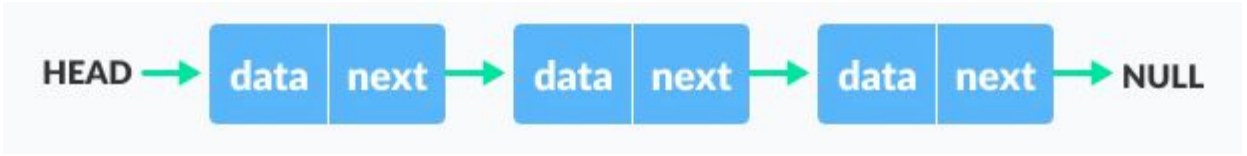
Linked Lists

Ar. Gör Elif Ece Erdem

Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.



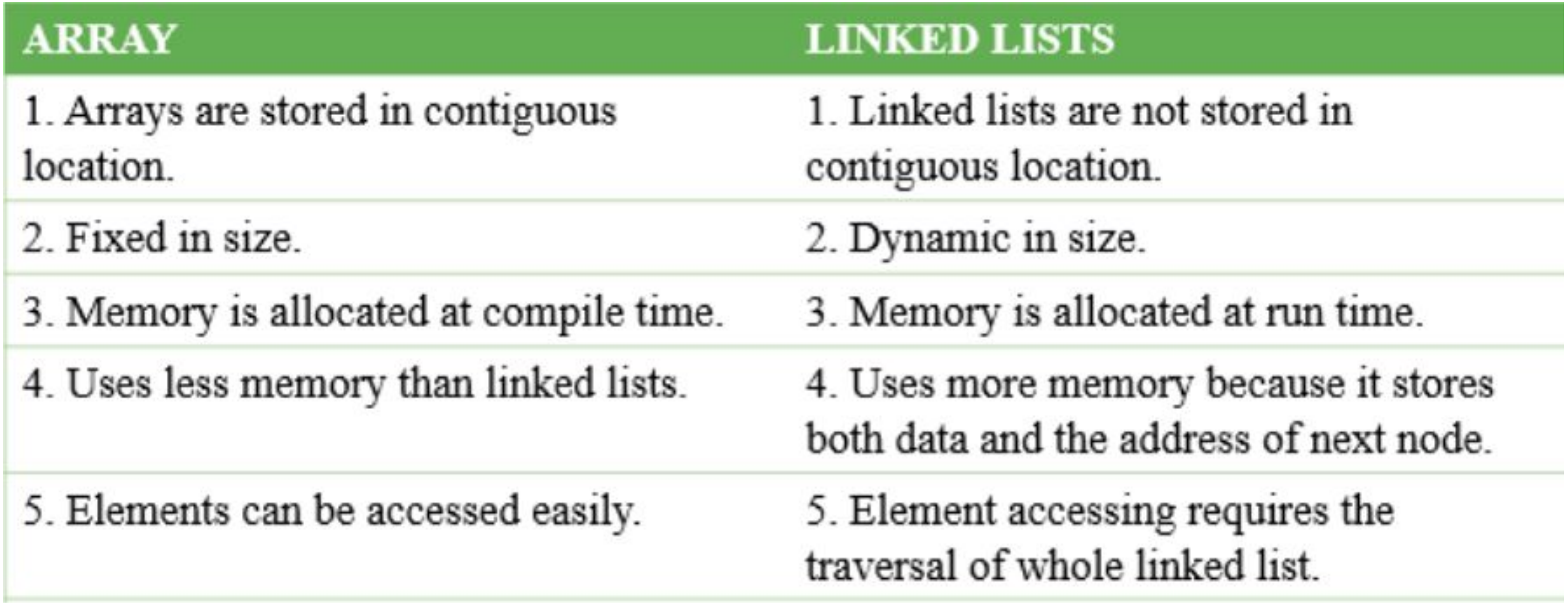
Node Structure: A node in a linked list typically consists of two components:

Data: It holds the actual value or data associated with the node.

Next Pointer: It stores the memory address (reference) of the next node in the sequence.

