



RIPHAH
INTERNATIONAL UNIVERSITY

Comparing the performance of linked lists and arrays in terms of insertion and deletion.

1. Arrays

Insertion

- **At the End:**
 - **Time Complexity:**
 - $O(1)$ if there's space available.
 - If the array is full and needs to be resized, the complexity becomes $O(n)$ due to the need to copy all elements to a new array.
- **At the Beginning or Middle:**
 - **Time Complexity:** $O(n)$.
 - All elements after the insertion point must be shifted one position to the right.

Deletion

- **From the End:**
 - **Time Complexity:**
 - $O(1)$ if deleting the last element.
- **From the Beginning or Middle:**
 - **Time Complexity:** $O(n)$.
 - All elements after the deletion point must be shifted one position to the left.

2. Linked Lists

Insertion

- **At the Beginning:**

Time Complexity: $O(1)O(1)O(1)$.

- A new node is created and linked to the head without shifting any elements.

- **At the End:**

Time Complexity:

- $O(n)O(n)O(n)$ if traversing from the head to find the last node.
- If you maintain a tail pointer, it can be $O(1)O(1)O(1)$.

- **At a Specific Position:**

Time Complexity: $O(n)O(n)O(n)$.

- You must traverse to the desired position first.

Deletion

- **From the Beginning:**

Time Complexity: $O(1)O(1)O(1)$.

- Simply update the head to point to the next node.

- **From the End:**

Time Complexity: $O(n)O(n)O(n)$ unless a tail pointer is maintained.

- **From a Specific Position:**

Time Complexity: $O(n)O(n)O(n)$.

- Traverse to the position and then adjust pointers.

Summary of Time Complexities

Operation	Arrays	Linked Lists
Insertion (at end)	$O(1)O(1)O(1)$	$O(n)O(n)O(n)$ or $O(1)O(1)O(1)$ (with tail)
Insertion (at start)	$O(n)O(n)O(n)$	$O(1)O(1)O(1)$
Insertion (at middle)	$O(n)O(n)O(n)$	$O(n)O(n)O(n)$
Deletion (from end)	$O(1)O(1)O(1)$	$O(n)O(n)O(n)$ (unless tail)
Deletion (from start)	$O(n)O(n)O(n)$	$O(1)O(1)O(1)$
Deletion (from middle)	$O(n)O(n)O(n)$	$O(n)O(n)O(n)$

Conclusion

- **Use Cases:**
 - **Arrays** are more efficient for accessing elements via indices and when there are many read operations.
 - **Linked Lists** are better for frequent insertions and deletions, particularly at the beginning or when the size of the data structure is dynamic and can change frequently.
- **Memory Considerations:**
 - Arrays have contiguous memory allocation, which can lead to better cache performance. However, they have fixed sizes (unless resized). ◦ Linked lists use more memory per element due to storing pointers, but they can dynamically grow or shrink as needed.

In summary, the choice between using an array or a linked list depends on the specific requirements of your application, particularly concerning how often insertions and deletions occur compared to access operations.