
-Challenge 1:

```
void insertfront(int val){
    Node* newNode = new Node;
    newNode->val = val;
    newNode->next = head;
    head = newNode;
    n++;
}
```

explanation :
-we take the value from parameter and then create a new Node with it
-we assign the new Node's next to head
-lastly we assign head to point to new Node since it is now the first of the list

Q: Complexity : O(1)

Discussion :
-Compare to inserting at the beginning of the array this is much more efficient since We do not need to shift any elements.
-To insert ann elements in the beginning of the linkedlist we just have to create a newnode and then point that new Node to the first node of the linked list.

-Challenge 2:

```
void insertattheend(int value){
    Node *newNode = new Node;
    newNode->val = value;
    newNode->next = nullptr;
    if (head == nullptr){
        head= newNode;
    } else {
        Node *cur = head;
        while (cur->next) cur = cur->next;
        cur->next = newNode;
    }
    n++;
}
```

explanation :
-we take value from the parameter and then create a new Node with it
-we point the new Node's next to null becuase it is going to be the last Node of the list
-we check if the head is null then the list is empty so the new Node will be the first node of the list.
-if the list is not empty then we traverse the list until the end and then point the last node's next to our new node.

Q: Complexity : O(N)

Discussion :
-In this Case we have to traverse the whole list because we did not

keep track of the tail.
-if we are using array we do not have to traverse the whole list to insert at the end, Therefore it is more efficient to work with array in this case but There is a limitation of the array being full while linkedlist does not have that problem.but if we keep track of tail then inserting at the end of linked list is O(1) thus better than array.

-Challenge 3:

```
void insert(int target, int value){
    if (target > n){
        cout<<"index out of range"<<endl;
        return;
    }
    if (target == n){
        insertattheend(value);
        return;
    }
    if (target < 0){
        cout<<"the position cant be negative."<<endl;
        return;
    }
    if( target == 0 ){
        insertfront(value);
        return;
    }
    Node* newNode = new Node;
    newNode->val = value;
    Node* cur = head;
    for (int i=0;i<target-1;i++) cur = cur->next;
    newNode->next = cur->next;
    cur->next = newNode;
    n++;
}
```

explaitonn :
-First we create new node with the value form parameter
-and then we traverse the list until we see the target node
-then we point our new Node's next to the next node and point our target node's next to our new node
-edge case :
    +the target node is out of range
    +if the target node is at the end then it will be just inserting at the end
    +if the target is negative number which is impossible
    +if the target is at the beginning then it will be just inserting at the front

Complexity : O(N)

Discussion :
-We have to point the new Node to the next node and then re-set the target node to point to the new Node. so there are 2 pointer that

needed to be changed.
-Compare to array, inserting in middle of linked list is a bit more efficient while it both take roughly O(n) time but when traverse the linked list we only need to read the memory but shifting elements in array require to read memory and write memory, Therefore linked list is more efficient.

-Challenge 4

```
void deletefront(){
    if (head == nullptr){
        cout<<"the list is empty."<<endl;
        return;
    }
    Node* temp = head;
    head = head->next;
    delete temp;
    n--;
}
```

explaitonn :
-we create a temporary node to point at the first node
-then we move head to the next node
-now we delete the old head by deleting the temporary node
-edge case :
    + if head is null it mean the list empty there's nothing to delete

Complexity : O(1)

Discussion :
-the head pointer is set to point to the next node.
-We created a temporary pointer to point at the node and after moving head to the next node we delete the old node.

-Challenge 5

```
void deleteback(){
    if (head == nullptr){
        cout<<"the list is empty."<<endl;
        return;
    }
    if(n == 1){
        delete head;
        head = nullptr;
        n--;
        return;
    }
    Node* cur = head;
    while(cur->next->next) cur = cur->next;
    Node* temp = cur->next;
    cur->next = nullptr;
    delete temp;
    n--;
}
```

explaiton :
- We traverse the list until we are one node behind the last node of the list
- then we create a temporary node to point to the last node of the list
- then we point the second last node of the list to nullptr indicating the end of the linked list
- lastly we delete the old node
- edge case :
    + if the head is null then it mean the list is empty so there is nothing to delete
    + if the list only has one node then we just delete the one node that in the list.

Complexity : O(N)

Discussion :
-We find the node before the last node by checking the next two value (cur->next->next) if it's nullptr.

Challenge 6:

```
void deletemiddle(int target){
    if (target == 0){
        deletefront();
        return;
    }
    if (target < 0 || target >= n){
        cout<<"target is out of range."<<endl;
        return;
    }
    Node* cur = head;
    for(int i=0;i<target-1;i++) cur = cur->next;
    if(cur->next == nullptr){
        deleteback();
        return;
    }
    Node* temp = cur->next;
    cur->next = cur->next->next;
    delete temp;
}
```

explaiton :
-Frist we traverse the list until we are one behind our target node
-then we have to create a temporary node to store the node that we are going to delete later
-then we point the one node behind target node's next to the next's next node passing the target node
-lastly we have to delete the target node using the temporary
-edge case :
    + the target is at the beginning of the list then we can just delete the front

    + if the target is negative or bigger than n then it is out of range
    + if the target is the last node of the list then we can just delete the back node

Complexity : O(N)

Discussion :
-We need to change the pointer of the node before the one that we want to delete to point to the node after the one we want to delete.
-If we forget to free the memory then we will lose access to the node and it will stay in the memory.

