IFN564 Assignment

Weighting: 50%

Individual submission via QUT Canvas

|  |  |
| --- | --- |
| **Name** | Replace this with your name |
| **Student ID** | Replace this with your student ID |
| **Declaration of assistance by generative artificial intelligence**  **For this assignment, you were given a choice of two paths:**   1. **To provide a solution entirely on your own.** 2. **To provide a solution when assisted by a generative artificial intelligence.**   **Delete one tick to indicate the path you selected** | |
| **✓** | The traditional approach: I answered all questions on my own and did not seek help from any generative artificial intelligence. |
|  | The AI assisted approach where I did use generative AI. I have provided the exact prompts that I used, the responses that I obtained and have reflected on whether these responses are appropriate (and why). I understand that this is crucial to assess that I am meeting the learning outcomes of the unit. |
| **Instructions to submit your solution.**   1. Be sure to provide your name and student ID on the cover sheet, above. 2. Remove a tick from the rows above to declare the path you selected to solve the assignment. 3. Provide a response to every question, to maximise your potential final mark. 4. **Do not modify the question headings in this template**. Use the *Navigation pane* (View -> Show, Navigation pane) to ensure that each question is bookmarked. Refer to the original template on Canvas to check if you have made accidental changes to the question headings. 5. Upload this file as **.docx** to Canvas before the due date. | |

Please make sure to answer all fields on the cover page.

# Question 1

## 1.What is the problem size?

The problem size, denoted as 𝑛, is the number of customers currently stored in the array. The array itself has a maximum capacity of 𝑁max.

## 2.What is the basic operation?

1. **Insert:** Inserting a new customer if there is space in the array.
2. **Search:** Finding the customer by iterating through each element in the array from index 0 to the last index.
3. **Delete:** Comparing elements to find the customer and shifting elements after deletion.

## 3.Does the algorithm have a best case and a worst case?

1. **Insert**:
   1. **Best Case**: 𝑂(1). Inserting a new customer at the end of the array.
   2. **Worst Case**: 𝑂(1). The operation remains constant time as long as there is space in the array.
2. **Search**:
   1. **Best Case**: 𝑂(1). The customer is the first element in the array, so no further iteration is needed.
   2. **Worst Case**: 𝑂(𝑛). The customer is the last element or not present in the array.
3. **Delete**:
   1. **Best Case**: 𝑂(𝑛). The customer is the last element, resulting in no shifts, but we need to iterate from index 0 to the size of the array (n-1).
   2. **Worst Case**: 𝑂(𝑛). The customer is the first element, requiring shifting of all subsequent elements.

## 4.Calculate the exact efficiency function, using a summation formula or recurrence

1. **Insert:** For InsertOptionA, inserting a new customer always takes constant time since the insertion is done at the end of the array:

Time Complexity: 𝑇(𝑛)=1

1. **Search:** For SearchOptionA, we need to iterate through the array to find the customer. The time complexity depends on the position of the customer in the array:
   * **Best Case**: The customer is the first element. This takes constant time:

Time Complexity: 𝑇best(𝑛)=1

* + **Worst Case**: The customer is not in the array or is the last element. This requires iterating through all 𝑛*n* elements:

Time Complexity: 𝑇worst(𝑛)= n

1. **Delete:** For DeleteOptionA, we first need to find the customer and then shift the remaining elements. The time complexity depends on the position of the customer:

* **Best Case**: The customer is the last element. We don't need to shift any elements:

Time Complexity: 𝑇best(𝑛)=𝑛

* **Worst Case**: The customer is the first element. We need to shift all 𝑛−1 remaining elements:

Time Complexity: 𝑇worst(𝑛)=𝑛+(𝑛−1) = 2𝑛−1

Simplified, we consider the linear term dominant:

Time Complexity: 𝑇worst(𝑛)=𝑂(𝑛)

