DELITION & INSERTION

✡ Type 1 : Deleting the Head

We will move the head to the next element and delete the element which was previously the head from memory.

```
Node * removeHead(Node * head) {
    if(head == NULL) {
        return head;
    }
    Node * temp = head;
    head = head -> next;
    delete temp;
    return head;
}
```

✡ Type 2 : Deleting the tail

We stop before the last element and turn its next to a null pointer.

```
Node * removeTail(Node * head) {
    if(head == NULL || head -> next == NULL){
        return NULL;
    }
    Node * mover = head;
    while(mover -> next -> next != nullptr){
```

```
        mover = mover -> next ;
    }
    delete mover -> next ;
    mover -> next = nullptr ;
    return head ;
}
```

✯ Type 3 : Deleting K^{th} Element

If K = 1, we delete the head of the linked list. If the list is empty we return null. In Linked lists questions our focus should be to satisfy few edge cases first instead of creating a full one size fits all logic.

```
Node * deleteK(Node * head, int K) {
    if (head == nullptr) {
        return nullptr ;
    }
    if (K == 1) {
        Node * temp = head ;
        head = head -> next ;
        delete temp ;
        return head ;
    }
    int c = 0 ;
    Node * temp = head ;
    Node * prev = nullptr ;
    while (temp != nullptr) {
        c++ ;
        if (c == K) {
```

```
            prev -> next = prev -> next -> next ;
            break ;
        }
        prev = temp ;
        temp = temp -> next ;
    }
    return head ;
}
```

☆ Type 4 : Delete by value

We make a small change to the above code. Instead of checking for an iteration counter, we just check if our current element is equal to the required one, in which case we perform the deletion

```
Node * removeEl(Node * head, int K) {
    if (head == nullptr) {
        return nullptr ;
    }
    int c = 0 ;
    Node * temp = head ;
    Node * prev = nullptr ;
    while (temp != nullptr) {
        if (temp -> data == K) {
            prev -> next = prev -> next -> next ;
            break ;
        }
        prev = temp ;
        temp = temp -> next ;
    }
```

```
    return head;
}
```

✡ Type 5 : Insert at head

We will just create a new node assign its value to the required value, assign our previous head to its next and just return it as the new head

```
Node* insertAtHead(Node* head, int K) {
    Node* newNode = new Node(K, head);
    return newNode;
}
```

✡ Type 6 : Insert at tail

We will just create a new Node which has our required value and its next points to null. We traverse through the entire Linked list, find the end and make the end's next attach to our new node

```
Node* insertAtTail(Node* head, int K) {
    Node* temp = new Node(K, nullptr);
    Node* mover = head;
    while(mover != nullptr) {
        if(mover->next == nullptr) {
            mover->next = temp;
            break;
        } else {
```

```
            mover = mover -> next ;
        }
    }
    return head ;
}
```

☆ Type 7 : Insert Element at given Index

We will create a new Node in which the data is assigned to our desired value. We will traverse through the entire LinkedList, find our index, make the previous' next point to our new node and new node's next to actual next.

```
Node * insertAt(Node * head, int K, int i){
    int c = 0 ;
    if (head == nullptr) {
        return nullptr ;
    }
    Node * temp = new Node(K, nullptr) ;
    if (i == 1) {
        temp -> next = head ;
        return temp ;
    }
    Node * mover = head ;
    Node * prev = nullptr ;
    while (mover != nullptr) {
        c++ ;
        if (c == i) {
            prev -> next = temp ;
```

```
            temp -> next = mover ;
            break ;
        }
        prev = mover ;
        mover = mover -> next ;
    }
    return head ;
}
```