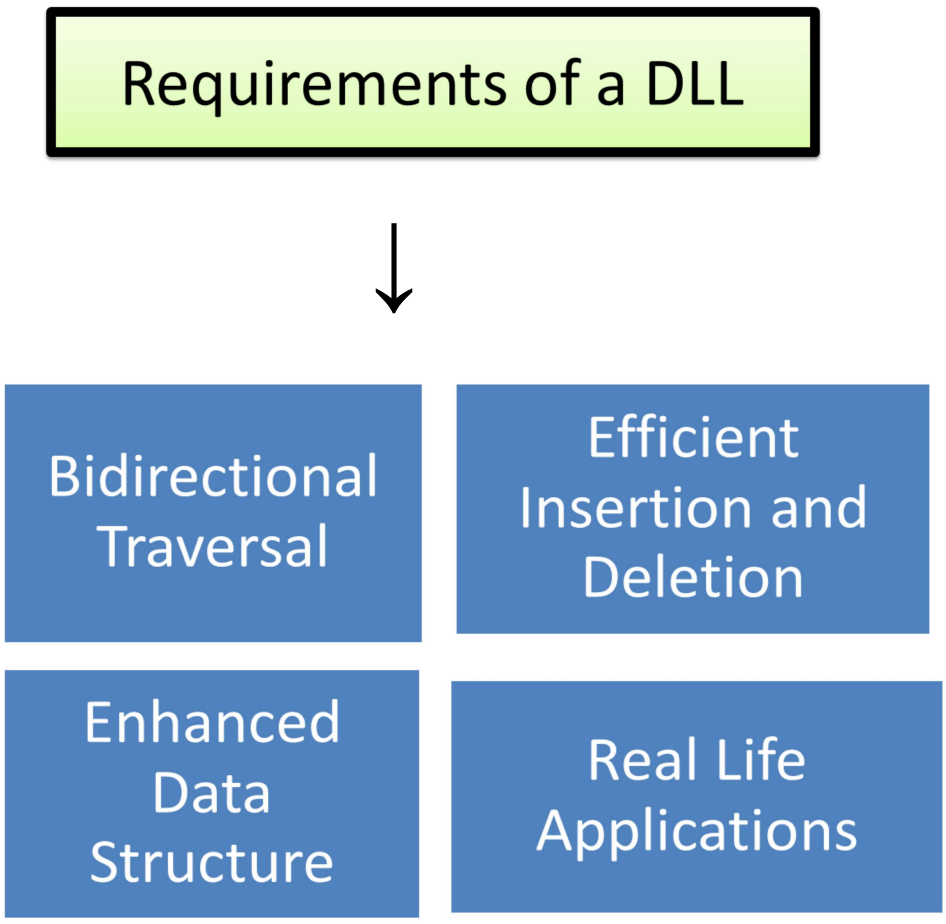
DOUBLY LINKED LIST

- **Why do you need a Doubly Linked list ?**

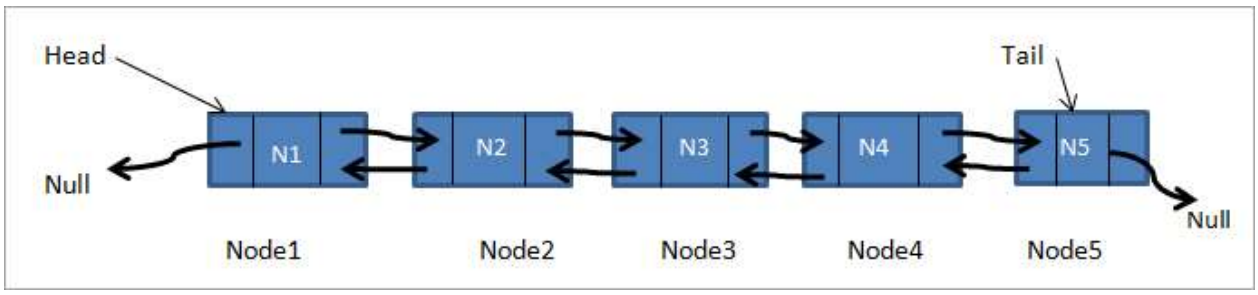
Doubly Linked Lists (DLLs) are an essential data structure that provides significant advantages over other types of lists. So, we will explore the reasons why DLLs are necessary and how they overcome limitations found in other list structures. Recognizing the significance and practicality of DLLs in various applications is crucial for understanding their importance and value in the field of data structures.