Head and Tail: The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

Linked List



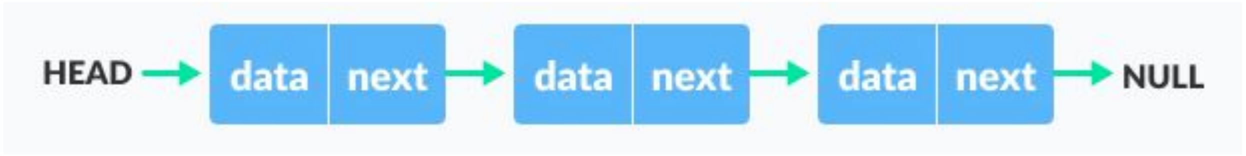
ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.

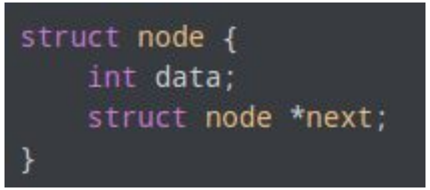
Advantages & Disadvantages of Linked Lists

Advantages	**Disadvantages**
Dynamic size: Linked lists do not have a fixed size, so you can add or remove elements as needed, without having to worry about the size of the list. This makes linked lists a great choice when you need to work with a collection of items whose size can change dynamically.	**Slow Access Time:** Accessing elements in a linked list can be slow, as you need to traverse the linked list to find the element you are looking for, which is an O(n) operation. This makes linked lists a poor choice for situations where you need to access elements quickly.
Memory Efficiency: Linked lists use only as much memory as they need, so they are more efficient with memory compared to arrays, which have a fixed size and can waste memory if not all elements are used.	**Pointers:** Linked lists use pointers to reference the next node, which can make them more complex to understand and use compared to arrays. This complexity can make linked lists more difficult to debug and maintain.
Easy to Implement: Linked lists are relatively simple to implement and understand compared to other data structures like trees and graphs.	**Higher overhead:** Linked lists have a higher overhead compared to arrays, as each node in a linked list requires extra memory to store the reference to the next node.
	Cache Inefficiency: Linked lists are cache-inefficient because the memory is not contiguous. This means that when you traverse a linked list, you are not likely to get the data you need in the cache, leading to cache misses and slow performance.

Singly Linked List

Traversal of items can be done in the forward direction only due to the linking of every node to its next node.

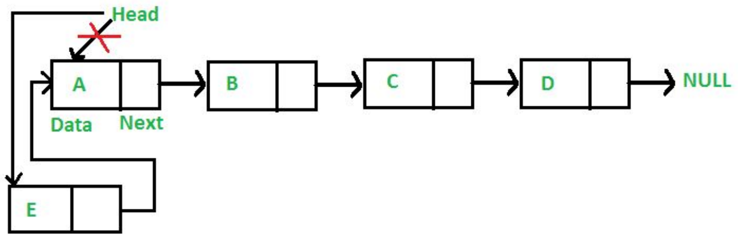




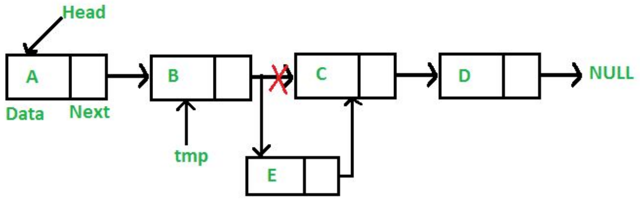
```
struct node {
    int data;
    struct node *next;
}
```

Singly Linked List - Insertion

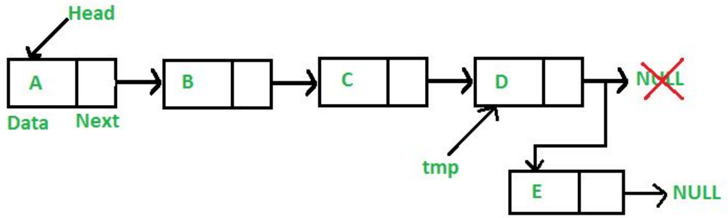
Insert at the beginning of the list



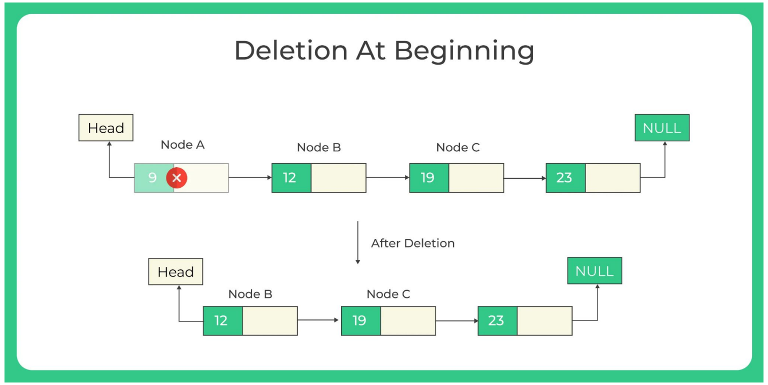
Insert a Node after a Given Node in Linked List

