Linked List Lab Report

Challenge 1 - Insert at the front

```
void insertEnd(int val){
            if (n==0){
                insertFront(val); //if empty, insert at front
                return;
            }
            cur = head; //traverse to the end of the list
            while (cur->next != nullptr){
                cur = cur->next; //move to next node
            }
            Node* nNode = new Node {val, nullptr}; //create new node
            cur->next = nNode; //link last node
            n++; //increment size

        }
```

- Time Complexity : O(1)

- Comparison : For array, you would need to shift all nodes to the right first before inserting, compared to linked list, where you only need to update the head pointer.

Challenge 2 - Insert at the end

```
void insertEnd(int val){
            if (n==0){
                insertFront(val); //if empty, insert at front
                return;
            }
            cur = head; //traverse to the end of the list
            while (cur->next != nullptr){
                cur = cur->next; //move to next node
            }
            Node* nNode = new Node {val, nullptr}; //create new node
            cur->next = nNode; //link last node
            n++; //increment size

        }
```

- Time Complexity : O(n)

- Comparison : For linked list, you would have to traverse through the entire list to find the last node, and then add the new node after it, whereas arrays only need to add the node at the end (except when the array is full, then you’ll have to create a bigger array and copy everything back in). The reason why you need to traverse is because there’s no tail pointer in the code, so you have to search from head until you find the last node.

<u>Challenge 3 - Insert in the middle</u>

```
void insertMiddle(int val, int pos){
            if (pos > n){
                cout << "Out of node range!\n"; //if pos is greater than size,
it's invalid
                return;
            }
            if (pos == 0){
                insertFront(val); //if pos is 0, insert at front
                return;
            }
            if (pos == n){
                insertEnd(val); //if pos is size, insert at end
                return;
            }
            cur = head; //traverse to the node before the position
            for (int i=0; i<pos-1; i++){
                cur = cur->next; //move to next node
            }
            Node* nNode = new Node {val, nullptr}; //create new node
            nNode->next = cur->next; //link new node to the next node
            cur->next = nNode; //link previous node to new node
            n++; //increment size
        }
```

-Pointers needed to change :

+nNode->next to cur->next (make new node points to the next node)

+cur->next to nNode (make the previous node points to the new node)

-Comparison : For linked list, you only need to update the pointer, whereas the arrays requires shifting the nodes in order to insert.

Challenge 4 – Delete at the front

```
void deleteFront(){
            if (n==0){
                cout << "Empty list!\n"; //there's nothing to delete when empty
                return;
            }
            Node* delNode = head; //node to be deleted
            head = head->next; //update head to next node
            delete delNode; //free memory to not cause memory leak
            n--; //decrement size
        }
```

-What happened : The head pointer update to the next node, thus making it the first node. The deleted note's memory is freed in order to not cause a memory leak.

Challenge 5 – Delete at the end

```
void deleteEnd(){
            if (n==0){
                cout << "Empty list!\n"; //there's nothing to delete when empty
                return;
            }
            if (n==1){
                deleteFront(); //if only 1 node, delete at front
                return;
            }
            cur = head; //traverse to the second last node
            while (cur->next->next != nullptr){
                cur = cur->next; //move to next node
            }
            Node* delNode = cur->next; //node to be deleted
            cur->next = nullptr; //update the second last node to be the last
node
            delete delNode; //free memory to not cause memory leak
            n--; //decrement size
        }
```

Finding node before the last : You need to traverse through the list until you encounter cur->next->next == nullptr, because cur is the second to last node. Then the memory is freed.

