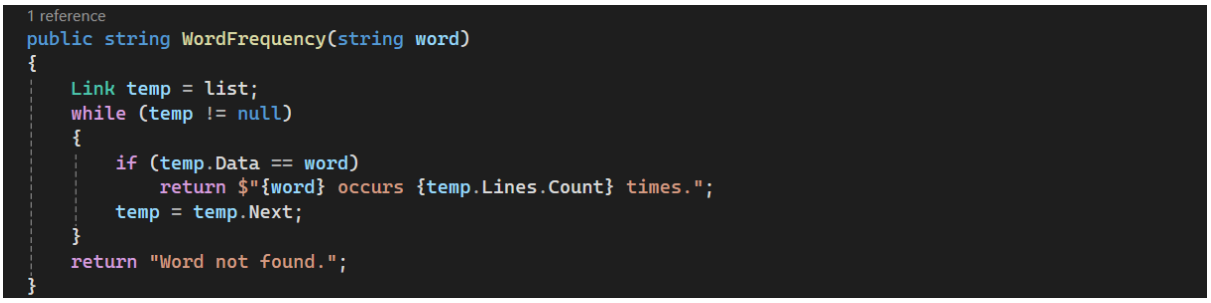


```
1 reference
public string SearchWord(string word)
{
    Link temp = list;
    while (temp != null)
    {
        if (temp.Data == word)
            return $"{word} appears on lines: {string.Join(", ", temp.Lines)}";
        temp = temp.Next;
    }
    return "Word not found.";
}
```

String.Join(", ", temp.Lines) joins all the line numbers into a comma-separated list (like 2, 5, 8). Then If the word didn't match, we move to the next node in the list.

WordFrequency Method:

The WordFrequency method finds out how many times a particular word appeared. It goes through the linked list one by one, checks each word, and if it finds the matching word, it returns the number of times it occurred,If it doesn't find the word, it returns "Word not found."

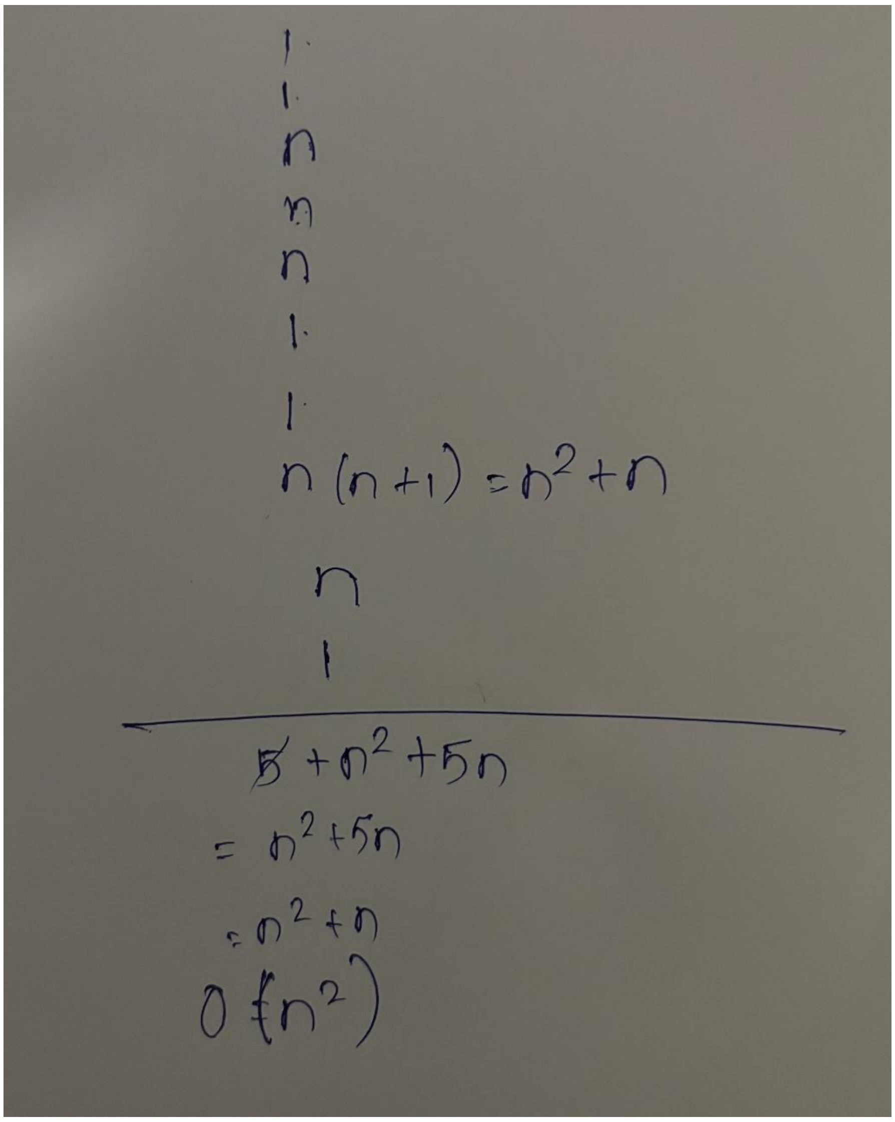


```
1 reference
public string WordFrequency(string word)
{
    Link temp = list;
    while (temp != null)
    {
        if (temp.Data == word)
            return $"{word} occurs {temp.Lines.Count} times.";
        temp = temp.Next;
    }
    return "Word not found.";
}
```

Algorithmic Complexity of solution & line-by-line breakdown of Complexity

```
public string DisplaySortedWords(bool ascending)

{

  List<Link> words = new List<Link>();

  Link temp = list;

  while (temp != null)

  {

    words.Add(temp);

    temp = temp.Next;

  }

  List<Link> sorted = words.OrderByDescending(w => w.Data).ToList();


  string output = "";

  for (int i = 0; i < sorted.Count; i++)

  {

    output += $"{sorted[i].Data} - {sorted[i].Lines.Count}\n";

  }
```

```
    return output;

}
```



1

1

n

n

n

1

1

$n(n+1) = n^2 + n$

n

1

$\cancel{5} + n^2 + 5n$

$= n^2 + 5n$

$= n^2 + n$

$O(n^2)$

Comparing the possible Data Structures

Static data structures, such as arrays, are efficient when retrieving items by index and convenient to use when the quantity is constant. However, for this text analysis application, they were unsuitable since the quantity of distinct words could not be anticipated beforehand. Arrays are of a specific size, and inserting or deleting items is time-consuming as other items must be moved.

Dictionaries are very fast for looking up and updating word counts, using keys to find values quickly. They work well to check word frequencies. But they don’t keep the words in order, which became a problem when sorting the output is needed.

Linked List was selected for this project because it provided an efficient storage system to maintain both word inventory and their respective line number records. The process of search and sort operations takes longer because we need to check every node sequentially, but the implementation remains straightforward to follow. The customizable word data management capabilities aligned well with the requirements of this project where custom-level word handling was necessary.

Binary Search Trees is considered to be a better approach since they are the best to store data in order and execute rapid searching when they are balanced. They were a bit harder to implement correctly, though, the implementation of this data structure proved challenging and they could run slowly especially when not properly managed.

Conclusion

The text analysis tool helps to show how Linked lists can effectively be used to control dynamic structured word data. Though not the fastest for search-and-sort operations, linked list proved to be of the right combination of simplicity and flexibility to implement functionality such as frequency tracking, line number mapping and word sorting. On the whole, this project gave a good insight into the data structure usage in non-theoretical real-world text processing settings.