The following reasons are :

1) <u>Bidirectional Traversal</u>

One of the primary reasons for using DLLs is their ability to support bidirectional traversal. Unlike singly linked lists, where traversal is limited to a single direction (from the head to the tail), DLLs allow traversing both forwards and backward through the list. This bidirectional traversal enables more efficient and flexible operations on the data.

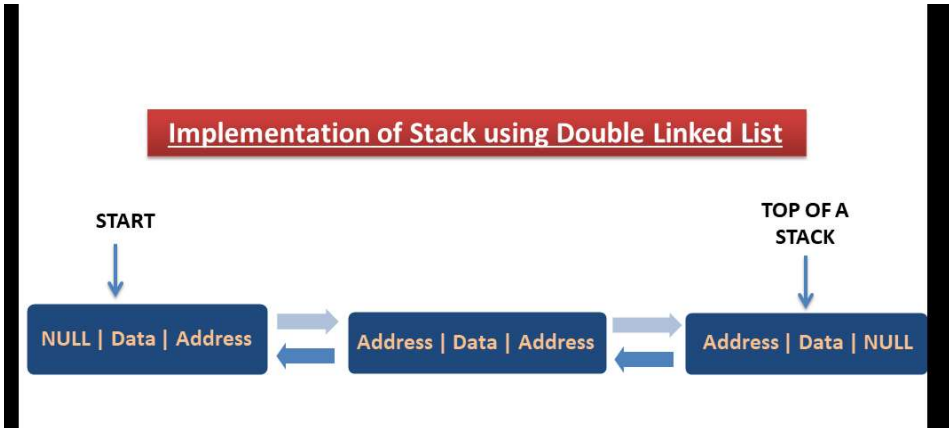


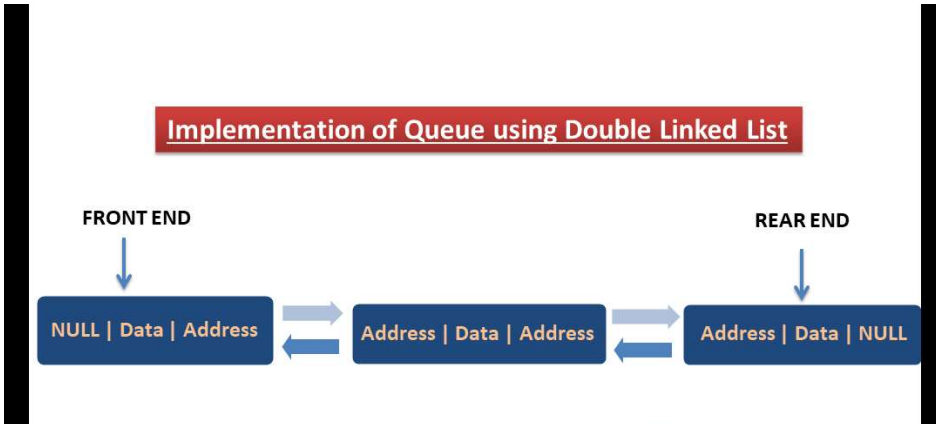
2) <u>Efficient Insertion and Deletion</u>

DLLs stand out in insertion and deletion operations compared to other list structures. In a DLL, inserting a new node or deleting an existing node at any position within the list is relatively straightforward. The presence of both previous and next pointers in each node allows for easy linking of nodes, making these operations efficient in terms of time complexity.

3) <u>Enhanced Data Structure</u>

DLLs serve as the foundation for implementing more complex data structures such as stacks and queues. By combining the bidirectional traversal and efficient insertion and deletion capabilities of DLLs, it becomes possible to design and implement versatile data structures that meet specific requirements.





4) Real Life Applications

DLLs find extensive use in various real-life applications, showcasing their practicality and importance. Some notable examples include:

A. Browser History: DLLs enable users to navigate backward and forward through web pages they have visited, providing a seamless browsing experience.