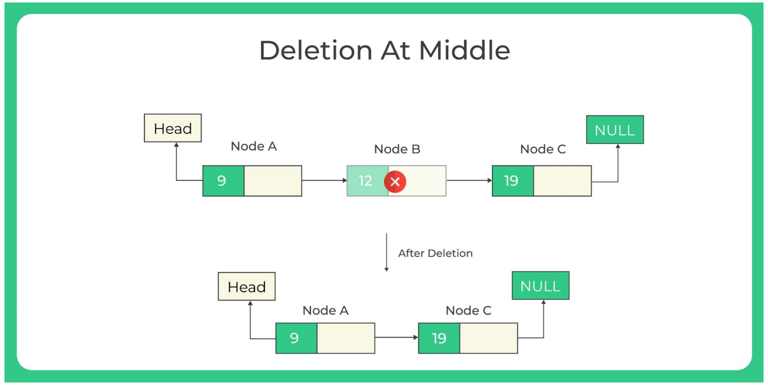


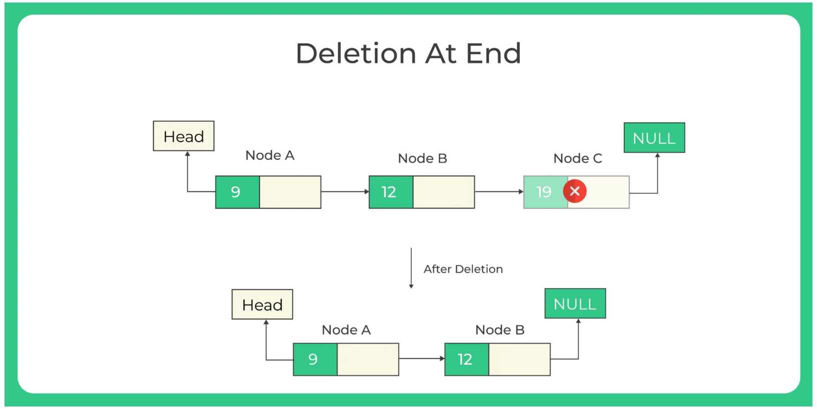
Insert at the end of the list



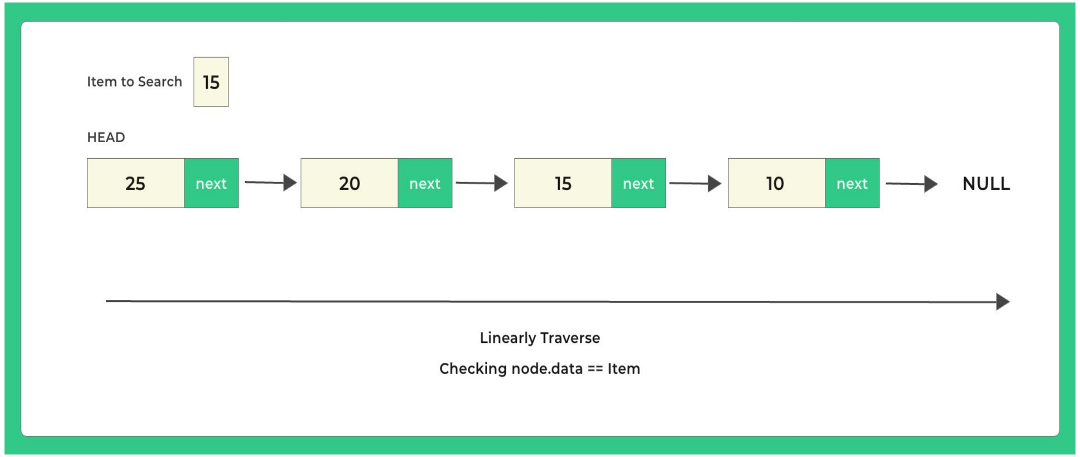
Singly Linked List - Deletion







Singly Linked List - Search

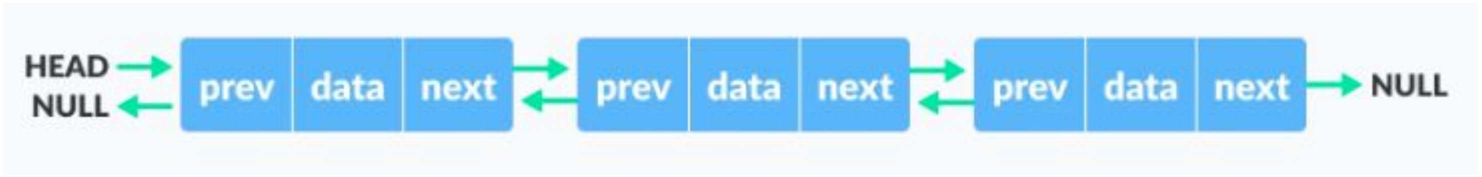


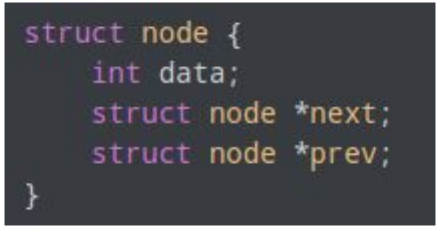
Algorithm:

1. Initialize head = Null
