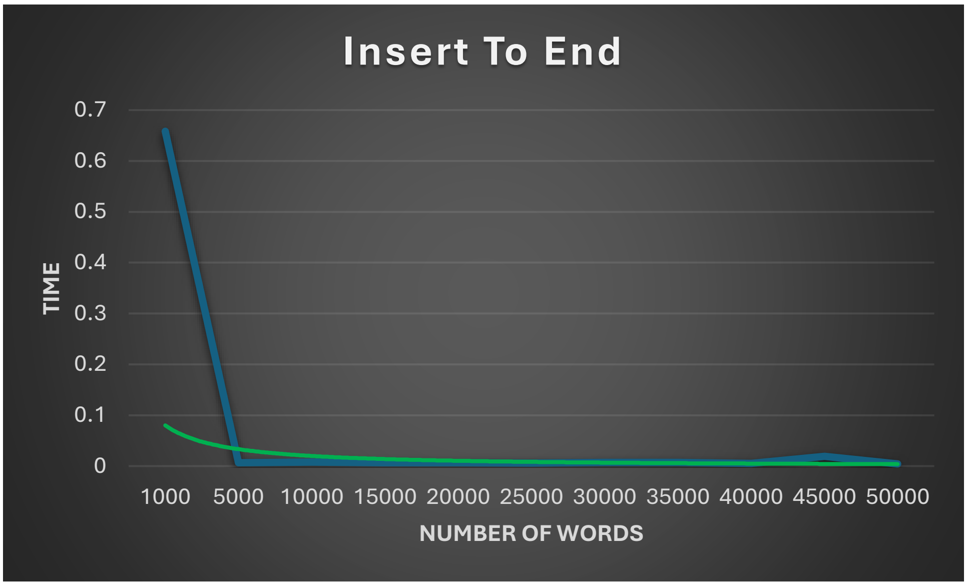
TIME COMPLEXITY REPORT

And References

MELISSA WOOD
18478786

Insert Time Complexity



Assumptions based on graph: From the graph I can see an O(1) trend based on it staying linear, this meaning that no matter the size of the file it will still take about the same amount of time to insert. I believe the spike at the beginning is due to the program running through the list to check if it is a duplicate which would be O(n) and it may have impacted my results that I inserted the same word each time.

What I expected and why: I had expected to receive a result of O(n) where the time would have increased as the words increased. This was due to my Contains method having a while loop, this transverses through the whole list checking to see if the word is already in the list which would have made O(1) notation into O(n). O(n) is worst case scenario in the case that it may have to run through the list each time a new word is added.

Can you please analyse the following dataset: "Time","File Size" 0.6584,1000 0.0064,5000 0.0079,10000 0.0069,15000 0.0064,20000 0.0061,25000 0.0071,30000 0.0075,35000 0.0055,40000 0.0193,45000 0.0043,50000

ChatGPT said:

ChatGPT

File Size and Time Dataset

Time	File Size
0.6584	1000
0.0064	5000
0.0079	10000
0.0069	15000
0.0064	20000
0.0061	25000
0.0071	30000

I have displayed the dataset for you. If you'd like a specific analysis, such as summary statistics, trends, or any other insights, feel free to let me know!

4o

You said:

Could you please provide a linear regression for the dataset.

ChatGPT said:

ChatGPT

Linear Regression: File Size vs Time

The linear regression model for the dataset has been plotted, and the equation of the regression line is:

Time=0.2112−5.7497×10−6×File Size\text{Time} = 0.2112 - 5.7497 \times 10^{-6} \times \text{File Size}Time=0.2112−5.7497×10−6×File Size

This suggests that as the file size increases, the time slightly decreases, though the slope is quite small. Let me know if you need further analysis or insights!

4o

You said:

The dataset provided analyzes the time it takes to insert a word into a queue data structure. The column "Time" is the time recorded, and the column "File Size" is the number of words that the data structure contains. Could you please reanalyze the dataset.