## 5.What is the efficiency class? (or classes, if the best and worst cases are different).

1. **Insert**
   1. **Analysis:** This algorithm performs a single check to see if there is space in the array (if n < N\_max).
      1. If there is space, it inserts the new customer at the end (Customers[n] ← new\_customer) and increments the count of elements (n ← n+1).
      2. If the array is full, it prints a message.
   2. **Efficiency Class:** Each of these steps (check, insertion, increment, print) takes constant time, regardless of the number of elements in the array.
      1. Time Complexity: 𝑇(𝑛)=1
      2. Efficiency Class: 𝑂(1)
   3. **Explanation:** The efficiency class 𝑂(1) (constant time) means that the time taken to execute the algorithm does not change with the size of the input 𝑛n. It remains constant.
2. **Search**
   1. **Analysis:** This algorithm iterates through the array from the first element to the last (for i ← 0 to n − 1).
      1. For each element, it checks if it matches the customer.
      2. If a match is found, it returns the index.
      3. If no match is found after checking all elements, it returns -1.
   2. **Efficiency Class:** 
      1. **Best Case:** The customer is the first element. The algorithm finds it immediately.
         1. Time Complexity: 𝑇best(𝑛)=1
         2. Efficiency Class: 𝑂(1)
      2. **Worst Case:** The customer is either the last element or not in the array. The algorithm must check all 𝑛 elements.
         1. Time Complexity: 𝑇worst(𝑛)=𝑛
         2. Efficiency Class: 𝑂(𝑛)
   3. **Explanation:** The best case 𝑂(1) means the time taken is constant when the element is found at the first position.
      1. The worst case 𝑂(𝑛) means the time taken increases linearly with the number of elements because each element must be checked.
3. **Delete**
   1. **Analysis:** The algorithm first searches for the customer in the array.
      1. Once found, it shifts all subsequent elements to fill the gap.
      2. If the customer is not found, it prints a message.
   2. **Efficiency Class:**
      1. **Best Case:** The customer is the last element. The algorithm finds it in 𝑛 comparisons but doesn't need to shift any elements.
         * Time Complexity: 𝑇best(𝑛)=𝑛
         * Efficiency Class: 𝑂(𝑛)
      2. **Worst Case:** The customer is the first element. The algorithm finds it in 𝑛 comparisons, and then all 𝑛−1 elements need to be shifted.
         * Time Complexity: 𝑇worst(𝑛)=𝑛+(𝑛−1) = 2𝑛−1
         * Simplified overall Time Complexity to 𝑂(𝑛)
         * Efficiency Class: 𝑂(𝑛)
   3. **Explanation:** Both the best case and worst case have a linear time complexity 𝑂(𝑛).
      1. The best case 𝑂(𝑛) means the time taken is proportional to the number of elements when the element is found at the last position.
      2. The worst case 𝑂(𝑛) means the time taken increases linearly with the number of elements because each element must be checked, and subsequent elements must be shifted.

# Question 2

## 1. What is the problem size?

The problem size, denoted as 𝑛, remains the same as in Option A. It represents the number of customers currently stored in the array.

## 2.What is the basic operation?

1. **Insert:** Inserting a new customer into the sorted array.
2. **Search:** Searching for a customer in the sorted array using binary search.
3. **Delete:** Deleting a customer from the sorted array.

## 3.Does the algorithm have a best case and a worst case?

1. **Insert**:
   1. **Best Case**: O(1). If the array is empty or the new customer is greater than all existing customers, insertion occurs at the end with a single iteration.
   2. **Worst Case**: O(n). If the new customer is smaller than all existing customers, insertion requires shifting all elements to the right.
2. **Search**:
   1. **Best Case**: O(1). If the customer is found at the middle of the array.
   2. **Worst Case**: O(logn). Binary search halves the search space in each step, resulting in logarithmic time complexity.
3. **Delete**:
   1. The time complexity for deletions in Option A remains O(n) in all cases. Regardless of whether the customer is found at the middle of the array (best case) or is the last element or not present in the array (worst case), shifting of elements is required to maintain the order of the array. Therefore, the deletion operation in Option A exhibits linear time complexity, as it necessitates traversing the array and shifting elements, if necessary, to remove the specified customer.

## 4.Calculate the exact efficiency function, using a summation formula or recurrence

**1. Insert:** If the new customer is greater than all existing customers, insertion occurs at the end with a single comparison and copying of elements.

Time Complexity (Best Case): T best(n)=1 (constant time).

**2. Search:**

* **Best Case:** If the customer is found at the middle of the array, the search terminates after a single comparison.

