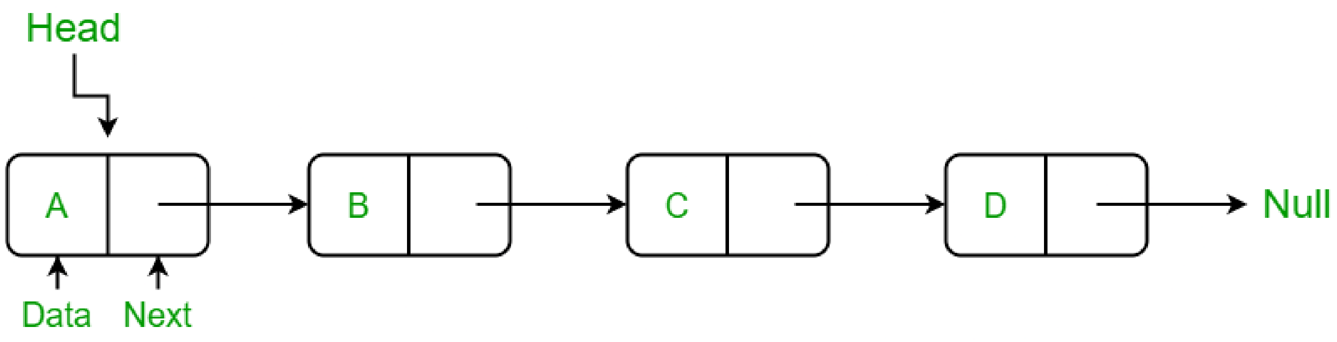
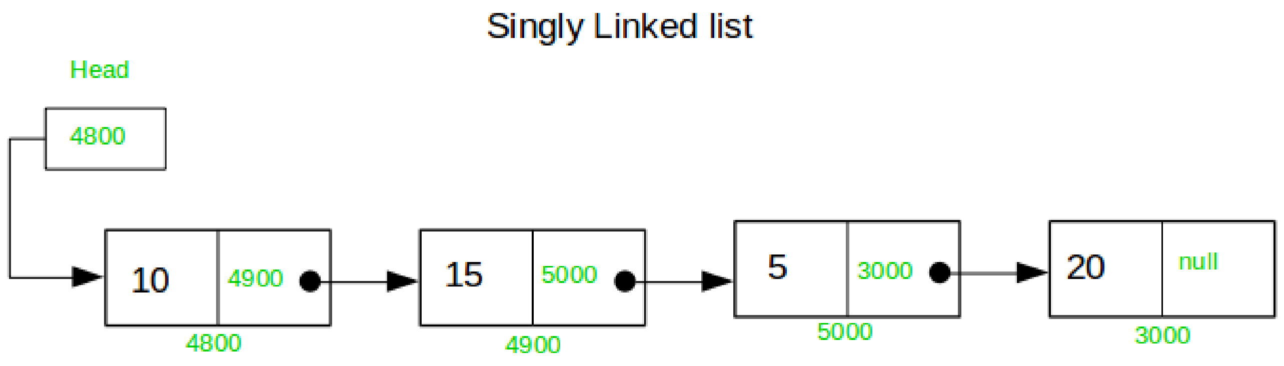
Session 4.5 - LinkedLists

A linked list is a data structure used to store a collection of elements. Unlike arrays or lists, linked lists do not require contiguous memory allocation. Instead, elements are stored in nodes, and each node contains a reference (or link) to the next node in the sequence.



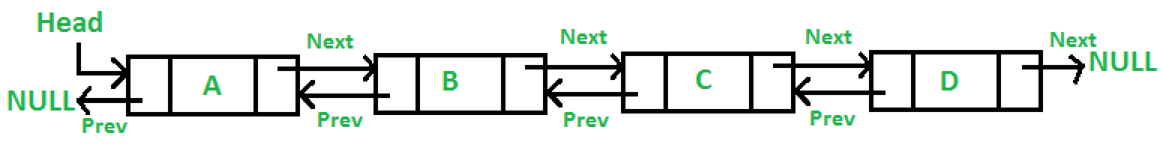
The two main types of linked lists are singly linked lists and doubly linked lists:

1. Singly Linked List:
    a. In a singly linked list, each node contains data and a reference to the next node.
    b. The last node in the list points to null to indicate the end of the list.
    c. Traversing a singly linked list can only be done in one direction, from the head (first node) to the tail (last node).



2. Doubly Linked List:

a. In a doubly linked list, each node contains data, a reference to the next node, and a reference to the previous node.
b. The first node's previous reference and the last node's next reference are set to null.
c. Traversing a doubly linked list can be done in both directions, forward and backward.



Common operations and characteristics of linked lists include:

- Insertion: Adding a new node to the list, either at the beginning (prepend), end (append), or a specific position.
- Deletion: Removing a node from the list, either from the beginning, end, or a specific position.
- Searching: Finding a specific element in the list by traversing through the nodes.
- Accessing: Retrieving the data stored in a specific node.
- Size: The number of elements in the linked list.
- Dynamic Size: Linked lists can dynamically grow or shrink as elements are added or removed, without requiring reallocation of memory.

