Week2 – Lab Practice

1. Insert at the Front:
   - Time complexity: O(1)
   - In an array, inserting at index 0 requires shifting all existing elements one position to the right, which takes O(n) time
   - In linked list, we just update pointers
   - ***Logic:***
     - Create a new node
     - Set the next pointer of the new node to the current head of the linked list
     - Update the head pointer to the new node
2. Insert at the End:
   - Time complexity: O(n)
   - Yes, we must traverse the entire list until we reach the last node then set the next pointer of the last node to the new node
   - The difference is in an array we can access the last index directly in O(1) time but in linked list we don't have the index to access, so we need to traverse until we reach the last node
   - ***Logic:***
     - Traverse the list to reach the last node
     - Create a new node
     - Set the next pointer of the last node to the new node
3. Insert in the Middle:
   - Time complexity: O(1)
   - The two arrows (pointers) need to be changed:
     - The next pointer of the node before the insertion point
     - The next pointer of the new node
   - In array, inserting in the middle required shifting all elements after that point to right one position O(n) time
   - In linked list, only pointer changed are need
   - ***Logic:***
     - Traverse the list to find the node after which we want to insert the new node
     - Create a new node
     - Set the next pointer of the new node point to the next of current node
     - Update the next pointer of current node points to the new node
4. Delete from the Front:
   - Time complexity: O(1)
   - After delete front the head pointer is moved to the second node
   - Then delete the old head node to prevent memory leaks
   - ***Logic:***
     - Check if the list is empty (head == NULL). If so, there's nothing to delete
     - Create temp node which point to head
     - Update the head pointer to point to the second node (head = head -> next)
     - Delete temp
5. Delete from the End:
   - Time complexity: O(n)
   - ***Logic:***

  - Check if the list is empty (head == NULL)
  - Traverse the list to find the second-last node
  - Point its **nex**t pointer to **NULL**
  - Free or delete the last node
- To find the node before the last one we use:
- while(cur->next->next != nulptr) cur = cur -> next;
- So, if we reach the last node our **cur node** will be stay at the second last node because our condition checks the **next node** of the **next cur node**

6. Delete from the Middle

- Logic:
  - Validate the position (ensure its within bounds)
  - Traverse to the node just before the target position
  - Update its **next** pointer to skip the target node
  - Free or delete the target node
- Only one pointer changes: the previous node's **next** of the node before the target position
- If we forgot to free memory, the deleted node still exists in memory and causes a memory leak.

7. Traverse the List:

- Time complexity: O(n)
- Start from the head and follow each **next** pointer until the pointer is NULL, printing or processing each node's value.
- In array, accessing the ith element is O(1) because memory is contiguous
- In a liked list, we can't directly access ith element, we must traverse from the head until reach the end

8. Swap two Nodes

- Time complexity: O(n)
- ***<u>Logic of Swapping Nodes without swapping data:</u>***
  - The first check if the **(posA == posB)** nothing needs to be swap just return
  - If either position is outside the list bounds, return
  - Initialize pointer: **prevA**, **currA**, **prevB** and **currB** for prepare to find the **current node** and **previous node** of **posA** and **posB**
  - Find nodes at **posA** and **posB**
  - **currA** points to node at position **posA**
  - **prevA** points to node before it
  - The same for position **posB**
  - After we got the previous and current of **posA** and **posB**
  - We check if **currA** is not head**,** update **prevA->next** to point to **currB**
  - Otherwise**, currA** is the head**,** so make **currB** the new head
  - The same logic for **currB**
  - After that we create a **temp** to store the **currB->next**
  - Then: **currB ->** next is set to **currA -> next**
  - **currA ->** is set to **temp**
- Swapping values is easier than swapping without values:
  - We just exchange the **value** fields of the two nodes
  - Take O(1) time after finding the nodes
  -

9. Search in Linked List:

- To search for a value in linked list, we start at the head and traverse the list, one node at a time. In each node we check if its data matches the target value
- If the target value is found, stop and return true
- If we reach the end of the list without finding the value, stop and return false
- Searching in Linked List is similar to Linear search because it both involve checking of all elements until matches the target
- The both have time complexity: O(n)
- For random access: Array is faster than linked list because array store its elements in contiguous memory locations. So, if we want to access element at index i-th we can access it directly, this is constant time: O(1).
- But in liked list we need to traverse from the head until the position which we want, so it takes O(n) time in worst case

10. Compare with Arrays

Operation	Array	Linked List
Access (by index)	O(1)	O(n)
Insert (at a specific position)	O(n)	O(n)
Delete (at a specific position)	O(n)	O(n)

- Insertion/Deletion at the beginning:
  - Linked List: O(1) create a new node and adjust the head pointer
  - Array: O(n) Every existing element must be shifting
- In situation is Linked list clearly better when we insertion/deletion the value especially at the beginning, or when the size of the data structure needs to change often.