2. Take input from the user for the item wanted to search
3. Linearly traverse the Linked List from head to the end until you hit the null node
4. For each node check if its data value == item user wants to search

Doubly Linked List

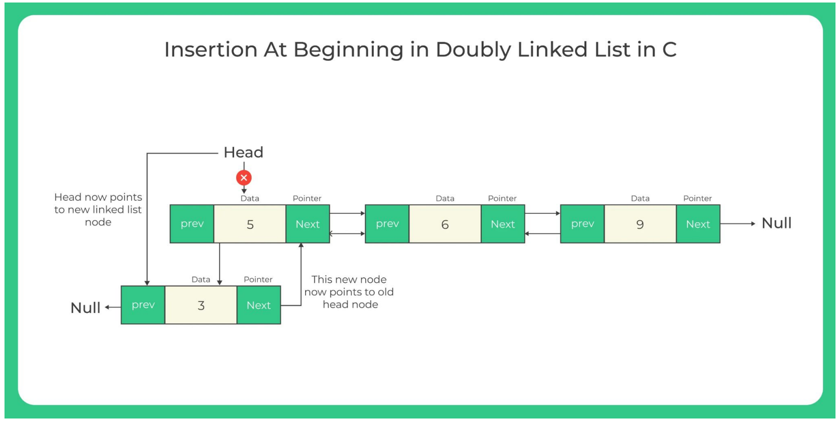
In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward reference.