Linked lists have advantages and disadvantages compared to other data structures. They provide efficient insertion and deletion at the beginning or end of the list but have slower access times for random elements compared to arrays. Additionally, linked lists use extra memory for storing the links between nodes.

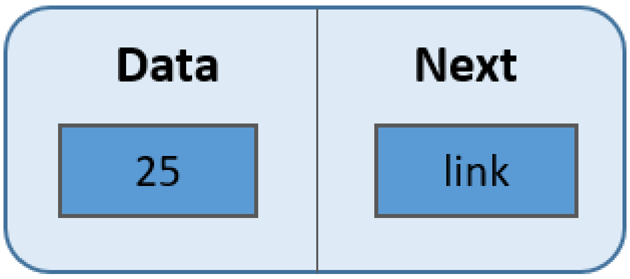
Overall, linked lists are commonly used in situations where efficient insertion and deletion operations are required, or when the size of the collection may change frequently.

Singly Linked List:

A singly linked list is a type of linked list where each node contains data and a reference (link) to the next node in the sequence. Here's an overview of singly linked lists:

Node Structure:

- Each node in the singly linked list consists of two parts:
    - Data: The actual value or information stored in the node.
    - Next: A reference (or link) to the next node in the list.

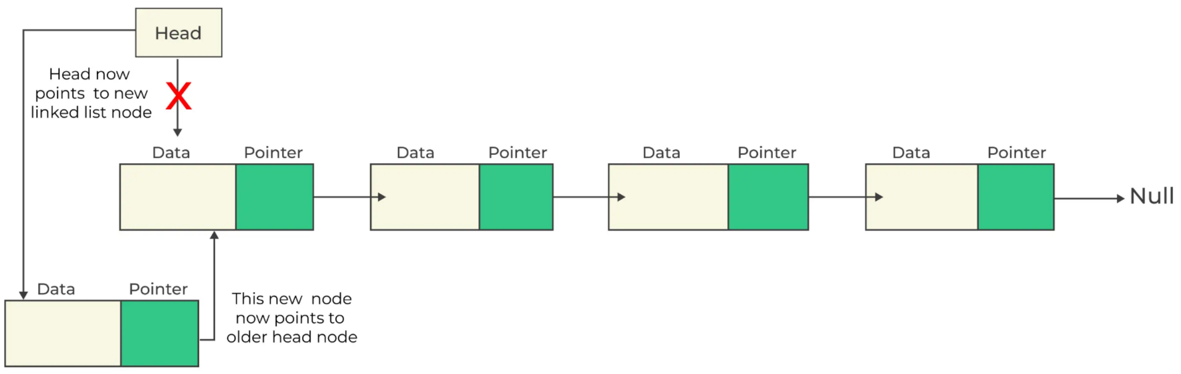


Head and Tail:

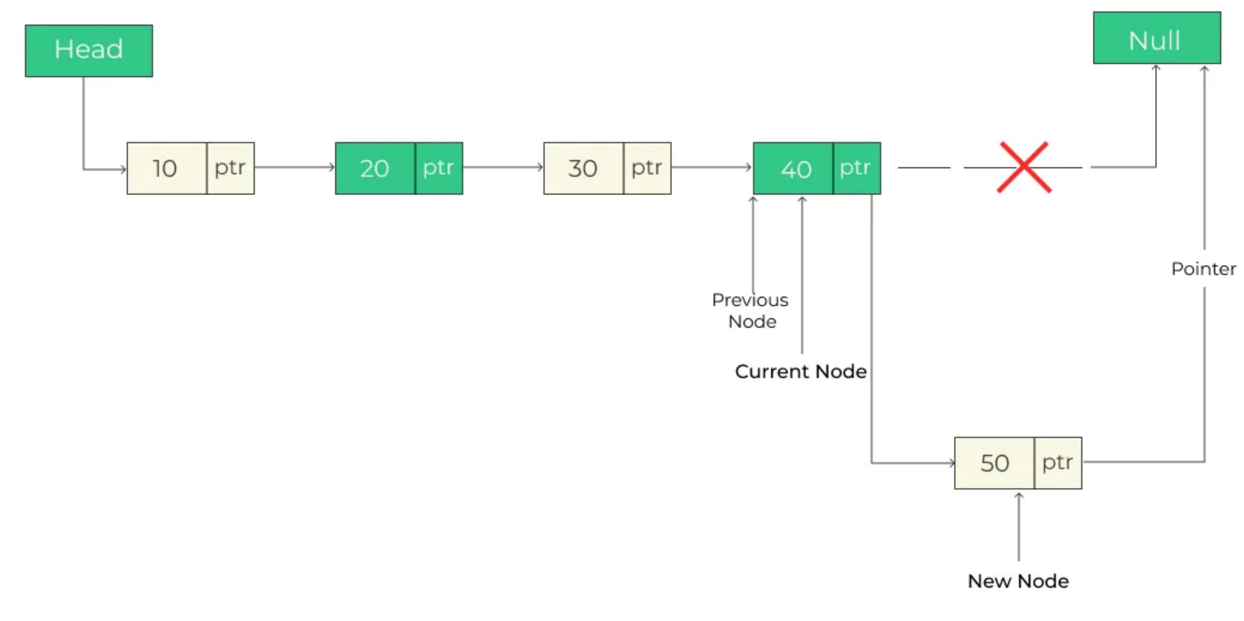
- The linked list is typically represented by a reference to the first node, called the head.
- The last node in the list points to null to indicate the end, often called the tail.

