Week2: LinkedList

1. Challenge : Insert at the front

```
void insertFront(int val){
        Node* nNode = new Node{val, nullptr};
        nNode->next= head;
        head=nNode;
        n++;
```

Explanation: A new node is created with the given value, and its next pointer is made to point to the current head of the list. Then, the head is updated to this new node. This operation takes constant time (O(1)), because it only changes a couple of pointers; Thus, no shifting or looping is needed. This is much easier than inserting at index 0 in an array, because in an array, every existing element must be moved one position to the right to make space at the front, which takes O(n) time.

2. Challenge : insert at the end

```
void insertEnd(int val){
        if(n==0){
            insertFront(val);
            return;
        }
        cur=head; while(cur->next !=nullptr){
            cur=cur->next;
        }
        Node* nNode=new Node{val,nullptr};
        cur->next = nNode;
        n++;
    }
```

Explanation: To insert a new node at the end of a linked list, we first check if the list is empty. If it is, we just call insertFront() since the new node will be both the head and tail. Otherwise, we start from the head and traverse the list until we reach the last node (where next is nullptr). Then, we create a new node and link it to the end. This process takes O(n) time because we must go through every node to reach the end. In contrast, inserting at the end of an array is faster because it takes O(1) time if there's space, since arrays have direct indexing and don't require traversal.

3. Challenge : Insert in the middle

```
void insertMiddle(int val,int pos){
        if(pos>n){
            cout<<"Out of range.\n";
            return;
        }
        if (pos==0){
            insertFront(val);
            return;
        }
        cur=head;
        for(int i=0;i<pos-1;i++){
cur = cur->next;
        }
        Node* nNode=new Node{val, nullptr};
        nNode->next=cur->next;
        cur->next=nNode;
        n++;
    }
```

Explanation : The two pointers that need to change are:

nNode->next = cur->next; → makes the new node point to the node after it.

cur->next = nNode; → makes the previous node (cur) point to the new node.

First you set the new node's next pointer to point at the node that currently follows your current node. Then you change the current node's next pointer to point at the new node you just created. This is different from an array because inserting into an array requires you to shift all the following elements one position to the right to make a space which is a much slower operation.

4.Challenge : Delete from the front

```
void deleteFront(){
        if(n==0){
            cout<<"List is empty\n";
            return;
        }
        Node* temp=head;
        head=head->next;
        delete temp;
        n--;
        cur=head;
        while(cur!=nullptr){
            cout<<cur->value<<"->";
            cur=cur->next;
        }
        cout<<"\n";
    }
```

Explanation : The head pointer is updated to the second node which has become the first node and the old first node is deleted to free its memory and avoid a memory leak.

5. Challenge : Delete from the end

```
void deleteEnd(){
        if(n==0){
        cout<<"No node to del!\n";
            return;
        }
        cur=head;
        while(cur->next->next!=nullptr){
            cur=cur->next;
        }
        delete cur->next;
        cur->next=nullptr;
        n--;
    }
```

Explanation : We have to walk the list until we reach the second last node which is right before the last node by checking cur->next->next != nullptr then delete the last node and make the second last node point to nullptr. This works because checking cur->next->next ensures that the current node is the second last one, as the node after it (the last node) points to nullptr.

6. Challenge : Delete from the middle

```
void deleteMiddle(){
        if(n==0){
            cout<<"No node to del!\n";
            return;
        }
        if(n==1){
            deleteFront();
            return;
        }
        int pos=n/2;
        cur=head;
        for(int i=0;i<pos-1;i++){
            cur=cur->next;
        }
        Node* temp=cur->next;
        cur->next=cur->next->next;
        delete temp;
        n--;
    }
```

Explanation : The arrow that changes is the next pointer of the node before the middle node (stored in cur). Specifically, cur->next is redirected to point to cur->next->next, effectively bypassing the middle node and linking the node before it directly to the node after it. If we forget to free memory using delete temp, the removed node will remain in memory even though it's no longer accessible through the linked list, causing a memory leak.

7.Challenge : Traverse List

```
void traverse(){
        if (n ==0 ){ //
             cout << "(empty!)\n";
             return ;
        }
       cur = head ;
        while (cur != nullptr){
             cout << cur-> value<<"->";
             cur = cur->next;
        }
    }
```

Explanation: Array access using arr[i] is O(1) constant time because elements are stored contiguously in memory, allowing direct calculation of any element's address. Linked list traversal is O(n) linear time because nodes are scattered in memory and connected by pointers, requiring us to start at the head and follow next pointers sequentially until reaching the desired position. This makes arrays faster for accessing elements by index, while linked lists are better for insertions and deletions.