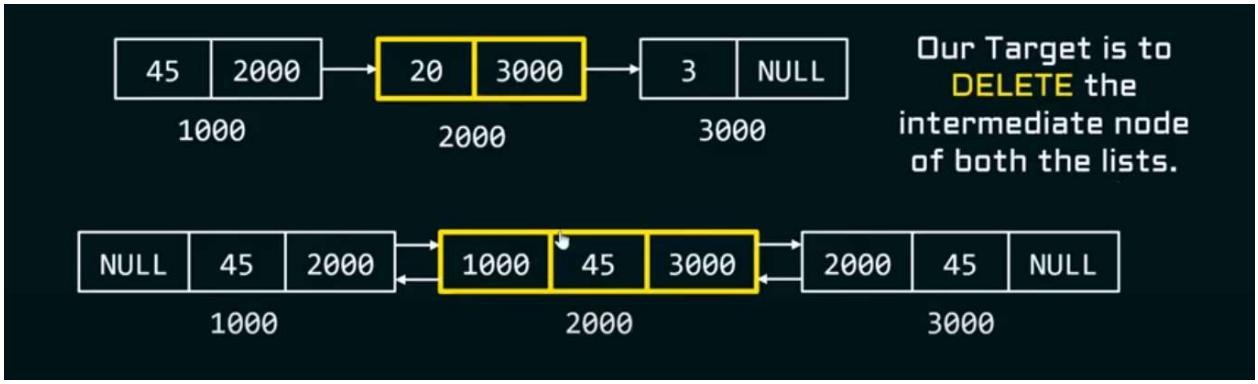
B. Undo/Redo Operations: DLLs facilitate undoing or redoing actions in software applications by storing the state of each action in separate nodes.

C. Music Player Playlist Management: DLLs allow users to manage playlists efficiently. The bidirectional traversal enables reordering songs in the playlist.

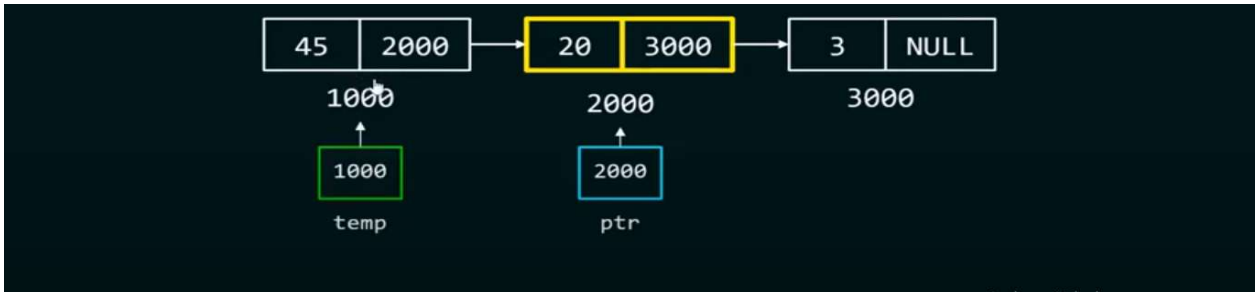
Advantages over Singly Linked List

A doubly linked list is a variation of a singly linked list where each node is also pointing to its previous node. This means we cannot go back to the previous node from the current node.
Thus, DLL have advantages over Singly Linked Lists (SLL) in terms of bidirectional traversal, efficient insertion and deletion, and simplified removal of nodes. These benefits make DLLs a preferred choice for flexible and efficient data manipulation.

Let's understand with the help of a simple example :



In the case of a Singly Linked List , it is important to keep one more pointer pointing to the node just before the node we want to delete. It is shown below.



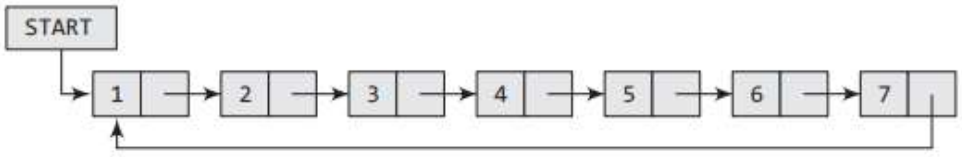
In the case of a Doubly Linked List, deleting the intermediate node is very easy. There is no need to keep an extra pointer to delete the particular node. We can simply move backward / forward to delete a node.