Basic Operations:

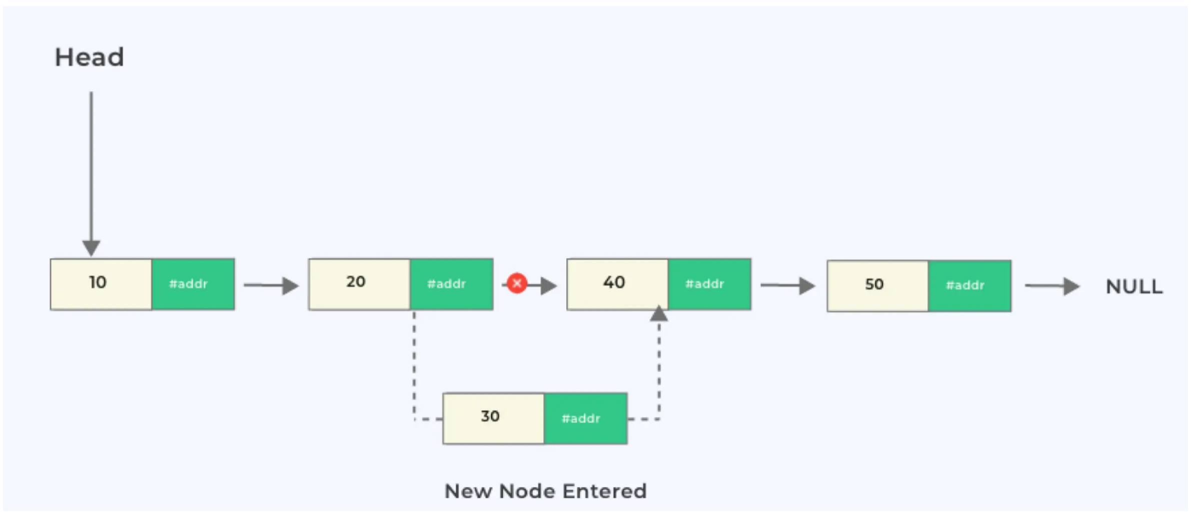
- Insertion at the Beginning (Prepend):
    - Create a new node with the desired data.
    - Set the next reference of the new node to the current head.
    - Update the head reference to the new node.


Head
Head now points to new linked list node
X
Data
Pointer
Data
Pointer
Data
Pointer
Data
Pointer
Null
Data
Pointer
This new node now points to older head node

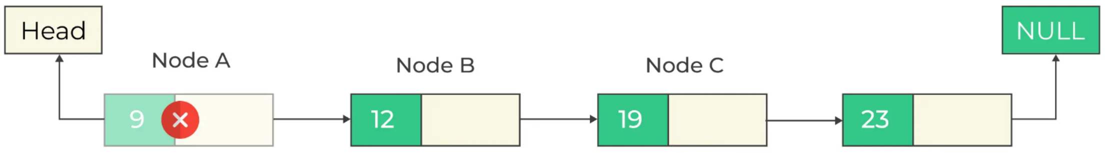
- Insertion at the End (Append):
    - Create a new node with the desired data.
    - If the list is empty (head is null), set the head and tail references to the new node.
    - Otherwise, set the next reference of the current tail to the new node and update the tail reference to the new node.



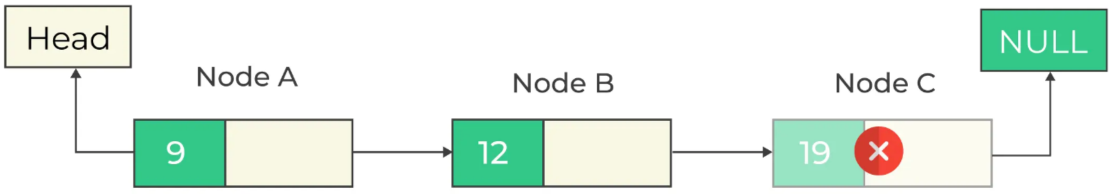
- Insertion at a Specific Position:
  - Traverse the list until reaching the desired position or the end of the list.
  - Create a new node with the desired data.
  - Set the next reference of the new node to the next node of the current position.
  - Update the next reference of the current position to the new node.