8. Challenge : Swap two nodes

```
void swapTwoNodes(int pos1,int pos2){
          if(pos1>=n || pos2>=n){
               cout<<"Out of range!!\n"; return;
          }
          if(pos1==pos2){
               return;
          }
          if(pos1>pos2){
               swap(pos1,pos2);
          }
          Node* prev1=nullptr,*prev2=nullptr,*node1=head,*node2=head;
          for(int i=0;i<pos1;i++){
          prev1=node1;
          node1=node1->next;
          }
          for(int i=0;i<pos2;i++){
          prev2=node2;
          node2=node2->next;
          }
          if(prev1!=nullptr){
          prev1->next=node2;
        }else{
                    head=node2;
                }
                if(prev2!=nullptr){
                prev2->next=node1;
                }else{
                    head=node1;
                }
                Node* temp=node2->next;
                node2->next=node1->next;
                node1->next=temp;
            }
```

Explanation : It is easier to swap values than to swap links. Swapping values requires only three lines with a temporary variable, while swapping links requires tracking four pointers (prev1, prev2, node1, node2), handling special cases for head nodes, and updating multiple next pointers to avoid breaking the list.

9. Challenge : Search in linked list

```
void searchInLinkList(int val){
        if(n==0){
            cout<<"List is empty\n";
            return;
        }
        cur=head;
        int pos=0;
        while(cur!=nullptr){
        if(cur->value==val){
                cout<<"Value "<<val<<" found at position
"<<pos<<"\n";
                return;
            }
            cur=cur->next;
            pos++;
        }
        cout<<"Value "<<val<<" not found in the list\n";
    }
```

Explanation : The similarity between search in linkedlist and search in array is that both use sequential traversal, checking each element until the target is found, giving them O(n) time complexity. Array linear search is faster because arrays have contiguous memory that provides better cache performance, while linked lists require slower pointer dereferencing at each step and have scattered memory locations.

10. Challenge : Compare with array

Discussion:

Insert Operation

Arrays require O(n) time for insertion because adding an element in the middle forces all subsequent elements to shift over. Linked lists achieve O(1) time once you have the insertion point, since you only update pointers without moving existing elements.

Delete Operation

Arrays need O(n) time for deletion as removing an element requires shifting remaining elements forward to fill the gap. Linked lists perform deletion in O(1) time when you have the node reference by simply updating pointers to bypass the deleted node.

Access Operation

Arrays excel with O(1) access to any element by index through direct memory calculation. Linked lists require O(n) access since you must traverse from the head, following pointers sequentially to reach your target.

When Linked Lists Are Clearly Better

Linked lists are better than arrays when you need frequent insertions/deletions at known positions, especially at the beginning or middle (O(1) vs O(n) for arrays). They're ideal for unpredictable or constantly changing sizes since they grow dynamically without costly reallocation. They also handle fragmented memory better, allocating small chunks anywhere instead of needing large contiguous blocks like arrays.

Reflection Prompts

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

Answer: Insert and delete operations are O(1) in linked lists (when you have the position) but O(n) in arrays.

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

Answer: Access/lookup by index is clearly faster in arrays (O(1)) than in linked lists (O(n)). Arrays allow direct memory calculation to jump to any position instantly, while linked lists require traversing from the head node by node to reach a specific position.

3. Why must we manage memory carefully in linked lists?

Answer: In linked lists, each node is dynamically allocated and must be manually freed when deleted to prevent memory leaks. Unlike arrays where the entire block is deallocated at once, linked lists can leave orphaned nodes in memory if pointers are lost or nodes aren't properly freed, wasting memory that can't be reclaimed.

4. What does the head pointer represent?

Answer: The head pointer represents the starting point or first node of the linked list. It's the entry point that allows access to the entire list, as all other nodes are reached by following pointers from the head.

5. What happens if we lose the head pointer?

Answer: The entire linked list becomes inaccessible and lost. Without the head pointer, there's no way to reach any nodes in the list since linked lists can only be traversed from the beginning. All the memory used by those nodes becomes leaked and unusable.

Scenario Analysis: Choose Array or Linked List

Read the following scenarios and decide whether an **array** or a **linked list** is a better fit. Justify your choice.

1. **Real-time scoreboard** where new scores are always added at the **end** and sometimes removed from the **front**.
2. **Undo/Redo feature in a text editor**, where operations are frequently added and removed at the **front**.
3. **Music playlist** that lets users add and remove songs anywhere in the list.
4. **Large dataset search** where random access by index is needed often.
5. **Simulation of a queue at a bank**, where customers join at the end and leave at the front.
6. **Inventory system** where you always know the item's index and need quick lookups.
7. **Polynomial addition program** where terms are inserted and deleted dynamically.
8. **Student roll-call system** where the order is fixed and access by index is frequent.

1. Real-time scoreboard: **Linked List**
2. Undo/Redo feature in a text editor: **Linked List**
3. Music playlist: **Linked List.**
4. Large dataset search: **Array.**
5. Simulation of a queue at a bank: **Linked List.**
6. Inventory system: **Array.**
7. Polynomial addition program: **Linked List.**: **Array.**
8. Student roll-call system: **Array.**