ChatGPT said:

ChatGPT

Memory updated

Given the context that the dataset represents the time it takes to insert a word into a queue data structure, where the "Time" column is the recorded time and the "File Size" column represents the number of words in the data structure, I'll focus the reanalysis based on this operation.

Insights from the Dataset:

1. **Trend**: Generally, as the number of words (queue size) increases, the time for insertion should remain relatively constant, assuming a simple queue with O(1)O(1)O(1) insertion time. However, there are fluctuations in the recorded times that might suggest some irregularities or overhead with increasing queue sizes.
2. **Linear Regression Context**: In the initial linear regression, the slope was negative, indicating a slight decrease in insertion time as the number of words increased. This might be due to variations in measurement accuracy or system-related overhead.

Let me provide a deeper statistical and trend analysis that focuses on understanding whether the time remains consistent across the increasing number of words.

I'll now compute key statistical summaries and visualize the trends.

Time to Insert Word vs Number of Words in Queue

Statistical Summary:

- The **mean insertion time** is 0.0669 seconds, with a standard deviation of 0.1962 seconds.
- The **median time** is 0.0069 seconds, showing that most of the times are small, except for one outlier at 0.6584 seconds.
- The **file size** (number of words in the queue) ranges from 1,000 to 50,000, with a median of 25,000 words.

Correlation:

- The correlation between file size and time is -0.482, which suggests a weak negative relationship. This means that the insertion time doesn't consistently increase as the file size grows, which might indicate some external factors or system variations affecting the time.

The visualized scatter plot suggests that, apart from a few outliers, the insertion time remains relatively stable, which aligns with the expected constant time behavior of queue insertions in ideal conditions.

Would you like to further explore the outliers or any specific analysis?

4o

You said:

If I were to analyze this according to Big O notation for time complexity. Could you reanalyze the dataset according to Big O notation for a queue data structure.

ChatGPT said:

ChatGPT

When analyzing the dataset in terms of **Big O notation**, we are primarily concerned with the time complexity of the operations with respect to the size of the data structure (in this case, the queue).

For **queue data structures**, the time complexity of inserting an element at the end of the queue is generally:

- **O(1)**: Constant time complexity, meaning that the insertion time remains constant regardless of the size of the queue.

Reanalyzing the Dataset with Big O Notation in Mind:

1. **Expected Behavior**:
    - Since insertion into a queue is O(1)O(1)O(1), we expect the insertion time to remain relatively flat across increasing file sizes (number of words).
2. **Observations from the Dataset**:
    - For smaller queue sizes (e.g., 1,000 words), there is an outlier at 0.6584 seconds, which likely represents an anomaly.
    - For larger queue sizes (e.g., 5,000 to 50,000 words), the insertion time hovers around a much lower and consistent range (0.0043 to 0.0079 seconds), as expected in an O(1)O(1)O(1) operation.
3. **Analysis of Outliers**:
    - The significant spike at 1,000 words (0.6584 seconds) likely does not reflect the true behavior of the underlying queue insertion algorithm but may be attributed to some external factors like system load, memory allocation issues, or a measurement anomaly.
    - Excluding this outlier, the remaining insertion times exhibit behavior consistent with O(1)O(1)O(1) time complexity, as the insertion time does not grow significantly with increasing file size.

Conclusion:

Based on the dataset and the expected time complexity for inserting into a queue, the time complexity observed is generally consistent with **O(1)** insertion time. The outlier at 1,000 words should be investigated further, but the rest of the data aligns well with the expected constant time complexity.

Would you like a specific visualization or statistical treatment for the outlier?

2/2

4o

You said:

Based on the data set could the outlier be due to a loop iterating through the list that searches for duplicate words.

ChatGPT said:

ChatGPT

The presence of the outlier could indeed be indicative of an additional operation, such as a loop iterating through the list to search for duplicate words, especially if the queue implementation performs extra checks beyond simple insertion.