Time Complexity (Best Case): T best(n)=1 (constant time).

* **Worst Case:** If the customer is not in the array or is the last element, binary search divides the array until the search space becomes empty.

Time Complexity (Worst Case): T worst(n)=log2n (logarithmic time).

**3. Delete:** If the customer to be deleted is at the middle of the array, deletion occurs after locating the customer with binary search and shifting elements and the customer is not in the array or is the last element, binary search divides the array until the search space becomes empty. Then, shifting elements may be required.

Time Complexity: T worst(n)=n (linear time).

## 5.What is the efficiency class? (or classes, if the best and worst cases are different).

## **Insert**

* 1. **Analysis:** This algorithm inserts a new customer into the sorted array.
     1. It first finds the correct position for insertion by iterating through the array.
     2. Then, it shifts elements to the right to make space for the new customer.
     3. Finally, it inserts the new customer at the correct position and updates the count of elements.
  2. **Efficiency Class:** Each step in this algorithm (checking, iteration, shifting, insertion, and printing) takes constant or logarithmic time, depending on the number of elements and their order in the array.
     1. Time Complexity: O(n) in the worst case for insertion and shifting, and O(1) for checking and printing.
     2. Efficiency Class: O(n) for the overall operation.
  3. **Explanation:** The efficiency class O(n) indicates that the time taken increases linearly with the number of elements in the array, primarily due to the need for iteration and shifting to maintain the sorted order.

#### **Search**

* 1. **Analysis:** This algorithm searches for a customer in the sorted array using binary search.
     1. It initializes two pointers, l and r, to the start and end of the array, respectively.
     2. It repeatedly divides the search space in half until the customer is found or the search space is empty.
  2. **Efficiency Class:** Binary search halves the search space in each step, resulting in logarithmic time complexity.
     1. Time Complexity: O(logn).
     2. Efficiency Class: O(logn).
  3. **Explanation:** The efficiency class O(logn) signifies that the time taken grows logarithmically with the number of elements in the array. Binary search efficiently reduces the search space, making it highly scalable for large datasets.

1. **Delete**
   1. **Analysis:** This algorithm deletes a customer from the sorted array.
      1. It first searches for the customer using binary search.
      2. If found, it shifts all subsequent elements to the left to fill the gap created by the deletion.
      3. If not found, it prints a message indicating that the customer is not in the array.
   2. **Efficiency Class:** The search operation is performed using binary search, resulting in logarithmic time complexity.
      1. Shifting elements and updating the count of elements may take linear time in the worst case.
      2. Time Complexity: O(logn) for search, and O(n) for deletion.
      3. Efficiency Class: O(n) for the overall operation.
   3. **Explanation:** The efficiency class O(n) suggests that the time taken increases linearly with the number of elements in the array. Although the search operation is efficient, the deletion operation may require shifting elements, which contributes to linear time complexity.

# Question 3

When comparing Option A and Option B for storing and managing customer data, several factors need consideration

it's crucial to acknowledge the differences in the search and deletion operations between Option A and Option B, especially regarding their efficiency and behaviour in the presence of duplicate elements.

**1. Insertions, Deletions, and Searches:**

* **Insertions:** Option A performs constant time insertions at the end of the array, making it efficient when new customers are added frequently.
  + In Option B, insertions involve searching for the correct position and possibly shifting elements, leading to a linear time complexity in the worst case. However, it maintains the array in a sorted order, which can be advantageous for certain search operations.
* **Deletions:** Option A requires linear time for deletions, especially when the deleted element is not at the end of the array. On the other hand, Option B's deletions also depend on the search operation, but it maintains the sorted order of the array, potentially enabling faster searches. However, Option A guarantees that the first inserted element will be deleted first during deletion operations in case of duplicates entries found, respectively.
  + Option B's deletion efficiency is influenced by the search operation, and it maintains the sorted order of the array. However, it's important to note that in the case of duplicate elements, binary search doesn't guarantee which occurrence will be deleted during delete operations.
* **Searches:** Option A's search operation is straightforward but has a linear time complexity. Option B employs binary search for searches, resulting in a logarithmic time complexity, which is advantageous for large datasets. However, Option A ensures that the first inserted element will be returned during search operations.
  + Option B utilizes binary search for searches, resulting in a logarithmic time complexity, which is advantageous for large datasets. However, in the presence of duplicate elements, it's essential to note that binary search may not guarantee that the first occurrence will be returned during search operations.