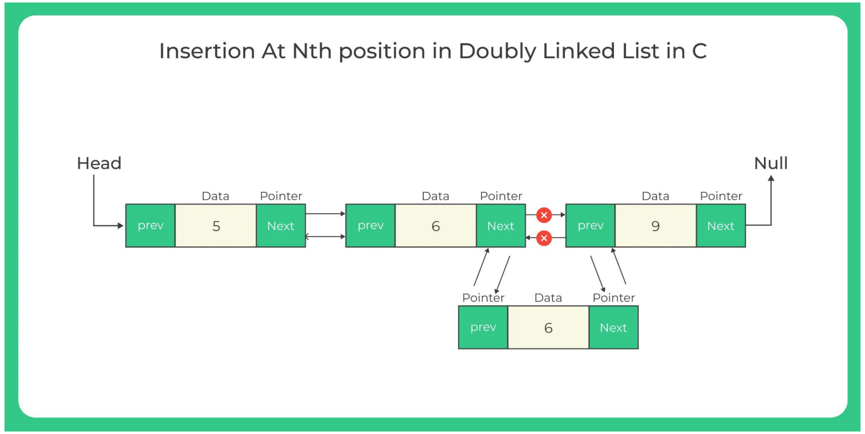


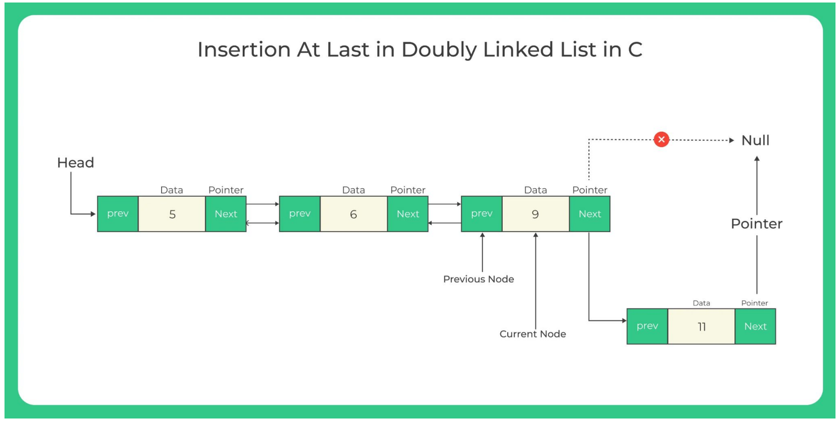


```
struct node {
    int data;
    struct node *next;
    struct node *prev;
}
```

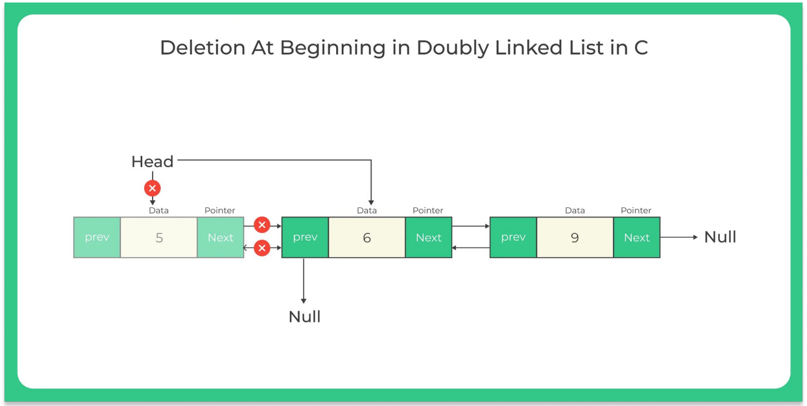
Doubly Linked List - Insertion

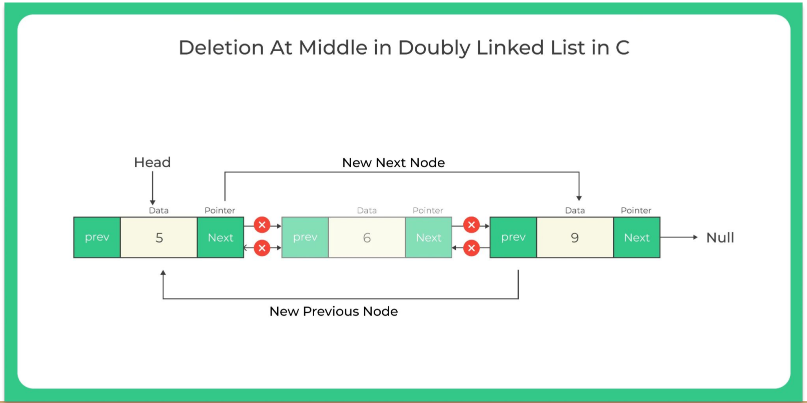

Insertion At Beginning in Doubly Linked List in C
Head
Head now points to new linked list node
Data
Pointer
prev
5
Next
prev
6
Next
prev
9
Next
Null
Data
Pointer
Null
prev
3
Next
This new node now points to old head node

