DOCUMENT

Challenge 1-9 are in lapPracticeW2.txt

Challenge 10 – Compare with Arrays

Q: For each operation (insert, delete, access), write the complexity for arrays vs. linked lists.

Operations	**Array**	**Linked List**	**Comments**
Access by Index	**O(1)**	**O(n)**	**Arrays have direct, random access. Linked lists require sequential traversal.**
Insert at Front	**O(n)**	**O(1)**	**Arrays require shifting all elements. Linked lists just change the head pointer.**
Delete from Front	**O(n)**	**O(1)**	**Same reasoning as insertion at the front.**
Insert at End	**O(1)**	**O(1)** (with tail pointer)	**Both can be efficient, but array appending may occasionally require resizing (O(n)).**
Delete from End	**O(1)**	**O(n)**	**Arrays can remove the last element directly. Linked lists need to find the new last node.**
Insert/Delete in Middle	**O(1)**	**O(n)**	**Both require finding the location (O(n) for lists, O(1) for arrays if index is known), but insertion/deletion itself is O(1) for lists after finding the spot, while arrays require shifting.**

Conclusions?

Discuss: In what situations is a linked list clearly better?
A linked list is clearly better when the primary operations involve frequent **insertions and deletions at the beginning or at given positions** (especially when the position is not the very end), and when the total number of elements is unknown. This is because insertions/deletions don't require shifting subsequent elements plus the list can grow and shrink dynamically without the need for resizing operations.

1. Which operations were O(1) in linked lists but O(n) in arrays?

- Inserting at the front.
- Deleting from the front.

2. Which operation is clearly faster in arrays than in linked lists?

- **Accessing an element by its index** (random access). This is O(1) for arrays but O(n) for linked lists.

3. Why must we manage memory carefully in linked lists?
Because every node is dynamically allocated on the heap. If we forget to use the delete operator after removing a node, we create a **memory leak**. We must use new for every insertion. This manual memory management is prone to error as we could forget, compared to arrays which are often managed automatically.

4. What does the head pointer represent?
The head pointer stores the memory address of the **first node** in the linked list. It is the entry point to the entire data structure. If you have the head, you can access every other element.

5. What happens if we lose the head pointer?
If the head pointer is lost (e.g., set to nullptr without being saved) and no other pointer references the first node, we lose all access to the entire linked list. (boohoohoo it's all gone lol) The memory for all nodes remains allocated but becomes unreachable, resulting in a massive memory leak. The program can no longer access or free that memory(ure absolutely cooked atp).

Scenario Analysis: Choose Array or Linked List

1. Real-time scoreboard where new scores are always added at the end and sometimes removed from the front.

- **Better Choice: Linked List**
- **Justification:** Adding at the end is O(1) with a tail pointer and removing from the front—is O(1) for a linked list but O(n) for an array. A Queue (often implemented with a linked list) is perfect for this.

2. Undo/Redo feature in a text editor, where operations are frequently added and removed at the front.

- **Better Choice: Linked List**
- **Justification:** This is a classic Stack data structure. Adding and removing from the front (push and pop) are both O(1) operations in a linked list. An array would require shifting all other elements on every undo/redo operation.

3. Music playlist that lets users add and remove songs anywhere in the list.

- **Better Choice: Linked List**
- **Justification:** While finding the song is O(n) in both, once the position is found, inserting or deleting a song from the middle of a linked list is more efficient (O(1) pointer manipulation) than in an array, which requires shifting all subsequent songs (O(n)).

4. Large dataset search where random access by index is needed often.

- **Better Choice: Array**
- **Justification:** If the primary operation is accessing elements by their position (e.g., get item 1,245), arrays provide instant O(1) access. A linked list would require a slow O(n) traversal for each access.

5. Simulation of a queue at a bank, where customers join at the end and leave at the front.

- **Better Choice: Linked List**
- **Justification:** A linked list with a head and tail pointer can perform both enqueue (insert at end) and dequeue (delete from front) operations in O(1) time. An array would make deleting from the front an O(n) operation taking more time.

6. Inventory system where you always know the item's index and need quick lookups.

- **Better Choice: Array** (or Vector)
- **Justification:** The key need is "quick lookups" by a known index. This is the core strength of arrays (O(1)) and the core weakness of linked lists (O(n)).

7. Polynomial addition program where terms are inserted and deleted dynamically.

- **Better Choice: Linked List**
- **Justification:** Polynomials are naturally sparse (many coefficients can be zero). A linked list allows efficient storage of only the non-zero terms and dynamic insertion/deletion of new terms as the polynomial is manipulated, without worrying about pre-allocated size or shifting.

8. Student roll-call system where the order is fixed and access by index is frequent.

- **Better Choice: Array**
- **Justification:** The "order is fixed" and "access by index is frequent" are the defining criteria. Students are called by roll number (1, 2, 3...), which maps directly to array indices, making access extremely fast (O(1)).