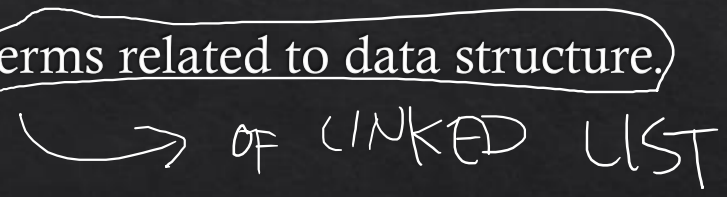
Data Structures & Algorithm

Linked List

Data Structure and Algorithms - Linked List

This chapter explains the basic terms related to data structure.

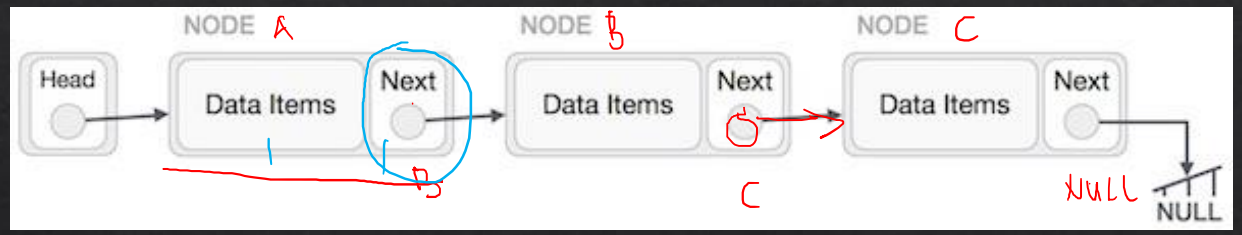


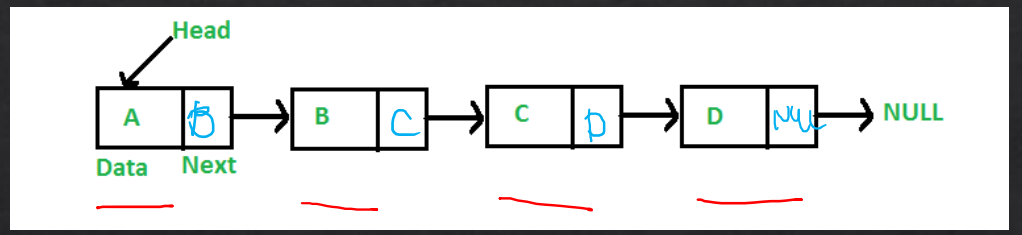
Linked List

- A linked list is a sequence of data structures, which are connected together via links.
- Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.
  - **Link** − Each link of a linked list can store a data called an element.
  - **Next** − Each link of a linked list contains a link to the next link called Next.
  - **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

- Linked list can be visualized as a chain of nodes, where every node points to the next node.

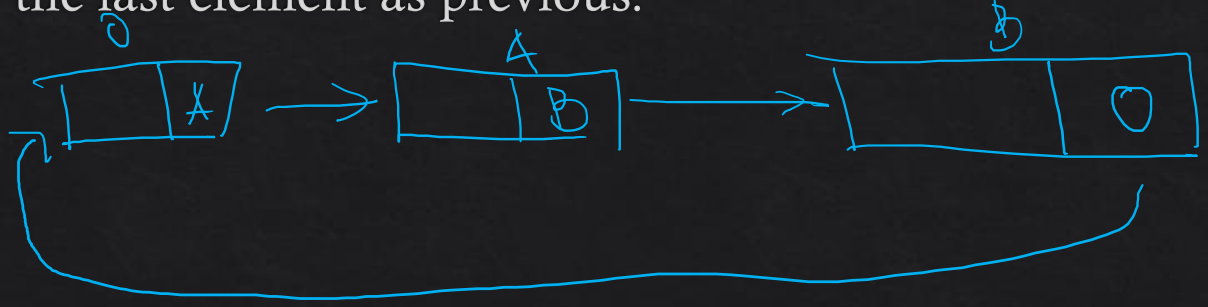




- As per the above illustration, following are the important points to be considered.
- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

- Following are the various types of linked list.
  - Simple Linked List – Item navigation is forward only.
  - Doubly Linked List – Items can be navigated forward and backward.
  - Circular Linked List – Last item contains link of the first element as next and the first element has a link to the last element as previous.

