Programming Assignment 1

Searching in 1-D Dynamic Data Structures

Data Set:

The dataset used for this assignment is from the Open English Word List (EOWL), available at EOWL GitHub Repository. This dataset contains a large collection of English words in plain text format. The words will be stored in a dynamic array (or list) that grows as more words are inserted.

Dynamic array growth strategies:

The three growth strategies implemented are:

a. **Incremental Strategy (A)**: The size of the array increases by 10 each time it reaches full capacity.
   E.g., if the current capacity is 20, it grows to 30 when full.
b. **Doubling Strategy (B)**: The size of the array doubles when it is full.
   E.g., if the current capacity is 16, it grows to 32 when full.
c. **Fibonacci Strategy (C)**: The array size increases by the next number in the Fibonacci sequence when full.
   E.g., if the current size is 13, it increases to 21 when full (because 21 is the next Fibonacci number).

Binary Search for sorted insertions:

Each word is inserted into the array in a sorted order. A **binary search** is used to determine the correct position to insert the word. Binary search has a time complexity of O(log n), where n is the number of words in the array. Once the insertion point is identified, the word is inserted by shifting the appropriate elements to the right to make space.

Pseudo Code

1. **DynamicArray Class**:
   - **Attributes**:
     - array: Starts with size 2.
     - capacity: Tracks how big the array is.
     - size: Tracks how many elements are in the array.
     - growth_strategy: Defines how the array grows (incremental, doubling, Fibonacci).

2. **append(value)**:
    - If the array is full, call _resize() to grow the array.
    - Insert the value into the correct sorted position using binary search.
3. **_resize()**:
    - If the growth strategy is **incremental**, increase size by 10.
    - If it's **doubling**, double the size.
    - If it's **Fibonacci**, increase by the next Fibonacci number.
    - Copy the elements to the new, bigger array.
4. **_binary_insert(value)**:
    - Find the correct spot for the new value using binary search.
    - Insert value and shift other elements if needed.
5. **print_status()**:
    - Print the size, capacity, and some key elements of the array.
6. **load_eowl_dataset(file_path)**:
    - Load words from a file and return them as a list.
7. **measure_performance(array, words)**:
    - For each word, insert it into the array.
    - Print the number of insertions, time taken, and array status after each growth.
8. **measure_python_list_performance(words)**:
    - Similar to above, but using Python’s built-in list.
9. **Main Steps**:
    - Load the word list.
    - Test the dynamic array using different growth strategies (incremental, doubling, Fibonacci).
    - Compare with Python's built-in list.

Time complexity and space complexity of all three strategy

→ Time Complexity Analysis

Binary Search for Insertion

The append() method uses binary search to find the correct position to insert the new element, ensuring that the array remains sorted. The time complexity of binary search is:

- **Binary Search**: O(log n) ,where n is the current number of elements in the array. This is because binary search divides the array into halves and checks whether the element fits in the left or right half.

Shifting Elements for Insertion

Once the binary search determines the correct position for the new element, all elements to the right of that position must be shifted to the right by one position to make space for the new element. This takes O(n) time in the worst case, where the element needs to be inserted at the beginning of the array.

- **Shifting**: O(n), where n is the number of elements that need to be shifted.

Resizing the Array

The time complexity of resizing depends on the growth strategy chosen:

1. **Incremental Strategy**:
    - Resizing occurs when the array is full, and the size is increased by a constant amount (10).
    - **Cost per resizing**: O(n), where n is the current size of the array, as all elements need to be copied into a new, larger array.
    - **Amortized Time Complexity**: Resizing happens frequently since the array only grows by 10 each time, leading to frequent copying. Over time, the amortized time complexity for resizing is O(n) for each insertion.
2. **Doubling Strategy**:
    - The array size doubles when it is full.
    - **Cost per resizing**: O(n), as all elements need to be copied to the new, larger array.
    - **Amortized Time Complexity**: Resizing occurs less frequently because the array size doubles, reducing the number of resizing. The amortized time complexity for insertions is O(1), meaning that in the long run, the cost of resizing is spread out across many insertions.
3. **Fibonacci Strategy**:
    - The array size increases according to the Fibonacci sequence.