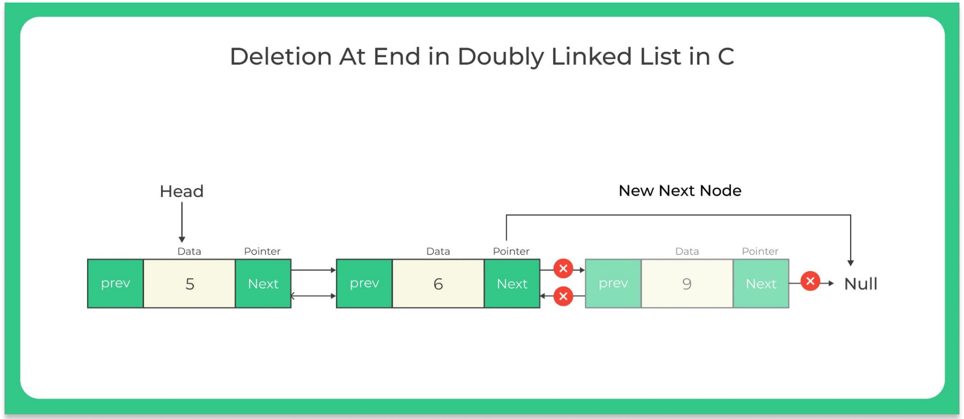




Doubly Linked List - Deletion

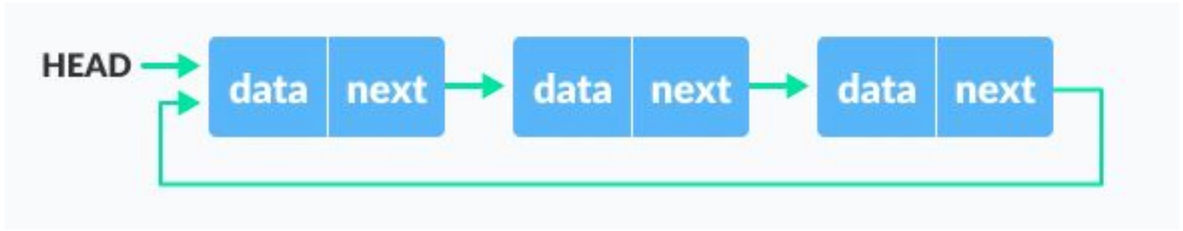


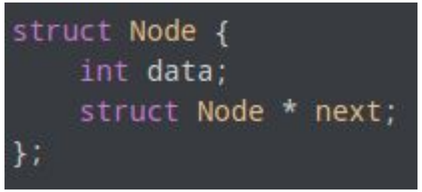




Circular Singly Linked List

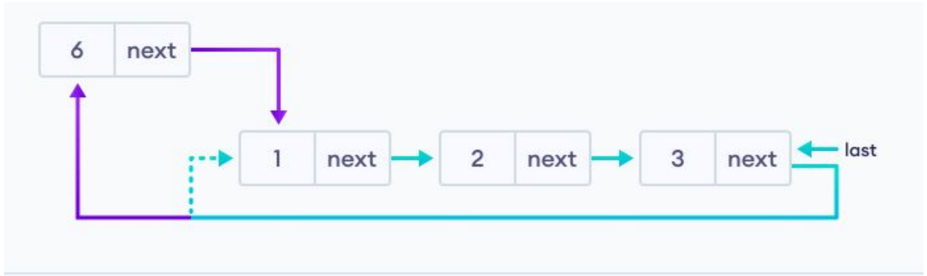
In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.



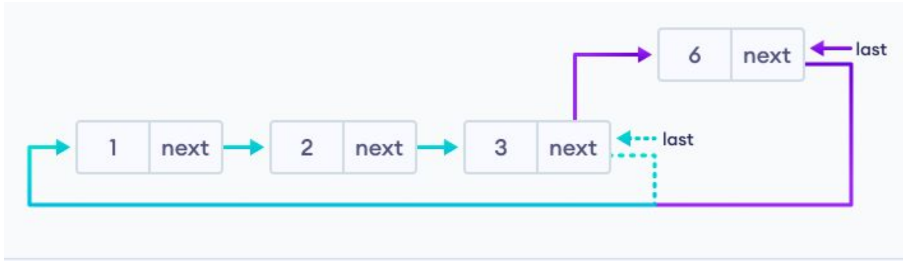


```
struct Node {
    int data;
    struct Node * next;
};
```