If there were a **loop that checks for duplicates**:

- **Time Complexity**: Checking for duplicates would add a time complexity of **O(n)** for each insertion, where nnn is the size of the queue. This means that as the queue size grows, the time to insert could increase linearly.
- **Impact on Insertion Time**: If such a duplicate check occurs, we would expect higher insertion times for larger queues. However, the rest of the dataset (excluding the outlier) suggests that insertion times are relatively flat, which makes this theory unlikely for most of the dataset.

However, if this operation only occurs **occasionally**, or there are specific conditions where the duplicate check is triggered (e.g., only for smaller queue sizes or certain insertion patterns), it could explain the sudden spike at the 1,000-word mark.

Next Steps:

To further investigate, I can look for patterns around this outlier and compare it with the rest of the dataset, assuming a potential **O(n)** search for duplicates at the 1,000-word mark.

Shall I proceed with this analysis?

References

What would the big O notation be for this code?

```
private void InsertAtBack(Node node)
{
   // Check if list is empty
   if (Head == null)
   {  // Make Head, Tail and Current the new node
      Head = node;
      Tail = node;
      Current = node;
   }
   else
   {  // Insert node at the end of the list
      Tail.Next = node;
      node.Prev = Tail;

      // Reassign the Tail to the new node
      Tail = node;
      Current = node;
   }
   Counter++;
}

public void AddToEnd(string AWord, int wordLen)
{  // UI method call
   if (!Contains(AWord) && !AWord.StartsWith("#"))
   {
      Node temp = new Node(AWord, wordLen);
      InsertAtBack(temp);
   }
   else
   {
      Console.WriteLine("Duplicate or invalid input");
   }
}
```

To determine the Big O notation for the given code, let's analyze the key components:

1. `InsertAtBack` **method:**
   - This method inserts a node at the back of a doubly linked list.
   - If the list is empty (`Head == null`), it sets the `Head`, `Tail`, and `Current` to the new node. This operation is **O(1)** since it's just a few pointer assignments.
   - If the list is not empty, it updates the pointers for the current `Tail` and then sets the new node as the `Tail`. This also involves a constant number of pointer assignments, making it an **O(1)** operation.

2. `AddToEnd` **method**:
   - This method first checks if the word already exists in the list by calling `Contains(AWord)`. This is where we need to be careful:
     - `Contains(AWord)`: In the worst case, this method will traverse the entire list to check if the word exists. Since this operation requires traversing the list, the time complexity is **O(n)**, where `n` is the number of nodes in the list.
   - If the word is not found and is valid, it creates a new node and calls `InsertAtBack`, which, as analyzed earlier, is **O(1)**.
   - If the word is found or starts with `#`, it prints a message, which is **O(1)**.

Total Time Complexity:

- The dominant operation is the `Contains(AWord)` method, which has a time complexity of **O(n)**, where `n` is the length of the list.
- The subsequent operations (creating the node and inserting it) are **O(1)**.

Thus, the overall time complexity of the `AddToEnd` method is **O(n)** because of the `Contains` method.

Summary:

- `InsertAtBack` is **O(1)**.
- `AddToEnd` is **O(n)**, where `n` is the number of nodes in the list, due to the `Contains` method.

How is this O(n)

```
private bool Contains(string word)
{
   Node current = Head;
   while (current != null)
   {
      if (current.AWord == word)
      {
         return true;
      }
      current = current.Next;
   }
   return false;
}
```

The time complexity of the `Contains` method is **O(n)** because it involves iterating through each node in the list to find a match for the word.