Thus from this we can conclude that insertion or deletion of a node in between is much simpler in case of a Doubly Linked list as compared to the Singly Linked List.

Alternative Techniques (Other than Doubly linked List)

1) Circular Linked List

Circular Linked Lists are a variant of linked lists where the last node points back to the first node, creating a circular structure. While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node we started. Thus, a circular linked list has no beginning and no ending.



Properties of a Circular Linked List

1) Circular Structure : Unlike traditional linked lists, Circular Linked Lists form a closed loop, This connectivity allows for seamless traversal from any node in the list.

2) No Null Pointers : In Circular Linked Lists, there are no null pointers as every node is connected to another node in the list. This eliminates the need for special handling of null pointers and simplifies the traversal logic.

3) Dynamic Size : Circular Linked Lists can dynamically grow and shrink as nodes are inserted or deleted. New nodes can be added anywhere in the list, and existing nodes can be removed, making it a flexible data structure.

Think it out!

Circular linked lists are widely used in operating systems for task maintenance. When we are surfing the Internet, we can use the Back button and the Forward button to move to the previous pages that we have already visited.

How is this done? The answer is simple.

A circular linked list is used to maintain the sequence of the Web pages visited. Traversing this circular linked list either in forward or backward direction helps to revisit the pages again using Back and Forward buttons.

2) Doubly Circular Linked List

Doubly Circular Linked List is similar to the doubly linked list except that the last node of the doubly circular linked list points to the first node and the first node of the list points to the last node.

Properties of a Doubly Circular Linked List

1) No Null Pointers **:** The difference between doubly linked and a circular doubly linked list is the same as that exists between a singly linked list and a circular linked list. The circular doubly linked list does not contain NULL in the previous field of the first node and the next field of the last node.

2) Circular Looping **:** One of the significant properties of doubly circular linked lists is their ability to form a continuous loop. This loop ensures that traversal can continue indefinitely, as the last node connects back to the first node. It eliminates the need for special cases when reaching the end of the list and allows for cyclic operations or data structures that require continuous looping, such as circular buffers or circular queues.

Difference between Linear and Circular Linked List

Terms	Circular Linked List	Linear Linked List
Connectivity	Forms a circular loop	Linear structure
Last Node Pointer	Points back to the first node	Points to null (in SLL) or next node (in DLL)
Beginning and End	No clear beginning or end	Clear beginning (head) and end (tail)
Looping Capability	Allows continuous looping through the list	Does not support continuous looping
Implementation Complexity	Slightly more complex	Simpler implementation
Memory Overhead	Slightly higher due to circular structure	No additional memory overhead
Use case	Circular operations, cyclic data structures (ex - undo - redo operations)	Linear data storage, general use cases (ex - any database Management System)

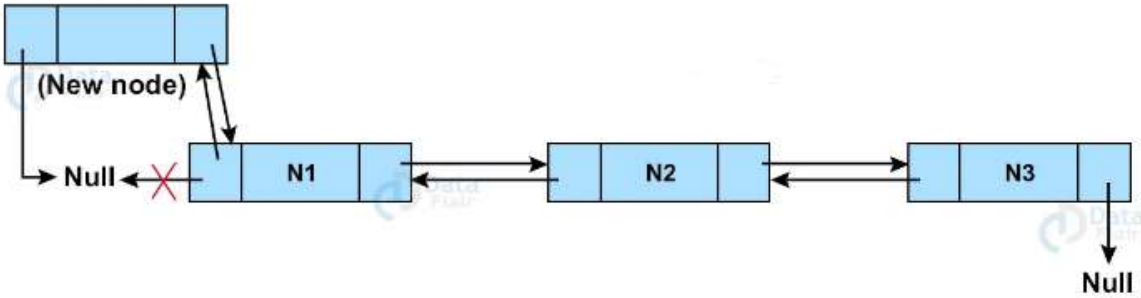
TIME AND SPACE COMPLEXITY ANALYSIS OF DOUBLY LINKED LIST

TIME COMPLEXITY :

Time complexity is a measure of the amount of time required to run an algorithm or perform a specific operation on a given input. It provides an estimation of how the algorithm's runtime grows as the input size increases.