-Challenge 7
```
void _traverse(){
    cur = head;
    while (cur){
        cout<<cur->val<<"->";
        cur = cur->next;
    }
    cout<<endl;
}
```

explaiton :
-we set cur to head
-using while loop we print out the value of each node and then go to the next node until the node is nullptr

Complexity : O(N)

Discussion : -Traversing the linked list mean that we have to visit every node of the linkedlist, whereas accessing a[i] do not need to visit every elements.

-Challenge 8
```
void swapNode(int node1, int node2) {
    if (node1 == node2) return;

    if (node1 > node2) swap(node1, node2);

    Node *prev1 = nullptr, *prev2 = nullptr;
    Node *n1 = head, *n2 = head;

    for (int i = 0; i < node2; ++i) {
        if (i < node1) {
            prev1 = n1;
            n1 = n1->next;
        }
        prev2 = n2;
        n2 = n2->next;
    }
    if (!n1 || !n2) return;
    if (n1->next == n2) {
        if (prev1) prev1->next = n2;
        else head = n2;

        n1->next = n2->next;
        n2->next = n1;
    } else {
        Node* temp = n2->next;

        if (prev1) prev1->next = n2;
        else head = n2;

        if (prev2) prev2->next = n1;
        else head = n1;

        n2->next = n1->next;
        n1->next = temp;
    }
}
```

explaiton :
-We init n1 and n2 to store the node that we want to swap and prev1 and prev2 to store the previous node of n1 and n2
-we do not need to store the next node since if we store the current node we can use .next to access the next node
-we check if the n1 and n2 are next to each other if yes then we check if prev1 is nullptr if not the we set prev1 point to n2 if yes we set the head to prev1 and then we set n1 to point to n2's next and set n2 to point to n1
-if the nodes are not next to each other then we create a temporary node to point to n2's if n1 is not head then we can set prev1's next which is n1 to point to n2 if it is head then we set head to point to node2
-(same thing goes to prev2)
-then we set the n2's next to n1'next and n1's next to temporary node.
-edge case :
+ if node1 and node2 are the same node then we return as there is nothing to swap
+ to ensure that node1 is always come before node2 so have to check and then swap it if it's in the wrong order

-Complexity : O(N)

-Discussion :
It is easier to just swap the value instead of the whole node because swapping the link(node) is more complicated and more dangerous since we can run into bug especially segmentation fault.

-Challenge 9

```
int search(int target){
    cur = head;
    int location = 1;
    while(cur){
        if (cur->val == target){
            return location;
        }
        location++;
        cur = cur->next;
    }
    cout<<"value not found."<<endl;
    return 0;
}
```

explanation :
-we take target as parameter. target here refer to the value not the position
-we traverse the list until we find a node with a value equal to target
-then we return the location of the target

Complexity : O(N)

Discussion :
-this is similar to linear search in array since we have to check every node in the list.
-for random access array is faster than linked list because if we know the location of the element we can access in O(1) time but we have traverse the linked list that cost O(N).

Challenge 10:

Operation	Scenario	Linked List	Array
Insert	At Front	O(1)	O(n)
Insert	At End	O(n)	O(1)
Insert	In Middle	O(n)	O(n)
Delete	From Front	O(1)	O(n)
Delete	From End	O(n)	O(1)
Delete	From Middle	O(n)	O(n)
Search	By Value	O(n)	O(n)
Access	By Index	O(n)	O(1)
Swap	Two Elements	O(n)	O(1)

Discussion :
-Inserting at the front is better with linked list since we don't need to shift element.
-Delete from the front : linked list did better since we only need to swap pointer and no shifting.
-Insert at the end : In this case linked list is a slower than array but linked list eliminate the issue of limited size.

Reflection :
1. operation that linkedlist has O(1) and array has O(N) are insert at the front and delete at the end.
2. accessing value in array is clearly faster than accessing value in linkedlist.
3. because linkedlist is built from value and address if don't handle them carefully we can cause dangling pointer,memory leakded and we could also lose addres to node and left it in the memory.
4. The head pointer represent where the linked list start.
5. if we lose the head pointer we will lose the whole linked list.

Scenario analysis:

1. Real-time scoreboard :
-linked list : because even tho array is more efficient to add in the end but it has limited size but linkedlist has no limited size and sometime we remove front which is more efficient with linkedlist.
2. Undo/Redo feature in a text editor:
-linked list : linked list is more efficient when we add and remove stuff from the front.
3. Music Playlist:
-Linked list : remove and add song anywhere in the list is faster in linked list and linked list also eliminate the issue of limited size.
4. Large data search :
-Array : random access in faster in array and also we can preform a binary search in a sorted list which is fast.
5. Simulation of a queue at a bank :
-linked list : join at the end and leave from the front is faster in linked list.
6. inventory system :
-Array : array has faster access time.
7. Polynomial addition program :
-linked list : insetion and delete anywhere is faster with linked list.
8. Student roll-call system :
-array : acces by index is faster with array and the order is fixed. limitation of size is not a problem since we know how many student are in one class.