CAB301 - Assessment 1

Braydan Newman - n11272031

Contents

1 Introduction

The provided algorithm is a method called Add, which adds a new member to a collection (members) while maintaining the collection in sorted a alphabetical order.

1.1 Preconditions

- The member collection is not full.
- The member is valid and is not null.

1.2 Postconditions

- The new member is added to member collection and the members in the collection remain in sorted, alphabetical, order by their full name.
- No duplicate can be added into the member collection.

2 Algorithm Analysis

2.1 Description

The Add method implements a Binary Search algorithm for finding the point at which to add the new member. This is an effective technique when working sorted arrays. This works by dividing the search in half on every iteration.

For this implementation, once the binary search has found the position for the new member, this is to keep alphabetical order of all members. All the members after the position is moved once space to the right to make room for the new member in the correct position without overwriting any data.

2.2 Flow

1. If the collection is full, return without adding member.
2. If the collection is empty, the new member is inserted at index 0, then return.
3. Perform a binary search to find the correct insertion position for the new member to keep alphabetical order.-
4. Determine insertion index with binary search.
   (a) Compare middle member in collection to new member
   (b) If new member is equal to middle member return without adding. (Dont add duplicate)
   (c) If new member comes before middle member, set middle as new upper bound
   (d) If new member comes after middle member, set middle as new lower bound
   (e) Repeat till lower and upper bound are equal.
5. Shift elements from insertion index to end right one posistion in collection.
6. Insert new member at insertion index

Algorithm 1 Add(member)

```
if NOT IsFull() then 1
    if count = 0 then                                        ▷ If collection empty add member at start
        $members[count] \leftarrow member$
        $count \leftarrow count + 1$
        return
    end if

    $low \leftarrow 0$
    $high \leftarrow count - 1$
    while $low \leq high$ do                                 ▷ Binary search for insertion index
        $mid \leftarrow low + \frac{(high-low)}{2}$
        if $member = members[mid]$ then
            return
        else if $member < members[mid]$ then
            $high \leftarrow mid - 1$
        else if $member > members[mid]$ then
            $low \leftarrow mid + 1$
        end if
    end while

    for $i \leftarrow count$ down to $low + 1$ do            ▷ Shift members in collection over
        $members[i + 1] \leftarrow members[i]$
    end for

    $members[low] \leftarrow member$                         ▷ Add member to collection
    $count \leftarrow count + 1$
end if
```

2.3 Complexity Analysis

2.3.1 Time Complexity

Time complexity Parts:

1. Binary Search:
    - Logarithmic Complexity
    - String Compare Method Complexity assumed constant time ($O(1)$)
    - $C_{worst}(n) = \sum_{k=1}^{n} 3 \log k + 1 = 3 \sum_{k=1}^{n} \log k + 1 = \log_2(n + 1)$
    - $C_{worst}(n) = \log_2(n + 1) \in O(\log n)$
2. Shifting elements:
    - Linear Complexity
    - $C_{worst}(n) = \sum_{i=1}^{n} 1 = n \in O(n)$

Overall Time complexity:

- The dominant part of this algorithm is shifting of the elements which takes $O(n)$ while the search only takes $O(\log(n)$. Due to the shifting having a higher complexity it is dominate and takes precedence
- $O(\log(n) + n) = O(n)$

2.3.2 Space Complexity

- None of the algorithms change the amount of memory used and it is all done in place; thus, this algorithm has a space complexity of $O(1)$, constant space.
- $O(1)$

2.4 Efficiency

- The algorithm Efficiency is decent with the search being $O(log(n))$ Time Complexity but due to shiffting being $O(n)$ Time Complexity the overall complexity turn out to be $O(n)$.
- The only way a binary search works is with a sorted list, that is why it is necessary for this approach to have a constantly sorted collection of members.

2.5 Potential Improvements

- Using a data structure like a binary tree would make adding to the collection $n(log(n))$, as there is no need for shifting.