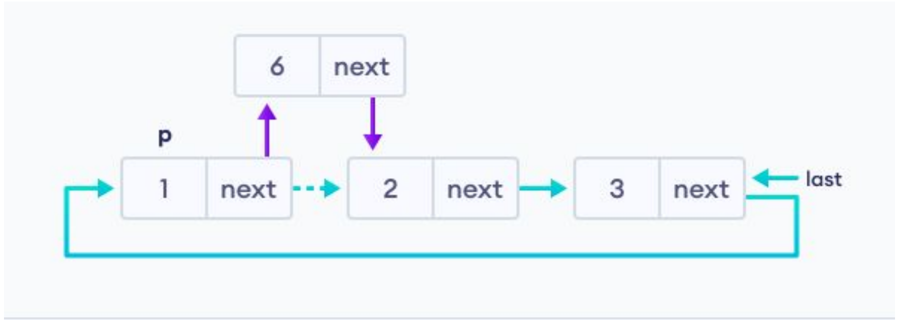
Circular Singly Linked List - Insertion



Insert at the beginning

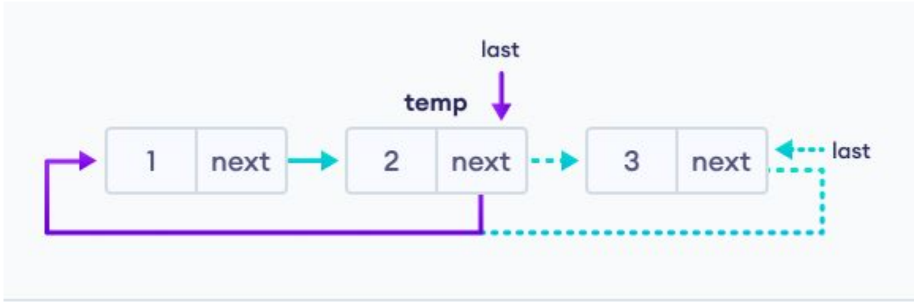


Insert at the end

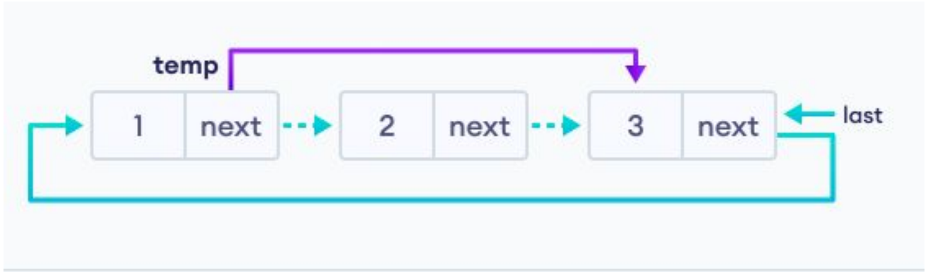


Insertion at a node

Circular Singly Linked List - Deletion



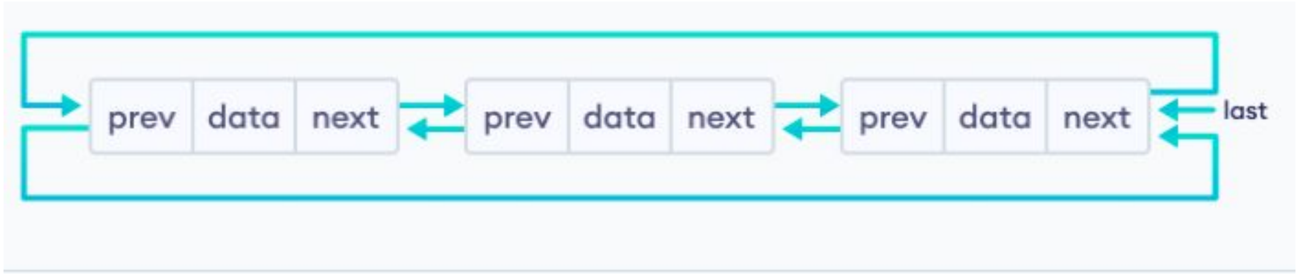
Delete the last node

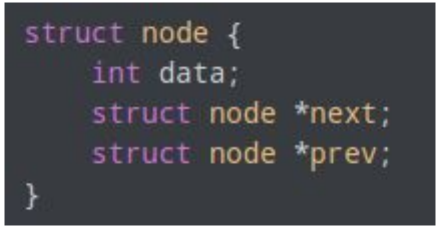


Delete a specific node

Circular Doubly Linked List