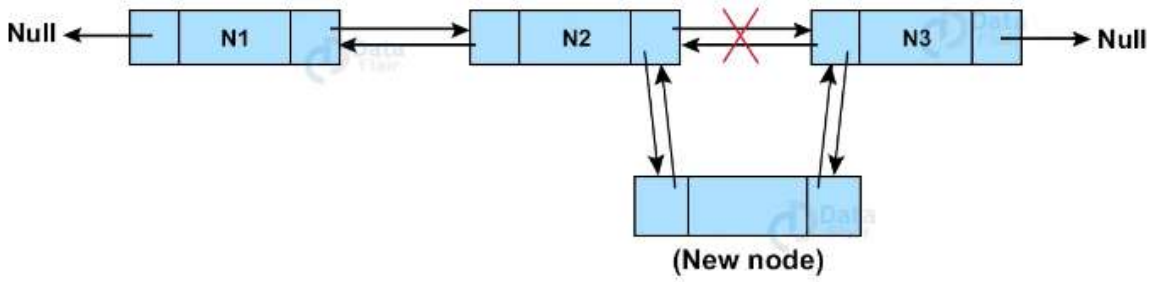
1) Best case (Insertion in the beginning)

The time complexity of inserting an element at the beginning of a linked list is **O(1)** because it takes a constant amount of time regardless of the size of the list.This is because we do not traverse the whole list , just making changes with the head node.



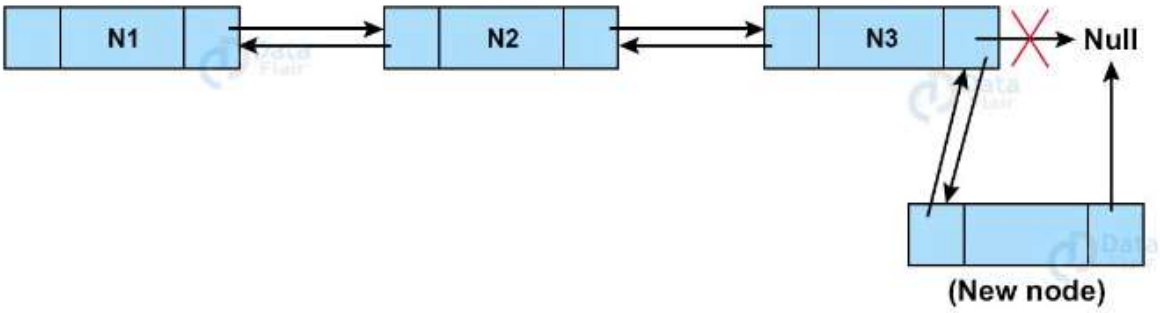
2) **Average case (Insertion in between)**

The time complexity of inserting an element in between of a linked list is **O(n)** , where n is the number of nodes in the list. It requires traversing the list from the head or a specific position to reach the desired insertion point.



3) **Worst case (Insertion at end) - Without using Tail pointer**

The time complexity of inserting an element at the end of a linked list is **O(n)** , where n is the number of nodes in the list. It requires traversing the list from the head to the last pointer which is pointing NULL.



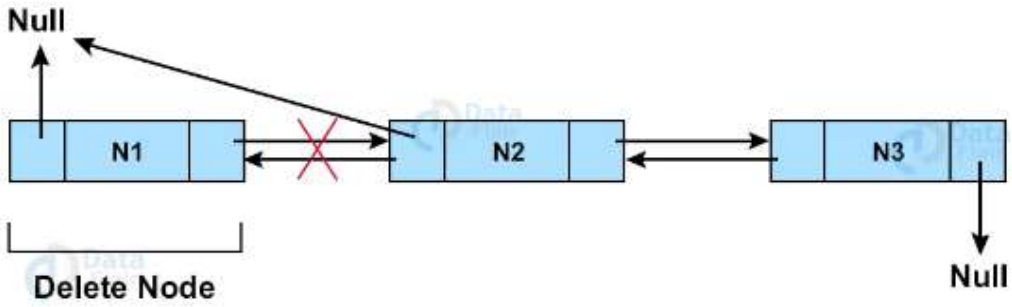
4) **Best case (Insertion at end) - using Tail pointer**

The time complexity of inserting an element at the end of a linked list can be made **O(1)** .
This is possible if last node is made tail node thus, we started traversing from last which eliminates the need of full traversal in a list.