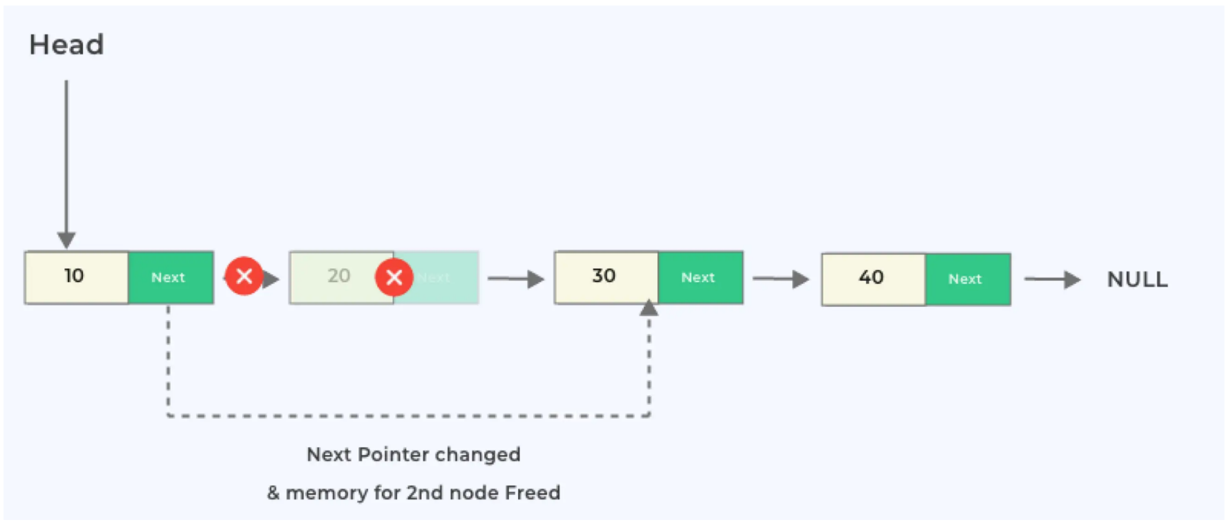
- Deletion at the Beginning:
  - If the list is empty, return or throw an error.
  - Update the head reference to the next node.
  - Optionally, free the memory of the removed node.



- Deletion at the End:
    - If the list is empty, return or throw an error.
    - Traverse the list until reaching the node just before the tail.
    - Update the next reference of the current node to null and update the tail reference to the current node.
    - Optionally, free the memory of the removed node.



- Deletion at a Specific Position:
    - Traverse the list until reaching the node just before the desired position.
    - Update the next reference of the current node to skip the node at the desired position.
    - Optionally, free the memory of the removed node.



- Searching:
    - Start from the head and traverse the list by following the next references until finding the desired data or reaching the end of the list.

- Size:
    - Maintain a counter variable that keeps track of the number of nodes in the list.
    - Increment the counter when inserting nodes and decrement it when deleting nodes.

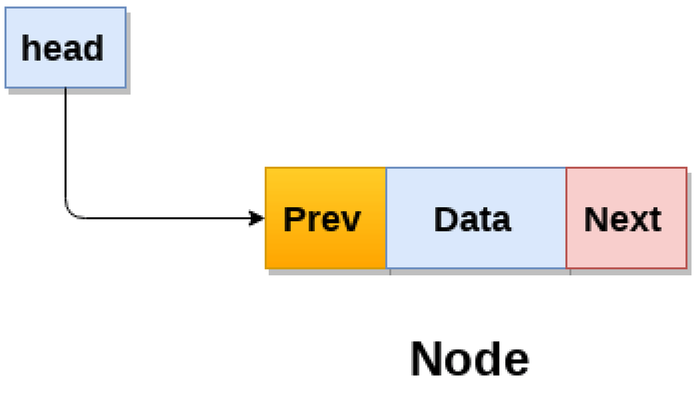
Singly linked lists are useful when efficient insertion and deletion at the beginning of the list are required, but random access to elements is less important. They are dynamic in size and can be easily modified by adjusting the links between nodes.

Doubly LinkedList:

A doubly linked list is a type of linked list where each node contains data, a reference to the next node, and a reference to the previous node. Here's an overview of doubly linked lists:

Node Structure:

- Each node in the doubly linked list consists of three parts:
    - Data: The actual value or information stored in the node.
    - Next: A reference (link) to the next node in the list.
    - Previous: A reference (link) to the previous node in the list.

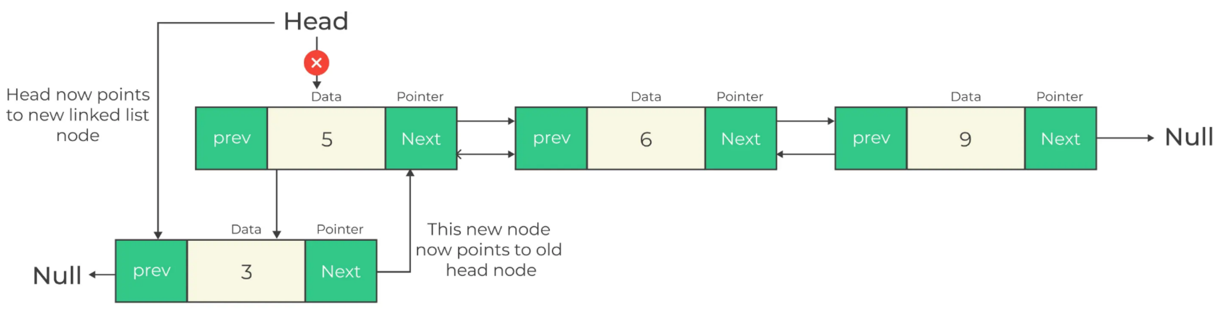


Head and Tail:

- The linked list is typically represented by a reference to the first node, called the head.
- The last node in the list is represented by a reference to null, often called the tail.