- **Cost per resizing**: O(n), as all elements need to be copied to a new array whose size is determined by the next Fibonacci number.
- **Amortized Time Complexity**: Fibonacci growth behaves similarly to doubling in that it reduces the number of resizing compared to the incremental strategy, but it doesn't grow as quickly as doubling. The amortized time complexity is closer to O(1) for large inputs, but resizing occur slightly more often than in the doubling strategy.

4. **Python's Built-in List**:
   - **Growth pattern**: Python's built-in list uses a **doubling-like strategy** to grow its internal storage.
   - **Performance**: Python's list is highly optimized. Since it uses a strategy similar to doubling, it performs very efficiently.
   - **Time complexity**: Python's list has **O(1)** amortized insertion time due to its efficient growth strategy.

Total Time Complexity for Insertions

For each strategy, the total time complexity for inserting a word involves:

- Binary search: O(log n)
- Shifting elements: O(n)
- Resizing (if needed): O(n)

Thus, the overall time complexity for **one insertion** can be summarized as:

1. **Incremental Strategy**: O(n + log n), with frequent resizing.
2. **Doubling Strategy**: Amortized O(log n), as resizing are less frequent.
3. **Fibonacci Strategy**: Amortized O(log n), but slightly more frequent resizing than doubling.

→ Space Complexity Due to Resizing

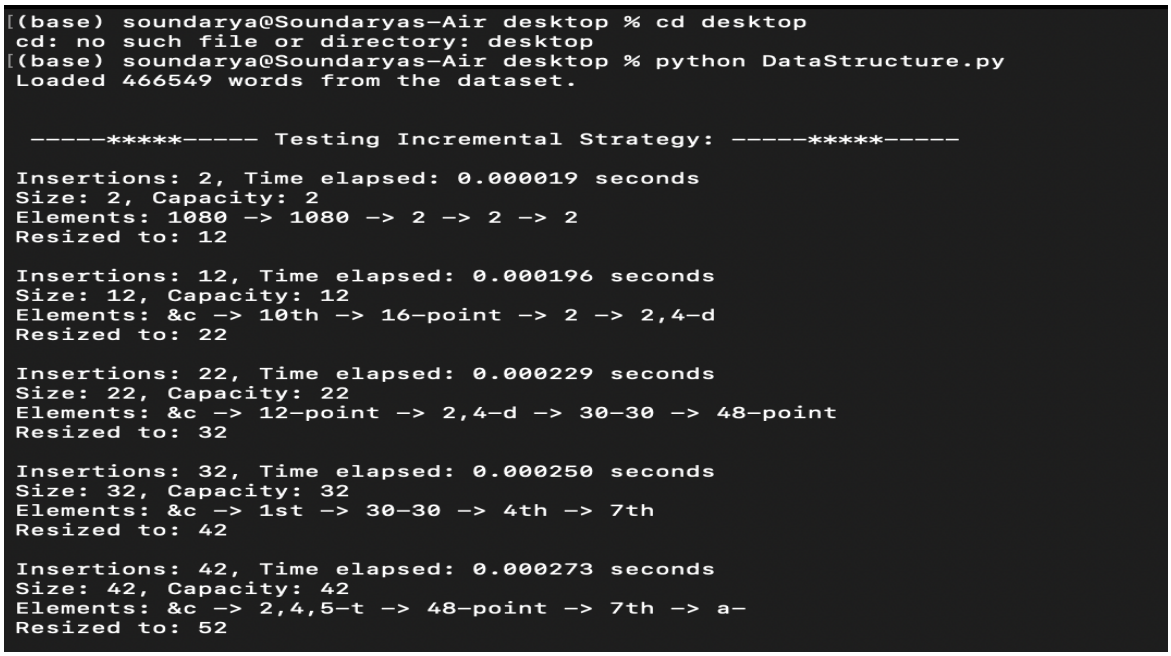
1. **Incremental Strategy**:
   - The array grows by 10 each time it is full. At any given moment, there could be up to 10 empty slots in the array, leading to an additional O(10) space overhead.
   - **Space complexity**: O(n+10) where n is the number of elements currently in the array.
2. **Doubling Strategy**:

- The array size doubles when it is full, leaving up to 50% of the array empty after each resize.
- **Space complexity**: O(n), where n is the number of elements. The space overhead is O(n) because at worst, half of the array is empty after a resize.