DELETION

1) **Best case (Deletion at first)**

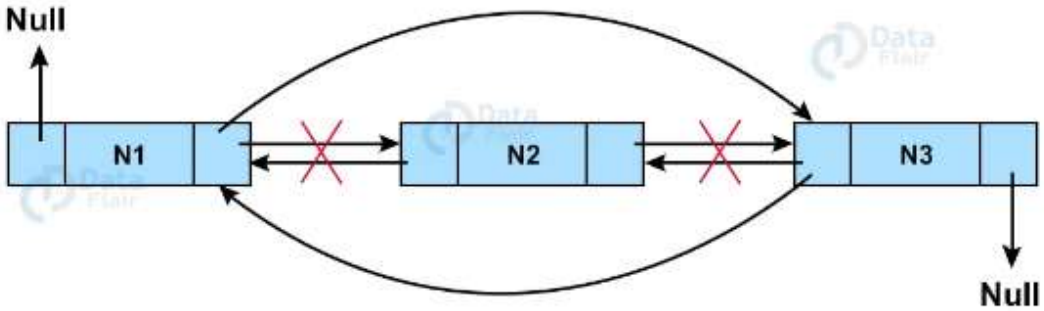
The time complexity of deleting an element from the beginning of a linked list is **O(1)** because it takes a constant amount of time regardless of the size of the list.This is because instead of traversing the whole list , we just free the head pointer.



.

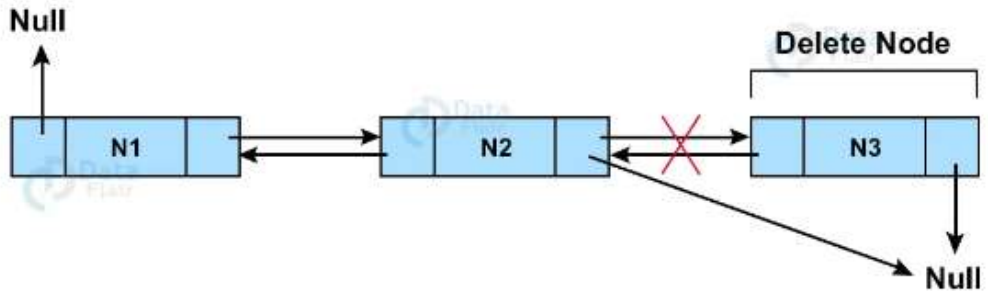
2) **Average case (Deletion in between)**

The time complexity of deleting an element in between of a linked list is **O(n)** , where n is the number of nodes in the list. It requires traversing the list from the head or a specific position to reach the desired deletion node.



3) Worst case (Deletion at end) - Without using tail pointer

The time complexity of deleting an element from the end of a linked list is **O(n)** , where n is the number of nodes in the list. It requires traversing the list from the head to a last node (pointing to the NULL).



4) Best case (Deletion at end) - Using tail pointer

The time complexity of deleting an element from the end of a linked list can be made **O(1)** , This is possible when we started traversing from the Tail pointer (end of the list) , thus eliminating the need of traversdal from the beginning of the list.

5) Deletion of a Complete Linked List

To delete the complete DLL, we would need to traverse each node in the list and deallocate the memory associated with it. This process needs to be repeated for every node in the DLL, resulting in a **O(n)** time complexity.