**2. Dataset Characteristics:**

* The relative number of insertions, deletions, and searches in the dataset can significantly impact the choice between Option A and Option B.
* If the dataset undergoes frequent insertions with occasional deletions and searches, Option A may be more suitable due to its simplicity and constant time insertions.
* However, if the dataset requires frequent searches with a relatively static set of data, Option B's sorted structure and efficient search operations can offer better performance.

**3. Space Complexity:**

Both options have the same space complexity, as they utilize arrays with a fixed maximum size 𝑁max.

**4. Overall Efficiency:**

* Option A may be preferable for dynamic datasets with frequent insertions and modest search requirements.
* Option B could be more appropriate for datasets with a balance between insertions, deletions, and searches, especially when searches need to be optimized.

**5. Edge Cases to Test:**

1. **InsertOptionA:**
   1. Insert into a full array.
   2. Insert null.
   3. Insert an empty string.
   4. Insert a string with different case sensitivity to check case-insensitive search.
2. **SearchOptionA:**
   1. Search for a customer not in the array.
   2. Search for null.
   3. Search for an empty string.
   4. Search with different case sensitivity.
3. **DeleteOptionA:**
   1. Delete a customer not in the array.
   2. Delete null.
   3. Delete an empty string.
   4. Delete with different case sensitivity.

# Question 4

## Source Code (Java Implementation):

## OPTION A:

public class CustomerArray {  
 public String[] getCustomers() {  
 return customers;  
 }  
  
 private String[] customers;  
 private int n;  
 private int N\_max;  
  
 public CustomerArray(int size) {  
 this.N\_max = size;  
 this.customers = new String[N\_max];  
 this.n = 0;  
 }  
  
 public void InsertOptionA(String new\_customer) {  
 if (new\_customer == null) {  
 System.*out*.println("Cannot insert null value");  
 return;  
 }  
 if (n < N\_max) {  
 customers[n] = new\_customer;  
 n++;  
 } else {  
 System.*out*.println("The array is already full");  
 }  
 }  
  
 public int SearchOptionA(String customer) {  
 if (customer == null) {  
 System.*out*.println("Cannot search for null value");  
 return -1;  
 }  
 customer = customer.toLowerCase(); // Convert search query to lowercase  
 for (int i = 0; i < n; i++) {  
 if (customers[i].toLowerCase().equals(customer)) { // Convert customer name to lowercase  
 return i;  
 }  
 }  
 return -1;  
 }  
  
 public void DeleteOptionA(String customer) {  
 if (customer == null) {  
 System.*out*.println("Cannot delete null value");  
 return;  
 }  
 customer = customer.toLowerCase(); // Convert search query to lowercase  
 int i = 0;  
 while (i < n && !customers[i].toLowerCase().equals(customer)) { // Convert customer name to lowercase  
 i++;  
 }  
  
 if (i == n) {  
 System.*out*.println("This customer is not in the array");  
 } else {  
 while (i < n - 1) {  
 customers[i] = customers[i + 1];  
 i++;  
 }  
 customers[n - 1] = null; // Clear the last element  
 n--;  
 }  
 }  
  
 // Test the algorithms  
 public static void main(String[] args) {  
 CustomerArray ca = new CustomerArray(3); // Test with smaller size to hit full array case sooner  
  
 System.*out*.println("Inserting Alice, Bob, Charlie:");  
 ca.InsertOptionA("Alice");  
 ca.InsertOptionA("Bob");  
 ca.InsertOptionA("Charlie");  
  
 System.*out*.println("Inserting another customer to a full array:");  
 ca.InsertOptionA("Dave"); // Should print "The array is already full"  
 System.*out*.println("print array: " + Arrays.*toString*(ca.getCustomers()));  
 System.*out*.println("Search for Bob: " + ca.SearchOptionA("Bob")); // Should print index of Bob  
 System.*out*.println("Search for Dave: " + ca.SearchOptionA("Dave")); // Should print -1  
 System.*out*.println("Search for null: " + ca.SearchOptionA(null)); // Should handle null gracefully  
  
 ca.InsertOptionA("dave"); // Testing case sensitivity  
 System.*out*.println("print array: " + Arrays.*toString*(ca.getCustomers()));  
 System.*out*.println("Search for dave: " + ca.SearchOptionA("DAVE")); // Should print -1 since Dave was not inserted  
  