Challenge 6 – Delete in the middle

```
void deleteMiddle(int pos){
            if (pos > n){
                cout << "Out of node range!\n"; //if pos is greater than size,
it's invalid
                return;
            }
            if (pos==0){
                deleteFront(); //if pos is 0, delete at front
                return;
            }
            if (pos==n-1){
                deleteEnd(); //if pos is size-1, delete at the end
                return;
            }
            cur = head; //traverse to the node before the position
            for (int i=0; i<pos-1; i++){
                cur = cur->next; //move to next node
            }
            Node* delNode = cur->next; //node to be deleted
            cur->next = cur->next->next; //link previous node to the next node
            delete delNode; //free memory to not cause memory
            n--; //decrement size
        }
```

Arrow changes : cur->next is updated to point the node after the one being deleted.

Memory : When you forget to free memory, it could cause a memory leak.

Challenge 7 – Traverse the list

```
void traverse(){
            if (n==0){
                cout << "(empty!)\n"; //if empty, print empty
            }
            cur = head; //start from head
            while (cur != nullptr){
                cout <<cur->value<<"->"; //print value
                cur = cur->next; //move to next node
            }
            cout<<endl;
        }
```

Comparison : Traversal differ from direct arr[i] access as arrays allow random access through index (making it O(1)) and linked list requires traversal through each node from head (making it O(n)).

Challenge 8 – Swap two nodes

```
void swapNodes(int pos1, int pos2){
            if (pos1 == pos2){
                return; //no need to swap if both pos are the same
            }
            if (pos1 > n || pos2 > n){
                cout << "Out of node range!\n"; //if pos is greater than size,
it's invalid
                return;
            }
            Node *node1 = head, *node2 = head; //start from head
            for (int i=0; i<pos1; i++){
                node1 = node1->next; //move to next node
            }
            for (int i=0; i<pos2; i++){
                node2 = node2->next; //move to next node
            }
            int temp = node1->value; //swap the two node values
            node1->value = node2->value; //swap the node values
            node2->value = temp; //assign temp to node2 value
        }
```

Comparison : It is much easier to swap values than to swap links, because swapping linked requires you to change several pointers and swapping value only need to change them in the node itself and only need little assignments.

Challenge 9 – Search in linked list

```
void searchValue(int val){
            cur = head; //start from head
            int pos = 0;
            while (cur != nullptr){
                if (cur->value == val){
                    cout << "Value "<<val<<" found at position "<<pos<<endl;
//print pos if found
                    return;
                }
                cur = cur->next; //move to next node
                pos++; //increment pos
            }
            cout << "Value "<<val<<" not found!\n"; //print not found if reached
the end
        }
```

Comparison : It’s similar to linear search in arrays as it both requires checking each and every elements continuously until finding the right target. Though arrays are much faster for random access index, as they use direct memory addressing.

Challenge 10 – Compare with arrays

Operations	Array	Linked List
Insert at front	O(n)	O(1)
Insert at end	O(1)	O(n)
Insert in middle	O(n)	O(n)
Delete at front	O(n)	O(1)
Delete at end	O(1)	O(n)
Delete in middle	O(n)	O(n)
Random Access	O(1)	O(n)
Search	O(n)	O(n)

Linked list is so much better when It needs frequent insertion/deletion at the beginning, as it’s time complexity is O(1).

Scenario Analysis : Array or Linked list

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

Linked list, because linked list is faster when removing from the front as arrays would require shifting all the elements.

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

Linked list, because both inserting/deleting elements in linked list is faster than arrays, with it's time complexity of O(1).

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

Linked list, even though both are O(n) when having to find position, for linked list, it's O(1) when having to actually insert, whereas array would be O(n) again.

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

Array, because when frequent random search is needed often, array could just use the index and direct memory addressing.

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

Linked list, because when linked list had a tail pointer, it's faster when adding elements at the end and removing at the front, with it's time complexity of O(1).

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

Array, because you already know the element's index and could just use direct index access, whereas linked list's time complexity would be O(n) for every lookups.

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

Linked list, because when dealing with dynamic size changes, linked list is better at handling it, and easier to insert and delete than arrays.

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

Array, because it’s already a fixed order, and doesn’t require insert/delete any elements and easier to access through index.