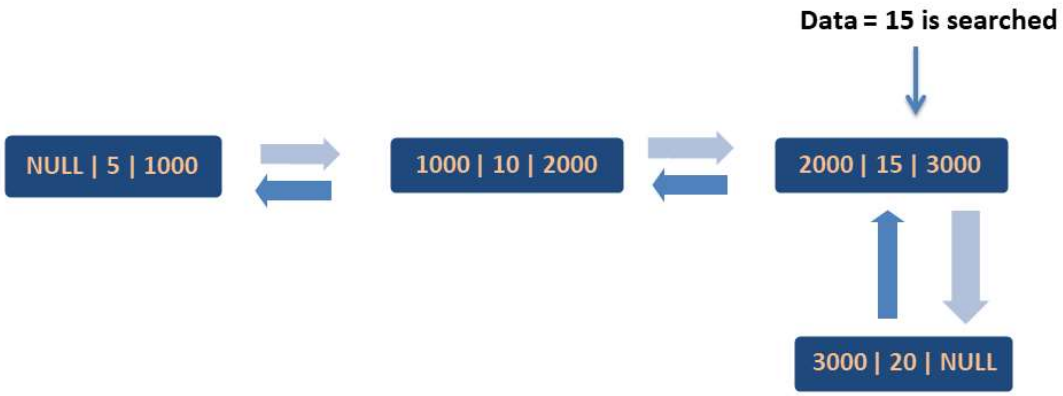
SEARCHING / TRAVERSING

1) **Best case (1st node is searched)**

If we want to search a first element from the doubly linked list , then it is considered as the best case and its time complexity is **O(1).** This is because we don't have to traverse the whole list , thus saving time.

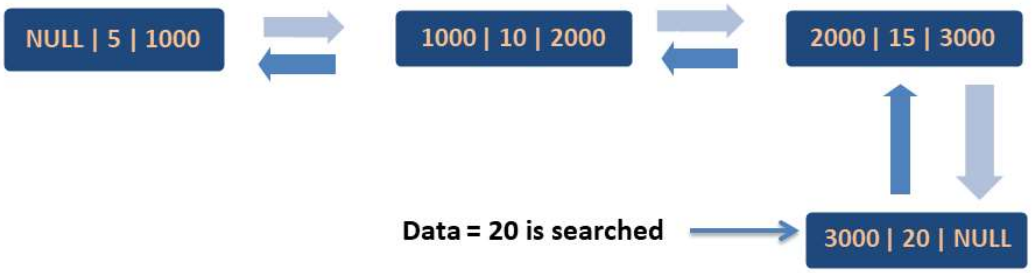
2) **Average case (Any node in between is searched)**

If we want to search any element (between first and last node) in the doubly linked list , then it is considered as the average case and its time complexity is **O(n).** This is because we have to traverse almost more than half of the list which takes Order n time.



3) Worst case (Last node is searched) - Without using Tail pointer

If we want to search the last element from the doubly linked list , then it is considered as the worst case and its time complexity is **O(n).** This is because we have to traverse the whole list in order to reach the last node.



4) Best case (Last node is searched) - Using Tail pointer

If we want to search the last element from the doubly linked list , then its time complexity can converted to **O(1).** This is because we can start searching the element from end of the list using tail pointer , thus eliminating the need of traversal of a complete linked list.

- Creating the structure of a DLL involves allocating memory for the head and tail pointers and setting their initial values to null. This operation does not depend on the size of the DLL or the number of elements to be inserted, so it can be done in constant time.Hence, time complexity will be **O(1)**.

SPACE COMPLEXITY :

The space complexity refers to the amount of memory or space required to store and manipulate data. It quantifies the amount of additional memory needed as the input size grows.

The space complexity of individual insertion and deletion operations in a Doubly Linked List (DLL) is O(1), which means it requires a constant amount of additional memory.

In a DLL, when you insert or delete a node, you only need to allocate or deallocate memory for that specific node. The amount of memory required for the DLL does not change with the size of the list. Whether you insert or delete one node or a hundred nodes, the memory requirements for the DLL operations remain constant.

TIME AND SPACE COMPLEXITY OF ALL THE OPERATIONS ON DOUBLY LINKED LIST

Operation	Time complexity (N → Number of nodes)	Space Complexity
Insertion - at begin	O (1)	O (1)
Insertion - in between	O (N)	O (1)
Insertion - at end (Tail)	O (1)	O (1)
Insertion - at end (No tail)	O (N)	O (1)
Deletion - at begin	O (1)	O (1)
Deletion - in between	O (N)	O (1)
Deletion - at end (Tail)	O (1)	O (1)
Deletion - at end (No tail)	O (N)	O (1)
Traversing - at begin	O (1)	O (1)
Traversing - in between	O (N)	O (1)
Traversing - at end (Tail)	O (1)	O (1)
Traversing - at end (No tail)	O (N)	O (1)
Creation of DLL	O (1)	O (1)
Deletion of DLL	O (N)	O (1)