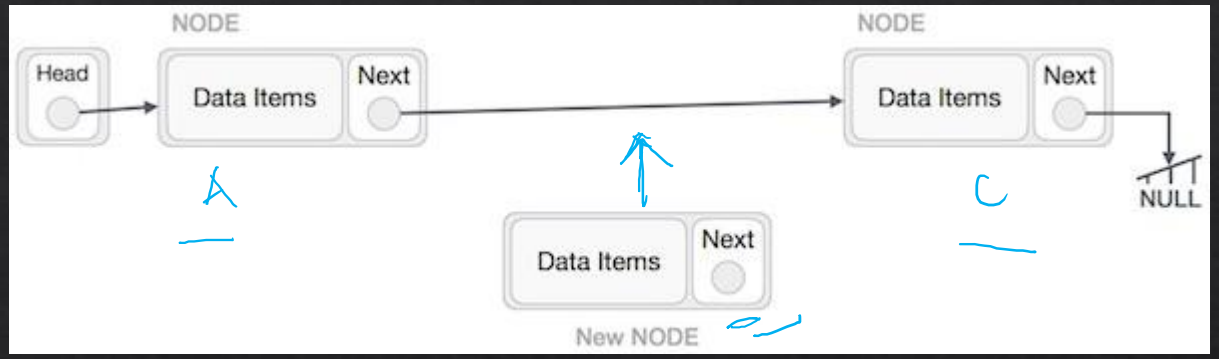


Basic Operations

- Following are the basic operations supported by a list.
  - Insertion – Adds an element at the beginning of the list.
  - Deletion – Deletes an element at the beginning of the list.
  - Display – Displays the complete list.
  - Search – Searches an element using the given key.
  - Delete – Deletes an element using the given key.

Insertion Operation

- Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.

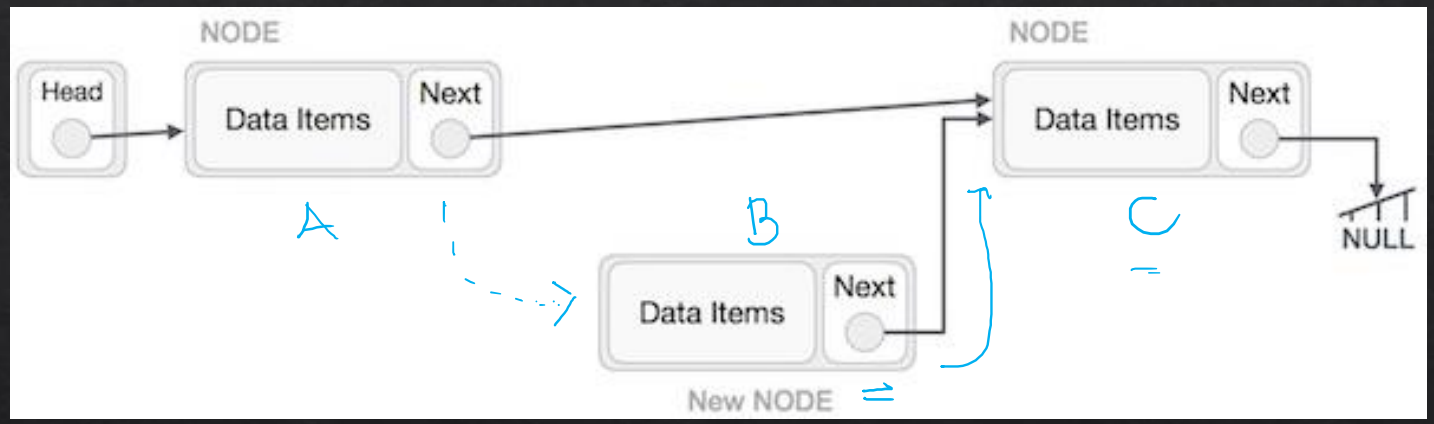


- Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C –

Insertion Operation

NewNode.next –> RightNode;