 System.*out*.println("Delete Bob:");  
 ca.DeleteOptionA("Bob");  
 System.*out*.println("print array: " + Arrays.*toString*(ca.getCustomers()));  
 System.*out*.println("Search for Bob after deletion: " + ca.SearchOptionA("Bob")); // Should print -1  
  
 System.*out*.println("Delete a customer not in the array:");  
 ca.DeleteOptionA("Eve"); // Should print "This customer is not in the array"  
  
 System.*out*.println("Delete null:");  
 ca.DeleteOptionA(null); // Should handle null gracefully  
  
 System.*out*.println("Inserting Charlie again to test duplicate:");  
 ca.InsertOptionA("Charlie"); // Should handle insertion normally  
 System.*out*.println("print array: " + Arrays.*toString*(ca.getCustomers()));  
 System.*out*.println("Search for Charlie after duplicate: " + ca.SearchOptionA("Charlie")); // Should print the correct index  
  
 System.*out*.println("Inserting empty string:");  
 ca.InsertOptionA(""); // Should handle empty string gracefully  
 System.*out*.println("Search for empty string: " + ca.SearchOptionA("")); // Should print the correct index  
 ca.DeleteOptionA(""); // Should delete the empty string gracefully  
 System.*out*.println("Search for empty string after deletion: " + ca.SearchOptionA("")); // Should print -1  
 }  
}

## OPTION B

public class CustomerArray {  
  
 private String[] customers;  
 private int n; // the number of customers currently stored  
 private int N\_max; // maximum capacity of customers array  
  
 public CustomerArray(int size) {  
 this.N\_max = size;  
 this.customers = new String[N\_max]; // initialize null value of n size of customer array  
 this.n = 0; // default set current customer to 0  
 }  
  
 public void InsertOptionB(String new\_customer) {  
 if (new\_customer == null) {  
 System.*out*.println("Cannot insert null value");  
 return;  
 }  
 if((new\_customer.trim()).isBlank()){  
 System.*out*.println("Cannot insert empty value");  
 return;  
 }  
 if (n < N\_max) {  
 int i = 0;  
 while (i < n && customers[i].compareTo(new\_customer) < 0) {  
 i++;  
 }  
 int j = n - 1;  
 while (j >= i) {  
 customers[j + 1] = customers[j];  
 j--;  
 }  
 customers[i] = new\_customer;  
 n++;  
 } else {  
 System.*out*.println("The array is already full");  
 }  
 }  
  
 public int SearchOptionB(String customer) {  
 if (customer == null) {  
 System.*out*.println("Cannot search for null value");  
 return -1;  
 }  
 if((customer.trim()).isBlank()){  
 System.*out*.println("Cannot search for empty value");  
 return -1;  
 }  
 customer = customer.toLowerCase(); // Convert search query to lowercase  
 int l = 0;  
 int r = n - 1;  
 while (l <= r) {  
 int m = (l + r) / 2;  
 String currentCustomer = customers[m].toLowerCase(); // Convert customer name to lowercase  
 if (currentCustomer.equals(customer)) {  
 return m;  
 } else if (currentCustomer.compareTo(customer) > 0) {  
 r = m - 1;  
 } else {  
 l = m + 1;  
 }  
 }  
 return -1;  
 }  
  
 public void DeleteOptionB(String customer) {  
 if (customer == null) {  
 System.*out*.println("Cannot delete null value");  
 return;  
 }  
 customer = customer.toLowerCase(); // Convert search query to lowercase  
 int l = 0;  
 int r = n - 1;  
 while (l <= r) {  
 int m = (l + r) / 2;  
 String currentCustomer = customers[m].toLowerCase(); // Convert customer name to lowercase  
 if (currentCustomer.equals(customer)) {  
 while (m < n - 1) {  
 customers[m] = customers[m + 1];  
 m++;  
 }  
 customers[n - 1] = null; // Nullify the value after deletion  
 n--; // reduce number of customer count by 1  
 return;  
 } else if (currentCustomer.compareTo(customer) > 0) {  
 r = m - 1; // search in left side of array by changing right index to middle index - 1  
 } else {  
 l = m + 1; // search in right side of array by changing left index to middle index + 1  
 }  
 }  
 System.*out*.println("This customer is not in the array");  
 }  
  
