**Comparing Linked Lists and Dynamic Arrays**

**Time Complexity**

**Linked Lists:**

* **Insert at index**: O(n) (you need to go through the list to reach the index)
* **Delete at index**: O(n) (you need to go through the list to reach the index)
* **Get size**: O(n) (if you don't have a size counter)
* **Is empty**: O(1)
* **Rotate right**: O(n)
* **Reverse**: O(n)
* **Append**: O(n) (if you don't have a tail pointer), O(1) (if you do have a tail pointer)
* **Prepend**: O(1)
* **Merge**: O(n) (depends on the length of both lists)
* **Interleave**: O(n + m) (n and m are sizes of the two lists)
* **Get middle**: O(n)
* **Index of**: O(n)
* **Split at index**: O(n)

**Dynamic Arrays:**

* **Insert at index**: O(n) (you need to move elements to make space)
* **Delete at index**: O(n) (you need to move elements to fill the gap)
* **Get size**: O(1)
* **Is empty**: O(1)
* **Rotate right**: O(n)
* **Reverse**: O(n)
* **Append**: Amortized O(1)
* **Prepend**: O(n) (you need to move elements to make space)
* **Merge**: O(n + m) (n and m are sizes of the two arrays)
* **Interleave**: O(n + m) (n and m are sizes of the two arrays)
* **Get middle**: O(1)
* **Index of**: O(n)
* **Split at index**: O(n)

**Space Complexity**

**Linked Lists:**

* Each node needs extra space for a pointer/reference.
* O(n) for n elements plus O(n) for pointers.

**Dynamic Arrays:**

* Needs a continuous block of memory.
* O(n) for n elements.
* Extra space for resizing: can be up to O(n) (usually, the array doubles in size when full).

**Advantages and Disadvantages**

**Linked Lists:**

**Advantages:**

* **Dynamic Size**: Can grow and shrink easily without needing a big block of memory.
* **Efficient Insertions/Deletions**: Adding or removing nodes is easy if you know the position.
* **Memory Utilization**: Better if you do a lot of adding and removing because no resizing is needed.

**Disadvantages:**

* **Memory Overhead**: Needs extra memory for storing pointers.
* **Sequential Access**: Accessing elements is slow (O(n)) because you have to go through the list.
* **Cache Locality**: Bad performance because memory locations are scattered.

**Dynamic Arrays:**

**Advantages:**

* **Random Access**: Accessing any element is very fast (O(1)).
* **Memory Utilization**: Good if the array size is stable or grows slowly.
* **Compact Memory**: Stores elements in continuous memory, which is good for performance.

**Disadvantages:**

* **Fixed Size**: Needs a fixed initial size. Resizing can be expensive (O(n)).
* **Insertions/Deletions**: Adding or removing elements is slow (O(n)) because you need to move elements.
* **Wasted Space**: Can have unused memory, wasting space.

**Report on Comparison**

**Introduction:** Linked lists and dynamic arrays are key data structures in computer science for storing collections of elements. Each has its own strengths and weaknesses, making them good for different uses. This report compares these structures in terms of time complexity, space complexity, and practical pros and cons.

**Time Complexity:** Linked lists allow efficient adding and removing elements, especially at the beginning, with O(1) complexity. However, accessing elements in a linked list is slower (O(n)) because you have to go through the list. Dynamic arrays provide fast access (O(1)) but slow down when adding or removing elements because you need to move them.

**Space Complexity:** Linked lists need extra space for pointers, so they have O(n) space complexity plus O(n) for pointers. Dynamic arrays have O(n) space complexity but can need extra space when resizing.

**Advantages and Disadvantages:** Linked lists are good for frequent adding and removing elements but have poor performance with memory and accessing elements. Dynamic arrays are excellent for fast access to elements and better memory performance but are less efficient when resizing or moving elements.

**Conclusion:** Choosing between linked lists and dynamic arrays depends on what you need for your application. Linked lists are great for frequent changes, while dynamic arrays are best for fast access and stable size. Knowing these trade-offs helps in choosing the right data structure for the best performance.

Top of Form