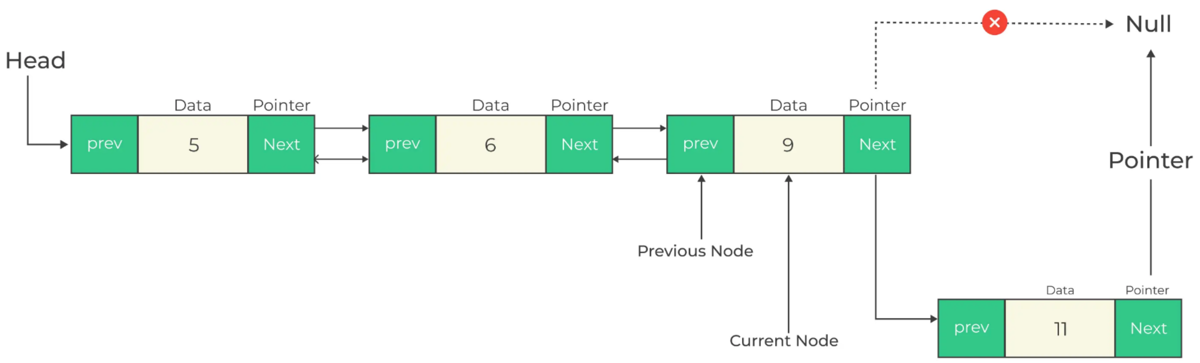
Basic Operations:

- Insertion at the Beginning (Prepend):
    - Create a new node with the desired data.
    - Set the next reference of the new node to the current head.
    - Set the previous reference of the new node to null.
    - If the current head is not null, update its previous reference to the new node.
    - Update the head reference to the new node.

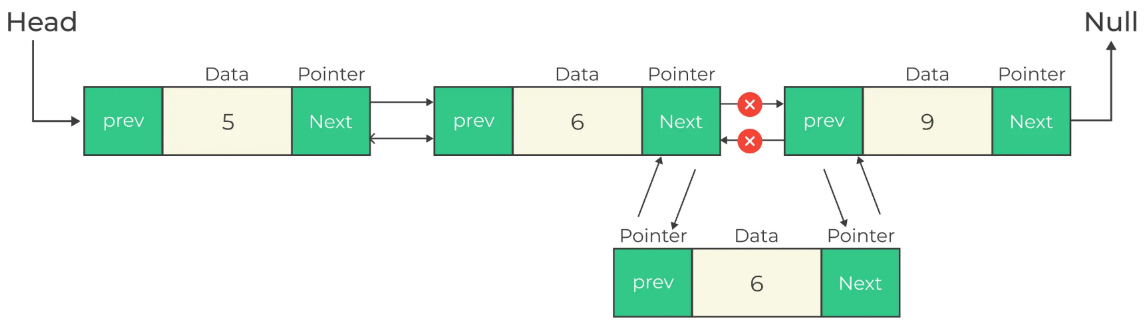


- Insertion at the End (Append):
    - Create a new node with the desired data.
    - Set the next reference of the new node to null.
    - If the list is empty (head is null), set the head reference to the new node.

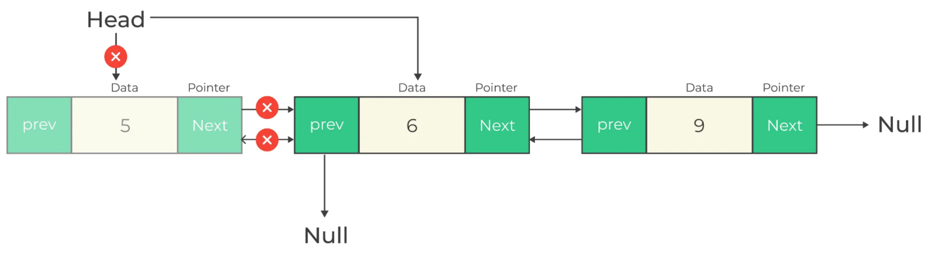
- Otherwise, set the next reference of the current tail to the new node.
- Set the previous reference of the new node to the current tail.
- Update the tail reference to the new node.