3. **Fibonacci Strategy**:
   - The array grows according to the Fibonacci sequence, which minimizes wasted space compared to doubling. The empty slots in the array are fewer compared to the doubling strategy.
   - **Space complexity**: O(n), with fewer empty slots than in the doubling strategy but more frequent resizing.
4. **Python's built-in lists**
   - Python's built-in lists use a **doubling-like strategy** for resizing. The space complexity is similar to the **Doubling Strategy** above.
   - **Space complexity**: O(n), with some extra unused space after each resizing.

Observations from Outputs:

Sample output screenshots where the number of elements inserted is set to 50.



```
-----**********----- Testing Doubling Strategy: -----**********-----

Insertions: 2, Time elapsed: 0.000002 seconds
Size: 2, Capacity: 2
Elements: 1080 -> 1080 -> 2 -> 2 -> 2
Resized to: 4

Insertions: 4, Time elapsed: 0.000016 seconds
Size: 4, Capacity: 4
Elements: &c -> 10-point -> 1080 -> 2 -> 2
Resized to: 8

Insertions: 8, Time elapsed: 0.000031 seconds
Size: 8, Capacity: 8
Elements: &c -> 1080 -> 11-point -> 16-point -> 2
Resized to: 16

Insertions: 16, Time elapsed: 0.000046 seconds
Size: 16, Capacity: 16
Elements: &c -> 11-point -> 1st -> 20-point -> 30-30
Resized to: 32

Insertions: 32, Time elapsed: 0.000071 seconds
Size: 32, Capacity: 32
Elements: &c -> 1st -> 30-30 -> 4th -> 7th
Resized to: 64
```

```
-----**********----- Testing Fibonacci Strategy: -----**********-----

Insertions: 2, Time elapsed: 0.000001 seconds
Size: 2, Capacity: 2
Elements: 1080 -> 1080 -> 2 -> 2 -> 2
Resized to: 3

Insertions: 3, Time elapsed: 0.000014 seconds
Size: 3, Capacity: 3
Elements: &c -> &c -> 1080 -> 2 -> 2
Resized to: 5

Insertions: 5, Time elapsed: 0.000028 seconds
Size: 5, Capacity: 5
Elements: &c -> 10-point -> 1080 -> 10th -> 2
Resized to: 10

Insertions: 10, Time elapsed: 0.000040 seconds
Size: 10, Capacity: 10
Elements: &c -> 1080 -> 12-point -> 18-point -> 2
Resized to: 65

-----**********----- Testing Python's Built-in List: -----**********-----

Insertions: 10, Time elapsed: 0.000004 seconds

Insertions: 20, Time elapsed: 0.000009 seconds

Insertions: 30, Time elapsed: 0.000014 seconds

Insertions: 40, Time elapsed: 0.000019 seconds

Insertions: 50, Time elapsed: 0.000023 seconds
(base) soundarya@Soundaryas-Air desktop %
```

- **Incremental Strategy:** This is the slowest strategy because it requires frequent resizing and shifting of elements. It also has higher space overhead due to smaller resizing steps. The frequent copying of elements during resizing increases the time complexity.
- **Doubling Strategy:** This is efficient in terms of both time and space. Doubling reduces the frequency of resizing, so the time complexity for insertions is amortized O(log n). However, it can leave a significant amount of unused space after resizing.
- **Fibonacci Strategy:** This strategy is a compromise between incremental and doubling. It resizes more frequently than doubling but uses space more efficiently. Insertion time is similar to doubling but with slightly higher overhead for resizing.
- **Python's Built-in List:** The built-in list in Python uses a doubling-like strategy, which provides good time complexity for insertions and relatively efficient space usage.

The resizing operations for each strategy indicate how the time complexities manifest. The Fibonacci strategy shows less predictable growth patterns, while doubling shows large gaps between resizes, reducing the frequency of costly operations. The incremental strategy provides a consistent growth pattern but can lead to more frequent expensive operations compared to the others.

- For applications where the size of the dataset is relatively predictable or grows linearly, the incremental strategy might be simpler and more cost-effective.
- For unpredictable growth or when large datasets are expected, the doubling strategy might provide a better balance between time spent on resizing and memory usage.
- The Fibonacci strategy might be suitable for scenarios where memory usage efficiency is more critical than resize overhead

References

- ChatGPT, Google, YouTube
- GeeksForGeeks
- Concepts thought in class and materials provided by Professor.