- It should look like this –



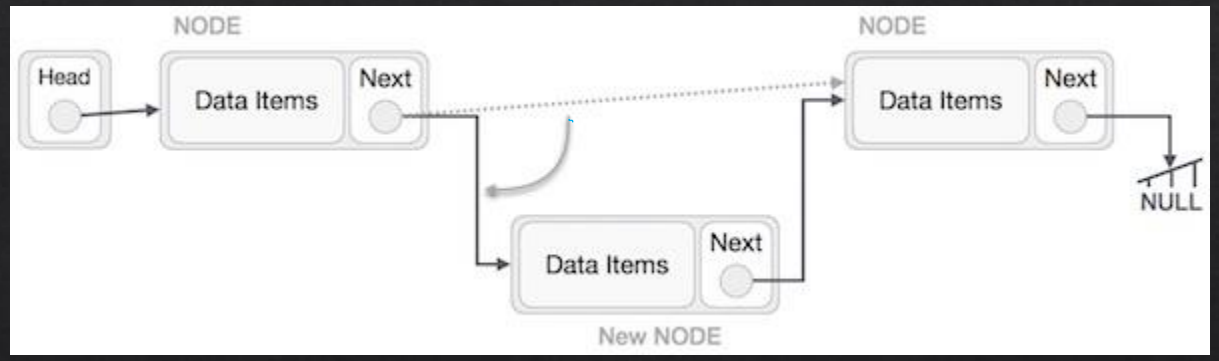
- Now, the next node at the left should point to the new node.

LeftNode.next –> NewNode;

Insertion Operation

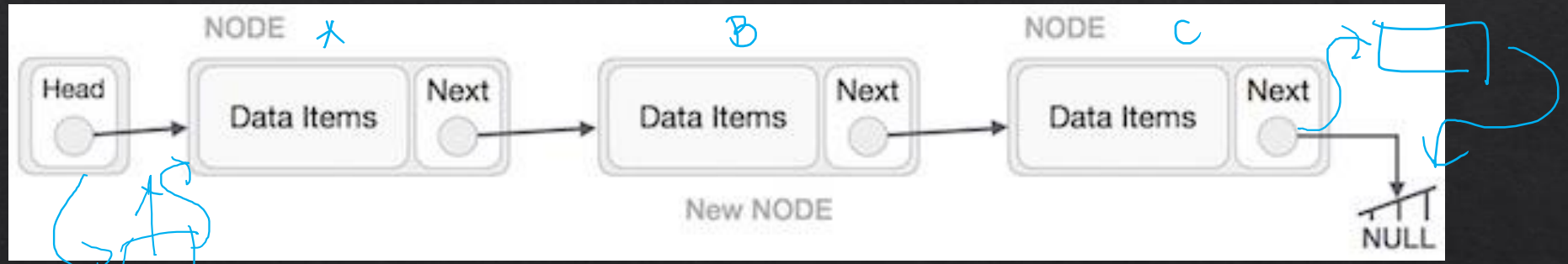
- Now, the next node at the left should point to the new node.

  LeftNode.next –> NewNode;