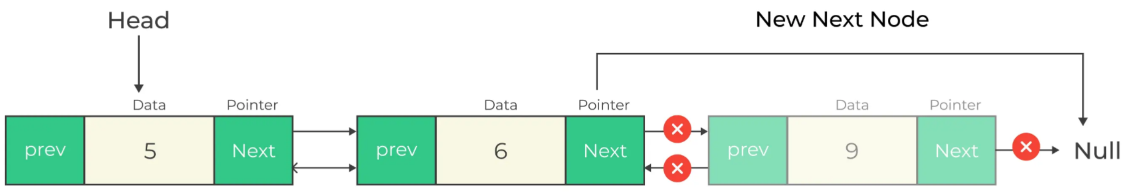
- Insertion at a Specific Position:
  - Traverse the list until reaching the desired position or the end of the list.
  - Create a new node with the desired data.
  - Set the next reference of the new node to the next node of the current position.
  - Set the previous reference of the new node to the current position.
  - Update the next reference of the current position to the new node.
  - If the next node is not null, update its previous reference to the new node.



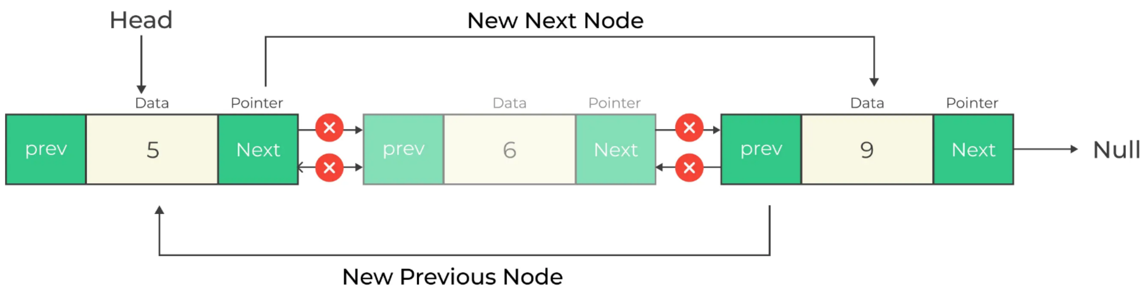
- Deletion at the Beginning:
  - If the list is empty, return or throw an error.
  - Update the head reference to the next node.
  - If the new head is not null, set its previous reference to null.
  - Optionally, free the memory of the removed node.



- Deletion at the End:
    - If the list is empty, return or throw an error.
    - Update the tail reference to the previous node of the current tail.
    - If the new tail is not null, set its next reference to null.
    - Optionally, free the memory of the removed node.



- Deletion at a Specific Position:
    - Traverse the list until reaching the desired position or the end of the list.
    - Update the next reference of the previous node to the next node of the current position.
    - If the next node is not null, update its previous reference to the previous node.
    - Optionally, free the memory of the removed node.

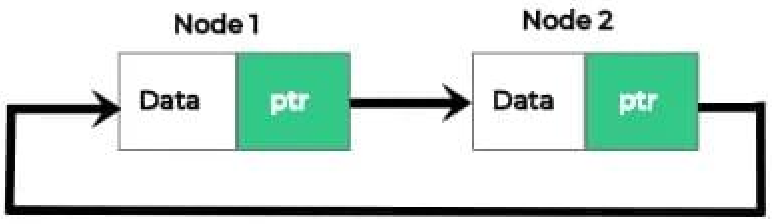


- Searching:
    - Start from the head or tail and traverse the list by following the next or previous references until finding the desired data or reaching the end of the list.
- Size:
    - Maintain a counter variable that keeps track of the number of nodes in the list.
    - Increment the counter when inserting nodes and decrement it when deleting nodes.

Doubly linked lists allow traversal in both forward and backward directions, which can be useful in various scenarios. However, they require additional memory for storing the previous references in each node.

Circular LinkedList:

The concept of a circular linked list revolves around creating a data structure where the last node in the list points back to the first node, forming a loop or circle.



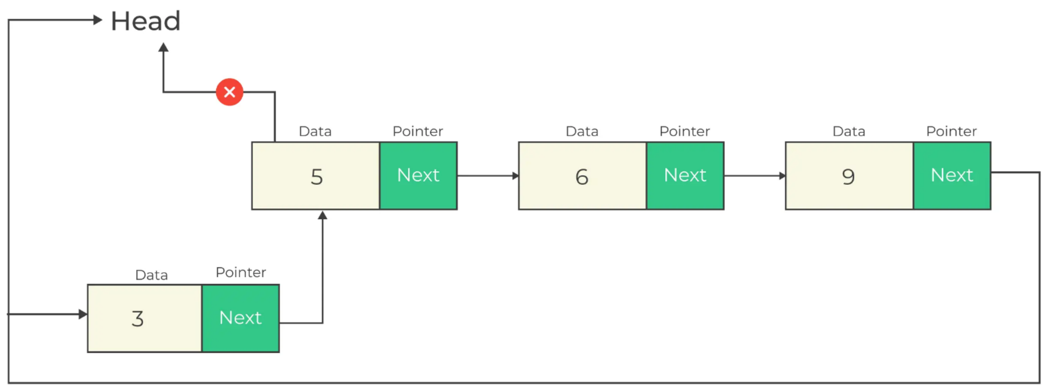
Unlike a traditional linked list, which ends with a null reference, the circular linked list provides a way to traverse the list indefinitely.
Here are a few key aspects of the circular linked list concept:

- Circular Structure: In a circular linked list, the last node's "next" pointer does not point to null but instead references the first node in the list. This creates a circular structure, allowing traversal from the last node back to the first node.
- Infinite Traversal: The circular structure enables you to traverse the entire list without encountering a null reference. You can start from any node and continue traversing until you reach the node you started from, and then repeat the process indefinitely.
- Insertion and Deletion: Inserting or deleting nodes in a circular linked list involves updating the "next" pointers accordingly. When inserting a node, you adjust the pointers of the neighboring nodes to include the new node. Similarly, when deleting a node, you update the pointers of the adjacent nodes to bypass the deleted node.
- Applications: Circular linked lists find applications in various scenarios. For example, they can be used to implement circular buffers or circular queues, where the elements wrap around in a circular fashion. They are also useful in applications where you need to repeatedly cycle through a list of items.
- Implementation Considerations: When implementing a circular linked list, you need to handle special cases such as an empty list (where the head is null), a list with a single node (where the head points to itself), and correctly updating the "next" pointers when inserting or deleting nodes.

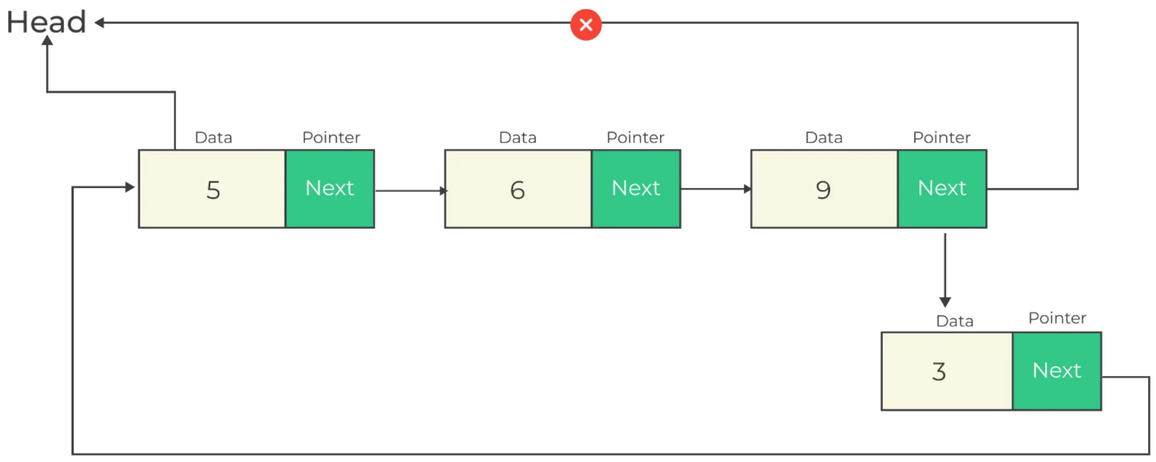
It's important to note that due to the circular nature of the list, you need to be cautious when iterating over it to avoid infinite loops. Typically, a condition is added to check if you have traversed the entire list by comparing the current node to the head node.
Overall, the circular linked list concept extends the functionality of traditional linked lists by creating a looped structure, allowing for seamless traversal and utilization in various applications.

A circular linked list supports various operations that can be performed to manipulate and manage the list. Here are the commonly used operations for a circular linked list:

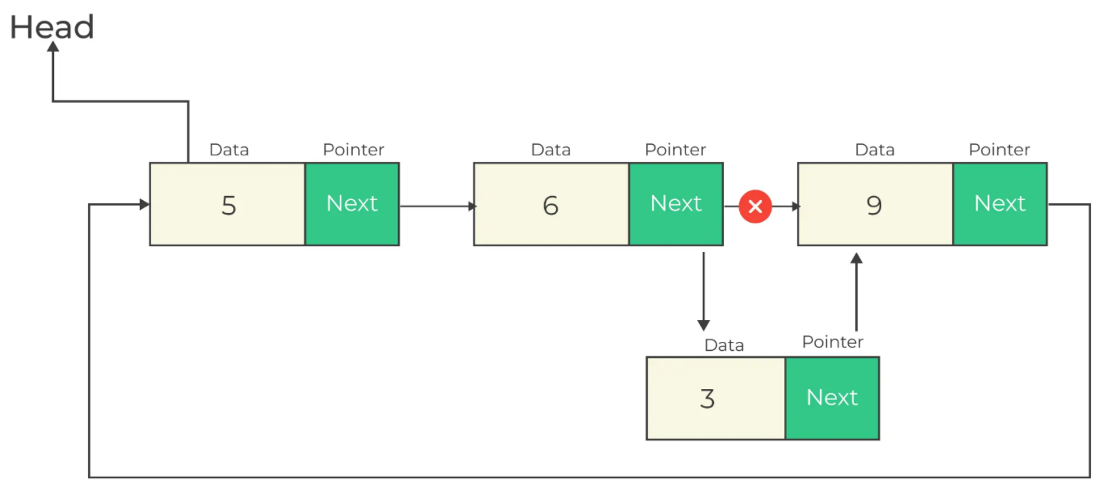
- Insertion at the Beginning: This operation involves adding a new node at the beginning of the circular linked list. It requires updating the "next" pointers of the new node and the last node to maintain the circular structure.