 public void increasedCapacity(int n\_max) {  
 this.N\_max = n\_max;  
// create new array with increased capacity  
 this.customers = Arrays.*copyOf*(this.customers,this.N\_max);  
 }  
  
 @Override  
 public String toString() {  
 return "\n" + "-".repeat(10) + "\nArray: " + Arrays.*toString*(customers) + "\nSize: " + this.n + "\nCapacity: " + this.N\_max + "\n" + "-".repeat(10) + "\n";  
 }  
  
 // Test the algorithms  
 public static void main(String[] args) {  
 CustomerArray ca = new CustomerArray(3); // Test with smaller size to hit full array case sooner  
  
 System.*out*.println("Inserting Alice, Bob, Charlie:");  
 ca.InsertOptionB("Alice");  
 ca.InsertOptionB("Bob");  
 ca.InsertOptionB("Charlie");  
 System.*out*.println(ca);  
  
 System.*out*.println("Inserting another customer to a full array:");  
 ca.InsertOptionB("Dave"); // Should print "The array is already full"  
  
 System.*out*.println("Search for Bob: " + ca.SearchOptionB("bob")); // Should print index of Bob  
 System.*out*.println("Search for Dave: " + ca.SearchOptionB("dave")); // Should print -1  
 System.*out*.println("Search for null: " + ca.SearchOptionB(null)); // Should handle null gracefully  
  
 System.*out*.println("Inserting dave: ");  
 ca.InsertOptionB("dave"); // Testing case sensitivity  
 System.*out*.println("Search for dave: " + ca.SearchOptionB("DAVE")); // Should print -1 since Dave was not inserted  
  
 System.*out*.println("Delete Bob:");  
 ca.DeleteOptionB("Bob");  
 System.*out*.println(ca);  
 System.*out*.println("Search for Bob after deletion: " + ca.SearchOptionB("Bob")); // Should print -1  
  
 System.*out*.println("Delete a customer not in the array:");  
 ca.DeleteOptionB("Eve"); // Should print "This customer is not in the array"  
  
 System.*out*.println("Delete null:");  
 ca.DeleteOptionB(null); // Should handle null gracefully  
  
 System.*out*.println("Inserting Charlie again to test duplicate:");  
 ca.InsertOptionB("Charlie"); // Should handle insertion normally  
 System.*out*.println(ca);  
  
 System.*out*.println("Search for Charlie after duplicate: " + ca.SearchOptionB("Charlie")); // Should print the correct index  
  
 System.*out*.println("Increase capacity of array to 4");  
 ca.increasedCapacity(4);  
 System.*out*.println("Inserting empty string:");  
 ca.InsertOptionB(" "); // Should handle empty string gracefully  
 System.*out*.println(ca);  
 System.*out*.println("Search for empty string: " + ca.SearchOptionB("")); // Should print the correct index  
 ca.DeleteOptionB(" "); // Should delete the empty string gracefully  
 System.*out*.println("Search for empty string after deletion: " + ca.SearchOptionB("")); // Should print -1  
  
 System.*out*.println("Final Customer Array: " + ca);  
 }  
}

## Justification:

* **Correctness Tests:** Run tests to verify that the algorithms function correctly under various scenarios, including edge cases such as empty arrays, full arrays, and different combinations of insertions, deletions, and searches.
* **Efficiency Tests:** Measure the execution time of each algorithm using datasets of varying sizes. Compare the results with the expected efficiency analysis to validate the time complexity assumptions made in the analysis.

# Question 5

When transitioning from an array to a linked list for storing customer data, several aspects of the approach would change, impacting efficiency in various ways:

**1. Data Structure:**

* Instead of using a fixed-size array, a linked list dynamically allocates memory for each element as needed. Each node in the linked list contains both the customer data and a reference to the next node.