Insertion Operation

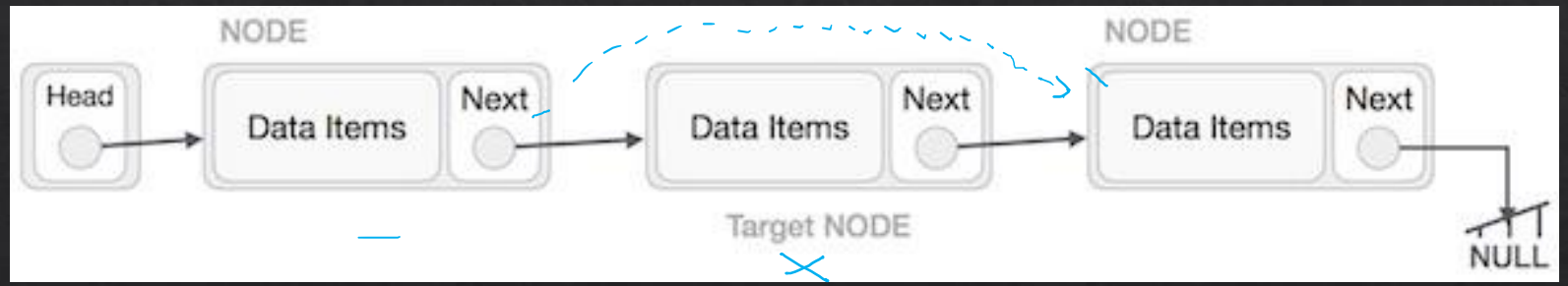
- This will put the new node in the middle of the two. The new list should look like this –



- Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

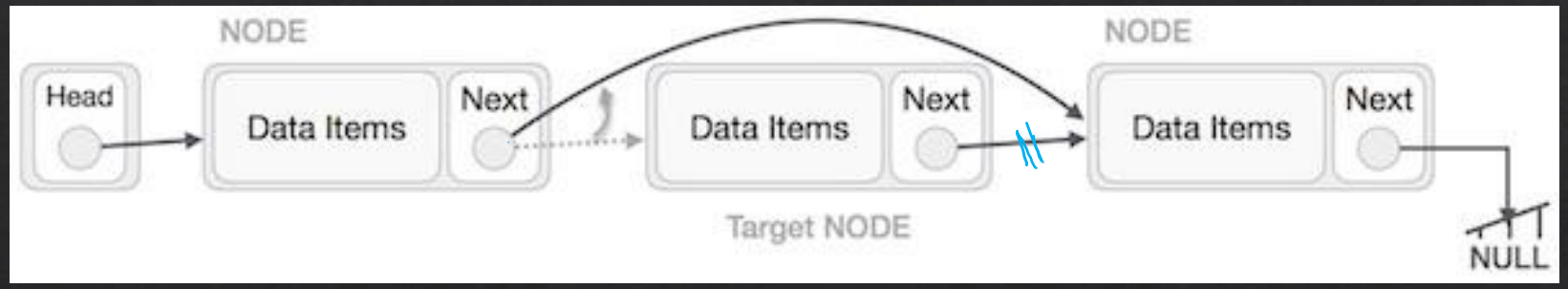
- Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



- The left (previous) node of the target node now should point to the next node of the target node −

```
LeftNode.next -> TargetNode.next;
```