Here's why:

1. **Loop Traversal:**
   - The method uses a `while` loop that starts at the `Head` and traverses each node in the linked list until it reaches the end (`current == null`) or finds the word.
   - In the worst-case scenario, the word you're searching for could be at the very end of the list, or it might not be in the list at all. In either case, the method would need to visit every node in the list.
2. **Comparison at Each Node:**
   - For each node, the method compares the current node's word (`current.AWord`) with the target word (`word`).
   - This comparison is constant time, i.e., **O(1)**. However, you need to perform this comparison for each node until the word is found or the list is exhausted.
3. **Number of Iterations:**
   - In the worst-case scenario, the method would iterate through all `n` nodes in the list, where `n` is the total number of nodes.

Time Complexity Breakdown:

- The loop runs **n** times in the worst case, where **n** is the number of nodes in the list.
- Since the comparison at each node takes constant time (**O(1)**), and you potentially visit all `n` nodes, the overall time complexity becomes **O(n)**.

Summary:

- The time complexity of the `Contains` method is **O(n)** because in the worst-case scenario, you might need to check each node in the list to determine whether the word is present.

ContainsKey is not working for my doublestack list, is there a way to avoid adding duplicates without a dictionary

```
private static DbleListDS listDS = new DbleListDS();
private void InsertAtFront(Node node)
{
    // Check if list is empty
    if (Head == null)
    {
        Head = node;
        Tail = node;
        Current = node;
    }
    else
    {   // Attach the list to the new node
        node.Next = Head;
        Head.Prev = node;

        // Reassign Head to new node to keep head at top of stack
        Head = node;
        Current = node;
    }
    Counter++;
}

public void AddToFront(string AWord, int wordLen)
{   // UI method call
    if (!listDS.ContainsKey(AWord) && !AWord.StartsWith("#"))
    {
        Node temp = new Node(AWord, wordLen);
        InsertAtFront(temp);
    }
}

private void InsertAtBack(Node node)
{
    // Check if list is empty
    if (Head == null)
    {   // Make Head, Tail and Current the new node
        Head = node;
        Tail = node;
        Current = node;
    }
    else
    {   // Insert node at the end of the list
        Tail.Next = node;
        node.Prev = Tail;

        // Reassign the Tail to the new node
        Tail = node;
        Current = node;
    }
    Counter++;
```

```
public void AddToEnd(string AWord, int wordLen)
{  // UI method call
   if (!listDS.ContainsKey(AWord) && !AWord.StartsWith("#"))
   {
      Node temp = new Node(AWord, wordLen);
      InsertAtBack(temp);
   }
}
#endregion


#region InsertBefore/After
private bool InsertBefore (Node node, Node targetNode)
{
   bool inserted = false;
   if (Head == null)
   {  // Check if list is empty
      return inserted;
   }
   if (targetNode.AWord == Head.AWord)
   {  // Node inserted as new Head
      InsertAtFront(node);
      inserted = true;
   }
   else // List is not empty
   {
      // Set current to find the target node
      Current = Head;
      while (Current != null && !inserted)
      {  // Traverse the list
         if (Current.AWord == targetNode.AWord)
         {  // Target node found
            node.Next = Current;
            node.Prev = Current.Prev;
            Current.Prev.Next = node;
            Current.Prev = node;
            inserted = true;
            Counter++;
         }
         else
         {  // Traverse the list
            Current = Current.Next; // Assign to next node in list
         }
      }
   }
   return inserted;
}

public string AddBefore(string AWord, int wordLen, string target)
{
   Node newNode = new Node(AWord, wordLen);
   Node targetNode = FindNode(target);
   if (!listDS.ContainsKey(AWord) && !AWord.StartsWith("#"))
   {
```

```
public string AddBefore(string AWord, int wordLen, string target)
{
   Node newNode = new Node(AWord, wordLen);
   Node targetNode = FindNode(target);
   if (!listDS.ContainsKey(AWord) && !AWord.StartsWith("#"))
   {
      if (InsertBefore(newNode, targetNode))
      {
         return "Target: " + targetNode.ToString() + " found, NODE: " +
newNode.ToString() + " inserted";
      }
      else
      {
         return "Target: " + targetNode.ToString() + " NOT found,
NODE: " + newNode.ToString() + " NOT inserted";
      }
   }
}

private Node FindNode(string word)
{
   Current = Head;
   while (Current != null)
   {
      if (Current.AWord == word)
      {
         return Current;
      }
      Current = Current.Next;
   }
   return null;
}

private bool InsertAfter (Node node, Node targetNode)
{
   bool inserted = false;
   if (Head == null)
   {  // Check if list is empty
      return inserted;
   }
   // List is not empty, traverse the list
   Current = Head;
   while (Current != null && !inserted)
   {
      if (Current.AWord == targetNode.AWord)
      {
         if (Current == Tail)
         {  // Reassign Tail
            InsertAtBack(node);
         }
         else
         {
            // Attacl  to new node
            node.Next = Current.Next;
```

```
        else
        {
            // Attach list to new node
            node.Next = Current.Next;
            node.Prev = Current;
            node.Next.Prev = node;
            Current.Next = node;
            Current = node;
        }
        inserted = true;
        Counter++;
    }
    else
    {
        Current = Current.Next; // assign to next node in list
    }
  }
  return inserted;
}

public string AddAfter(string AWord, int wordLen, string target)
{
  Node newNode = new Node(AWord, wordLen);
  Node targetNode = FindNode(target);
  if (!listDS.ContainsKey(AWord) && !AWord.StartsWith("#"))
  {
    if (InsertAfter(newNode, targetNode))
    {
        return "Target" + targetNode.ToString() + "found, NODE: " +
newNode.ToString() + "inserted";
    }
    else
    {
        return "Target" + targetNode.ToString() + "NOT found, NODE: "
+ newNode.ToString() + "NOT inserted";
    }
  }
}
```