**2. Insertions:**

* In a linked list, insertions can be more efficient than in an array, especially when inserting at the beginning or middle of the list. Since there's no need to shift elements, insertions typically involve updating pointers to insert the new node.
* In Option A, where insertions are made at the end of the array, the efficiency remains constant. This is because the insert operation directly places the new element at the last position of the array without needing to traverse the entire array. The time complexity of this operation is O(1), as it does not depend on the size of the array or the number of elements already present. Therefore, regardless of the size of the array or the number of elements, inserting a new element at the end of the array is always a constant-time operation.
* For Option B, where insertions are made in sorted order, the efficiency could still be reasonable depending on the approach used. Inserting a new node in sorted order may require traversing the list to find the correct position, potentially resulting in a linear time complexity like the worst-case scenario in the array-based approach.

**3. Deletions:**

* Deletions in Option A involve searching for the node to be deleted, which may require traversing the entire array in the worst case. If the position of the node to be deleted is known, the deletion operation is straightforward and can be performed in constant time O(1) by simply updating the value at the corresponding index. However, if the position is unknown, it may require a linear search to find the node to be deleted, resulting in a time complexity of O(n)
* In Option B, if the position of the node to be deleted is unknown, binary search may be employed to locate the node to be deleted. Once the node is found, the deletion operation still involves shifting elements after the deleted node to maintain the sorted order, resulting in a total time complexity of O(logn+n), where logn represents the time taken for binary search and n represents the time taken for shifting elements so overall time complexity is O(n).
* Deletions in a linked list can be efficient, especially when the node to be deleted is known. It involves updating pointers to bypass the node to be deleted, which can be done in constant time O(1). However, if the position of the node to be deleted is unknown and needs to be searched, the efficiency could degrade. Traversing the linked list to find the node to be deleted may require linear time O(n), particularly if the list is unsorted.

**4. Searches:**

* Searches in a linked list typically involve traversing the list sequentially until the desired element is found or reaching the end of the list.
* For Option A, searches could have a linear time complexity like array-based searches, depending on the length of the list and the position of the desired element. This is because, in Option A, the elements are stored in an array without any specific order, requiring a sequential search through the array.
* For Option B, searches may also involve traversing the list until finding the desired element or reaching a node greater than the target value, which could be more efficient than array-based searches & linked list in some cases due to the sorted order of the list.

**5. Impact on Efficiency:**

* Overall, the efficiency of operations in a linked list may vary depending on the specific operation and the characteristics of the data.
* Insertions and deletions may be more efficient in a linked list compared to an array, especially for large datasets or when operations are frequently performed at the beginning or middle of the list.
* Searches may have similar or slightly different efficiency compared to array-based searches, with potential advantages in Option B due to the sorted order of the list.

In conclusion, for the scenario of managing a customer array, each approach offers distinct advantages and trade-offs:

**Option A (Array):** Option A provides simplicity and constant-time insertions at the end of the array, making it suitable for scenarios where insertions are frequent, and the order of elements is not critical. However, deletions and searches may become less efficient as the array grows larger due to linear time complexity.

**Option B (Array):** Option B excels in maintaining a sorted order, which enhances search efficiency, especially for datasets with frequent searches. Although insertions and deletions may incur additional overhead due to the need for maintaining sorted order, binary search can still offer improved search performance compared to unsorted arrays.

**Linked List:** Linked lists offer flexibility in memory allocation and efficient insertions and deletions, particularly when the position of elements is known. However, searching for elements may degrade efficiency, especially when the list is unsorted, as it requires sequential traversal. Despite this, linked lists remain a suitable choice for dynamic datasets with frequent insertions and deletions.

In summary, the choice between the array-based options and linked lists depends on the specific requirements of the scenario, including the frequency of insertions, deletions, and searches, as well as the importance of maintaining sorted order for search efficiency.

# References

*[1]* [*https://www.geeksforgeeks.org/data-structures/linked-list/#what-is-a-linked-list*](https://www.geeksforgeeks.org/data-structures/linked-list/#what-is-a-linked-list)

*[2]* [*https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure*](https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure#:~:text=A%20linked%20list%20is%20the,reference%20to%20the%20next%20node.)

# Appendix