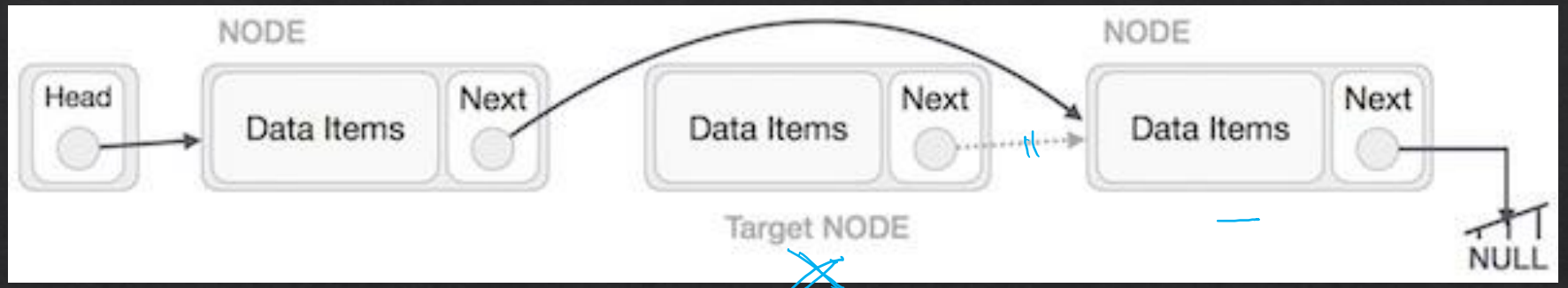
Deletion Operation



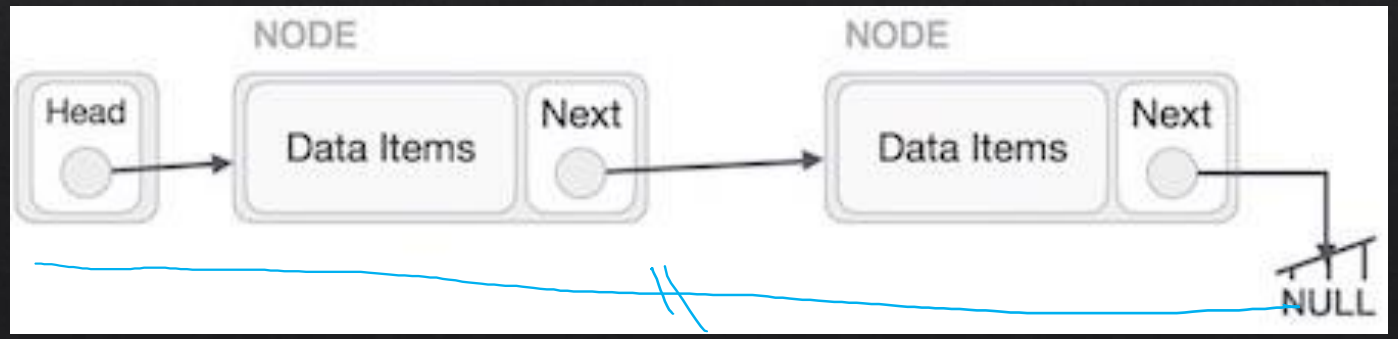
- This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```