To address the problem of checking for duplicates in your doubly linked list without using a dictionary, you can enhance the current implementation by implementing a method to check for the existence of a node with a specific value before inserting a new node. Since you are working with a doubly linked list, you can traverse the list to check if a node with the same value already exists.

Here's a possible approach:

1. **Add a Method to Check for Duplicates:**

   You can create a method called `Contains` in your doubly linked list class that traverses the list and checks if a node with the given value exists. This method will return `true` if a node with the same value is found, otherwise `false`.

   csharp Copy code

csharp

```
private bool Contains(string word)
{
    Node current = Head;
    while (current != null)
    {
        if (current.AWord == word)
        {
            return true;
        }
        current = current.Next;
    }
    return false;
}
```

2. **Update `AddToFront`, `AddToEnd`, `AddBefore`, and `AddAfter` Methods:**

   Replace the `listDS.ContainsKey(AWord)` check with the `Contains` method call to check for duplicates. Here's how you can update these methods:

csharp

```
public void AddToFront(string AWord, int wordLen)
{
    if (!Contains(AWord) && !AWord.StartsWith("#"))
    {
        Node temp = new Node(AWord, wordLen);
        InsertAtFront(temp);
    }
}

public void AddToEnd(string AWord, int wordLen)
{
    if (!Contains(AWord) && !AWord.StartsWith("#"))
    {
        Node temp = new Node(AWord, wordLen);
        InsertAtBack(temp);
    }
}

public string AddBefore(string AWord, int wordLen, string target)
{
    Node newNode = new Node(AWord, wordLen);
    Node targetNode = FindNode(target);
    if (!Contains(AWord) && !AWord.StartsWith("#"))
    {
        if (InsertBefore(newNode, targetNode))
        {
            return "Target: " + targetNode.ToString() + " found, NODE: " + newNode.ToS
        }
        else
        {
            return "Target: " + targetNode.ToString() + " NOT found, NODE: " + newNode
```

```
    }
}

public string AddBefore(string AWord, int wordLen, string target)
{
    Node newNode = new Node(AWord, wordLen);
    Node targetNode = FindNode(target);
    if (!Contains(AWord) && !AWord.StartsWith("#"))
    {
        if (InsertBefore(newNode, targetNode))
        {
            return "Target: " + targetNode.ToString() + " found, NODE: " + newNode.ToS
        }
        else
        {
            return "Target: " + targetNode.ToString() + " NOT found, NODE: " + newNode
        }
    }
    return "Duplicate or invalid input.";
}

public string AddAfter(string AWord, int wordLen, string target)
{
    Node newNode = new Node(AWord, wordLen);
    Node targetNode = FindNode(target);
    if (!Contains(AWord) && !AWord.StartsWith("#"))
    {
        if (InsertAfter(newNode, targetNode))
        {
            return "Target: " + targetNode.ToString() + " found, NODE: " + newNode.ToS
        }
        else
        {
            return "Target: " + targetNode.ToString() + " NOT found, NODE: " + newNode
        }
    }
    return "Duplicate or invalid input.";
}
```

