DOUBLY LINKED LIST

In a singly linked list, we had a pointer which was only pointing to the next node. Traversal was only possible in forward direction. But in a doubly linked list, we will also store reference to the previous element. So now both direction traversal is possible.

```
class Node {
    public :
    int data ;
    Node * next ;
    Node * prev ;

    Node(int d, Node * n, Node * p) {
        next = n ;
        prev = p ;
        data = d ;
    }

    Node(int d) {
        data = d ;
        prev = nullptr ;
        next = nullptr ;
    }
}
```

Converting an array to a DLL is very similar to converting it into

an LL.

```
Node * head = new Node(arr[0], nullptr,
                                nullptr);
Node * prev = head;
for (i = 1 → N-1) {
    Node * temp = new Node(arr[i], nullptr,
                           prev);
    prev → next = temp;
    prev = temp;
}
return head;
```

Deleting a Node in a doubly linked list involves a similar procedure as it was in a single linked list

☆ Type 1 : Deleting the head

If list is empty, or if it only has one element, we return null. Otherwise we just segregate the head, remove it from memory and return next element as new head.

```
Node * deleteHead(Node * head) {
    Node * newHead = head → next;
    newHead → prev = nullptr;
    head → next = nullptr;
    delete head;
    return newHead;
}
```

✯ Type 2 : Deleting the tail

Apart from the edge cases, to delete the tail, we just iterate to the end, segregate the last node and delete it from the memory

```
Node * deleteTail(Node * head) {
    Node * mover = head;
    while(mover -> next != nullptr) {
        mover = mover -> next;
    }
    mover -> prev -> next = nullptr;
    delete mover;
    return head;
}
```

✯ Type 3 : Delete K^{th} Element

Very similar to previous, we just iterate to K^{th} position, disconnect it from the list and reconnect its adjacents.

```
Node * deleteKth(Node * head, int K) {
    int c = 0;
    Node * mover = head;
    while(mover != nullptr) {
        c += 1;
        if(c == K) {
            mover -> next -> back = mover
                                -> back;
```

```
            mover -> back -> next = mover ->
                                    next;

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

☆ Type 4 : Delete Node by Value

Here, at each step in iteration, instead of checking for index, we check for value match.

```
Node * deleteValue(Node * head, int K) {
    Node * mover = head;
    while (mover != nullptr) {
        if (mover -> data == K) {
            mover -> next -> back = mover
                                    -> back;
            mover -> back -> next = mover ->
                                    next;

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

Inserting a node into the doubly

Linked list process is same as the singly linked list

✡ Type 1 : Inserting at Head

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

✡ Type 2 : Inserting at Tail

```
Node* insertAtTail(Node* head, int K) {
    Node* newNode = new Node(K);
    Node* mover = head;
    while(mover->next != nullptr) {
        mover = mover->next;
    }
    newNode->back = mover;
    mover->next = newNode;
    return head;
}
```

✡ Type 3 : Insert at K^{th} Position

```
Node* insertAtK(Node* head, int K,
                int i){
    int c = 0;
    Node* newNode = new Node(i);
    Node* mover = head;
    if(K == 1) {
```

```
        newNode -> next = head ;
        head -> back = newNode ;
        return newNode ;
    }
    while (mover != nullptr) {
        c += 1 ;
        if (c == K) {
            mover -> next -> back = newNode ;
            mover -> next = newNode ;
            newNode -> back = mover ;
            break ;
        }
        mover = mover -> next ;
    }
    return head ;
}
```