Deletion Operation

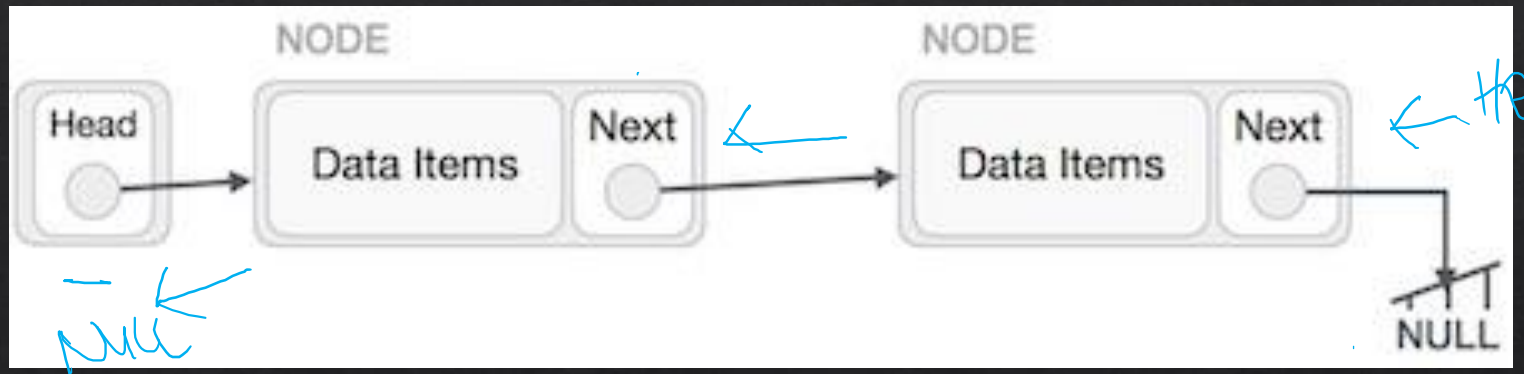


- We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



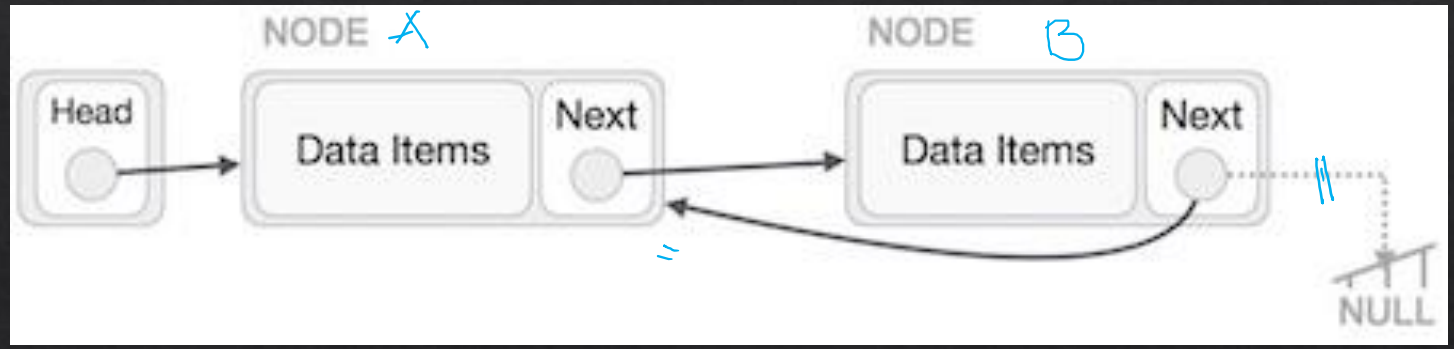
Reverse Operation

- This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



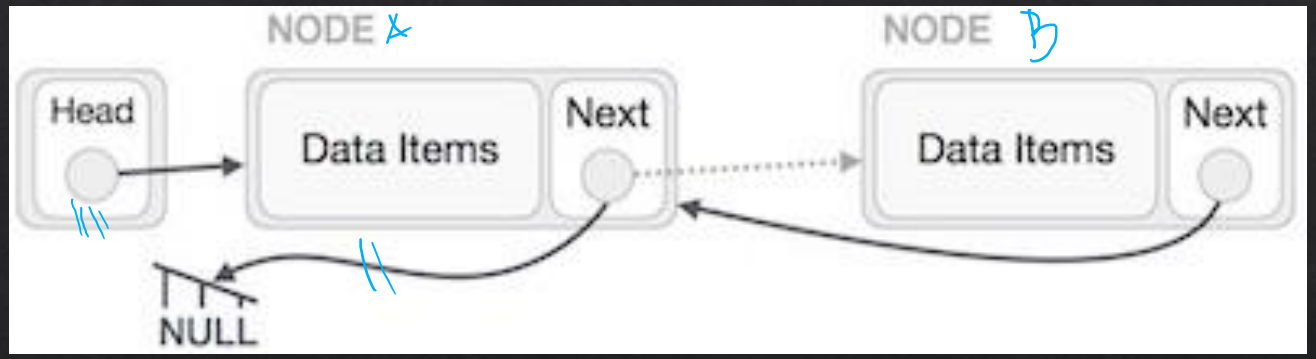
Reverse Operation

- First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



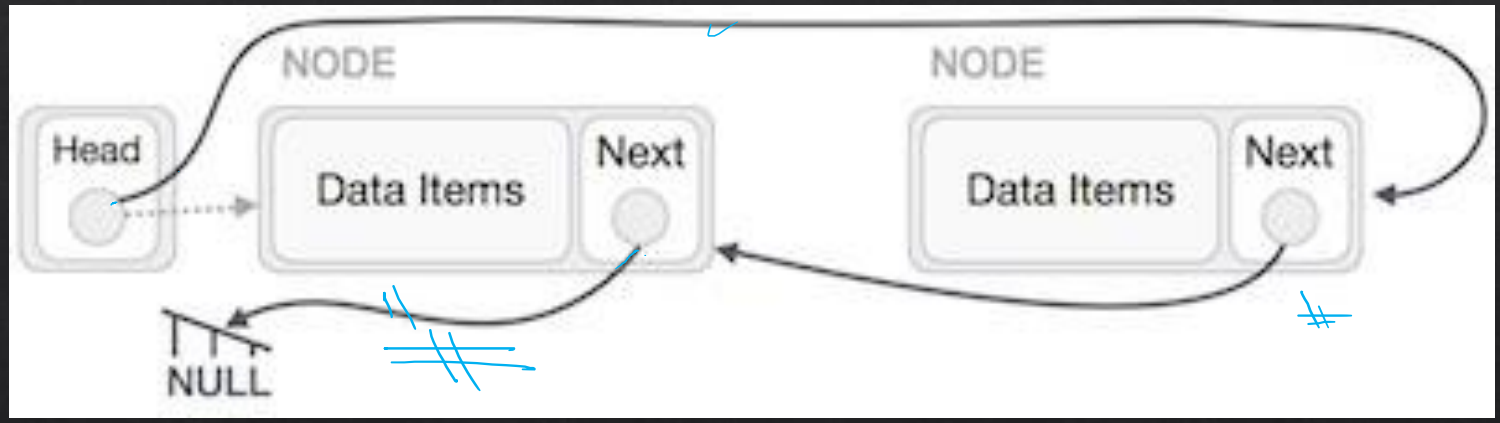
Reverse Operation

- We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



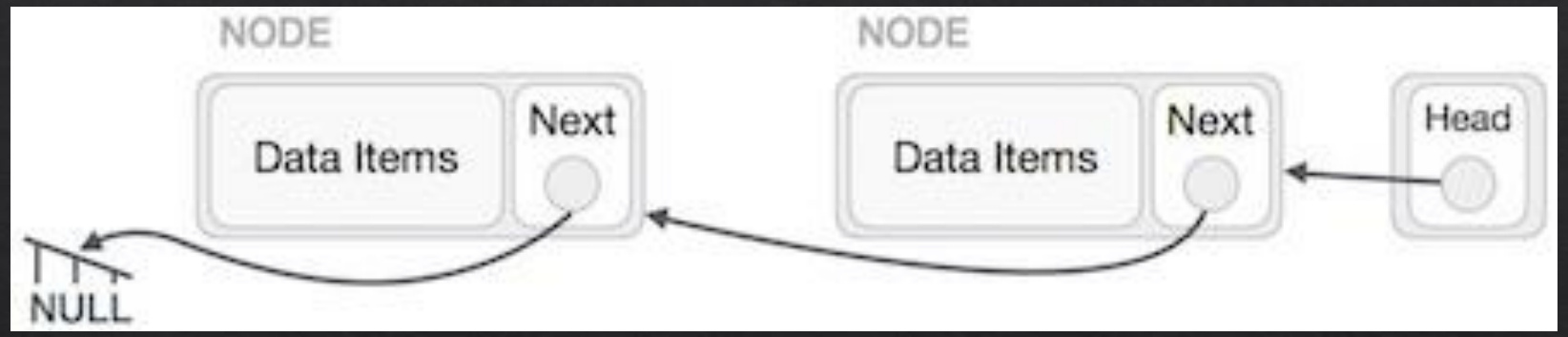
Reverse Operation

- Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



Reverse Operation

- We'll make the head node point to the new first node by using the temp node.



- The linked list is now reversed.

Why Linked List?

- Arrays can be used to store linear data of similar types, but arrays have the following limitations.
  - The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
  - Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

- For example, in a system, if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

Why Linked List?

id[] = [1000, 1010, 1050, 2000, 2040].

- And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays

- Dynamic size
- Ease of insertion/deletion

Drawbacks:

- Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
- Extra memory space for a pointer is required with each element of the list.
- Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.

Each node in a list consists of at least two parts:

- data
- Pointer (Or Reference) to the next node

- In C, we can represent a node using structures. In the next slide, is an example of a linked list node with integer data.
- In Java or C#, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

Representation

```c
// A linked list node in C
struct Node {
    int data;
    struct Node* next;
};
```

```cpp
// A linked list node in C++
class Node {
public:
    int data;
    Node* next;
};
```

C/C++ Example of Linked List

- Will discuss on codeblocks