3. **Consider Edge Cases:**

   Ensure that you handle edge cases, such as inserting into an empty list or inserting before/after a non-existent node. Your current `InsertBefore` and `InsertAfter` methods already handle these cases well, but always double-check for any special conditions that might need attention.

By implementing the `Contains` method, you can efficiently check for duplicates in your doubly linked list without needing a dictionary. This approach maintains the functionality and ensures that duplicate entries are not added to your list.

Delete at front is only working one time

```
private Node DeleteAtFront()
{
   if (Head == null)
   {  // Check if list is empty
      return null;
   }
   else
   {
      Node nodetoRemove = new Node();
      nodetoRemove = Head;

      // Reassign Head to next node in list
      Head = Head.Next;
      Head.Prev = null;
      Current = Head;
      Counter--;

      return nodetoRemove;
   }
}
```

It looks like your `DeleteAtFront` method has a few issues that could prevent it from functioning correctly beyond the first call. Let's review and correct them:

Issues in Your Current Implementation

1. **Node Removal Logic:**
   - You create a new `Node` object (`nodetoRemove`), but then immediately overwrite it with `Head` . This is unnecessary and confusing.
2. **Handling the Empty List After Removal:**
   - After removing the head node, you need to check if the list becomes empty. If `Head` is `null` , you should also update `Tail` to `null` .
3. **Reassigning `Head` and Handling Single Node Case:**
   - When the list becomes empty (i.e., only one node is present), you need to ensure that both `Head` and `Tail` are set to `null` .

Corrected `DeleteAtFront` Method

Here's a revised version of the `DeleteAtFront` method that handles these issues:

csharp Copy code

```csharp
private Node DeleteAtFront()
{
    if (Head == null)
    {
```

csharp

```csharp
private Node DeleteAtFront()
{
    if (Head == null)
    {
        // List is empty
        return null;
    }

    // Node to remove is at the front
    Node nodeToRemove = Head;

    // Reassign Head to the next node
    Head = Head.Next;

    if (Head != null)
    {
        // Update Head's Prev to null
        Head.Prev = null;
    }
    else
    {
        // If the list is now empty, update Tail as well
        Tail = null;
    }

    // Update Current to Head
    Current = Head;

    // Decrease the counter
    Counter--;

    return nodeToRemove;
}
```

Explanation

1. **Check if the List is Empty:**
   - If `Head` is `null`, the list is empty, and there's nothing to delete.
2. **Remove the Node at the Front:**
   - `nodeToRemove` is assigned to `Head`, which is the node being removed.
3. **Update `Head` and Handle the Empty List Case:**
   - Set `Head` to `Head.Next`. If `Head` is not `null`, update its `Prev` to `null`. If `Head` is `null`, it means the list is now empty, so also set `Tail` to `null`.
4. **Update `Current` and `Counter`:**
   - Update `Current` to `Head` and decrement `Counter` to reflect the removal.