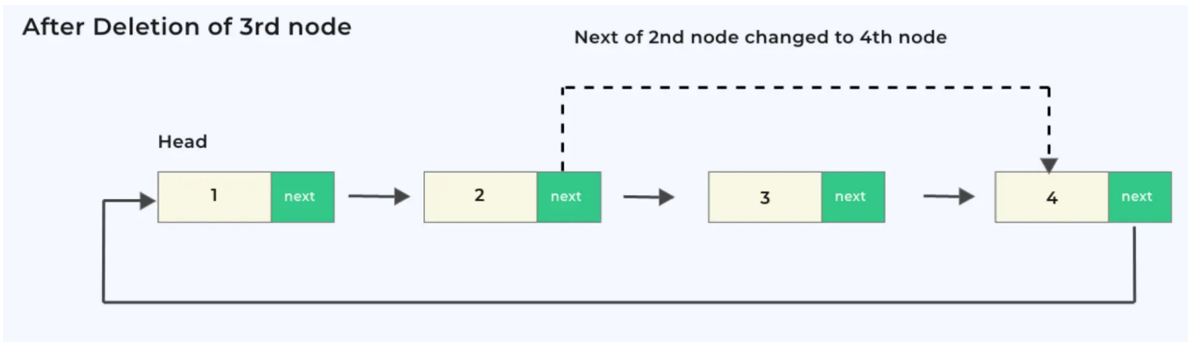
- Insertion at the End: This operation adds a new node at the end of the circular linked list. It involves updating the "next" pointers of the last node and the new node.



- Insertion at a Specific Position: This operation inserts a new node at a given position in the circular linked list. It requires updating the "next" pointers of the neighboring nodes and the new node.



- Deletion of a Node: This operation removes a specified node from the circular linked list. It involves updating the "next" pointers of the neighboring nodes to bypass the deleted node.


After Deletion of 3rd node
Next of 2nd node changed to 4th node
Head
1
next
2
next
3
next
4
next

- Searching for a Node: This operation searches for a specific value or condition in the circular linked list. It typically involves iterating through the list and comparing the values until a match is found or the entire list is traversed.
- Traversal: Traversing a circular linked list involves visiting each node in the list and performing some operation. Since the list is circular, traversal can start from any node and continue until the starting node is reached again.
- Counting the Number of Nodes: This operation determines the total number of nodes present in the circular linked list. It involves traversing the list and incrementing a counter for each visited node.
- Reversing the List: Reversing a circular linked list changes the direction of the links, making the last node become the first node and vice versa. It requires updating the "next" pointers of each node accordingly.

These are the basic operations performed on a circular linked list. You can implement additional operations based on your specific requirements, such as updating a node's value, checking if the list is empty, or finding the nth node in the list.

Node-based storage with arrays

Node-based storage with arrays refers to a data structure implementation where nodes are stored using arrays. This approach allows you to represent linked structures, such as linked lists or trees, using arrays instead of explicit pointers or references.

In a node-based storage with arrays, each node is typically represented as an element of an array, and the relationships between nodes are established by using indices or positions within the array.

Here's a general overview of how node-based storage with arrays can be used for different data structures:

- Linked List: In a linked list, each node contains a data element and a reference to the next node. With node-based storage using arrays, you can represent a linked list by creating an array where each element corresponds to a node. The index of each element represents the node's position in the list, and the value at that index stores the node's data. The reference to the next node can be stored as an index value in the current node's array element.
- Binary Tree: In a binary tree, each node has a data element, as well as references to its left and right child nodes. With node-based storage using arrays, you can represent a binary tree by creating an array where each element represents a node. The index of each element corresponds to the node's position in the array, and the value at that index stores the node's data. The indices of the left and right child nodes can be calculated based on the current node's index, allowing you to establish the tree structure within the array.
- General Trees or Graphs: For more complex structures like general trees or graphs, you can adapt the node-based storage approach by using additional arrays to represent relationships between nodes. For example, you can have an array to store child indices for each node in a tree or an adjacency matrix to represent connections between nodes in a graph.

Node-based storage with arrays provides a way to represent linked structures using a fixed-size array, which can be advantageous in scenarios where explicit pointers or dynamic memory allocation may not be feasible or desirable. However, it's worth noting that this approach can have limitations, such as a fixed maximum number of nodes determined by the array size and potentially increased memory consumption for sparse or large structures.

Implementing node-based storage with arrays requires careful management of array indices and mapping relationships between nodes and array elements. It may also require additional logic to handle resizing the array if the structure needs to grow dynamically.