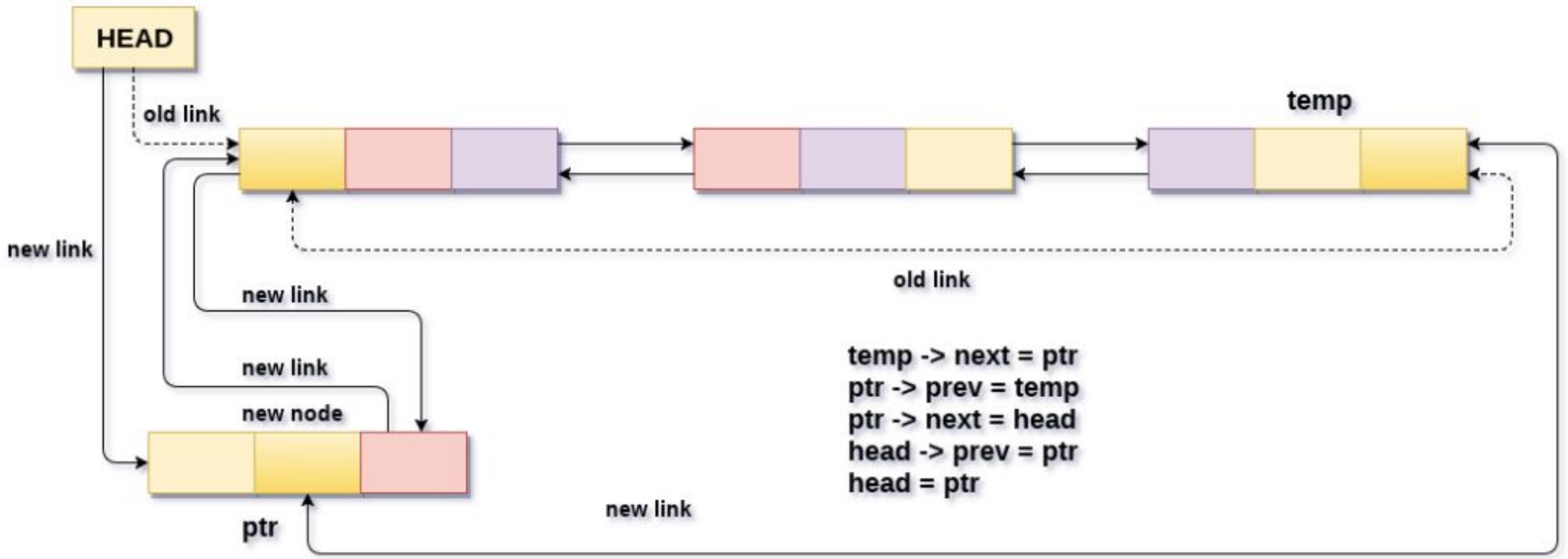
Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.





```
struct node {
    int data;
    struct node *next;
    struct node *prev;
}
```

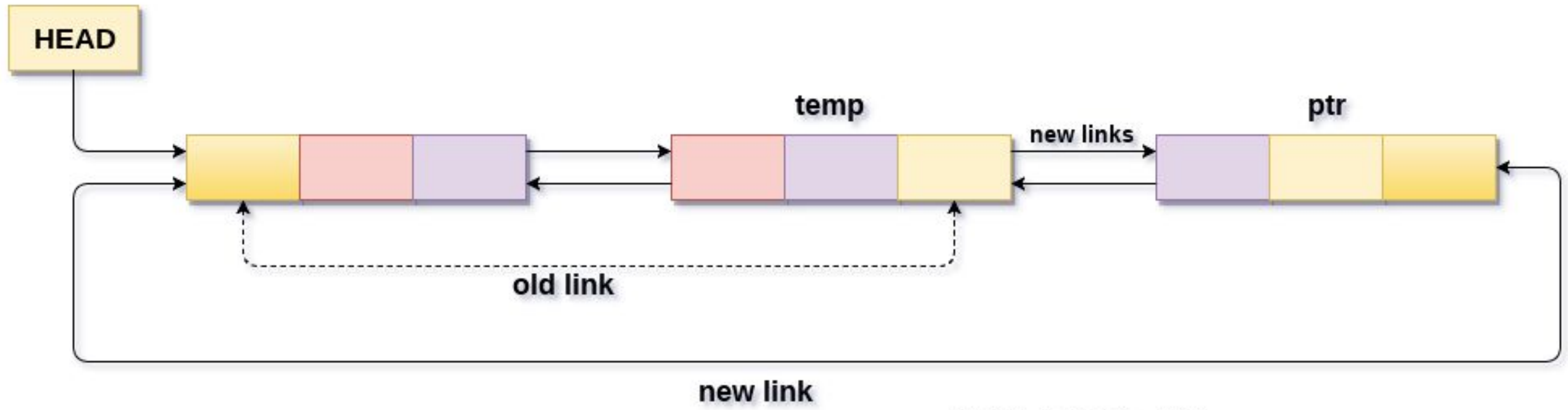
Circular Doubly Linked List - Insertion



temp -> next = ptr
ptr -> prev = temp
ptr -> next = head
head -> prev = ptr
head = ptr

Insertion into circular doubly linked list at beginning

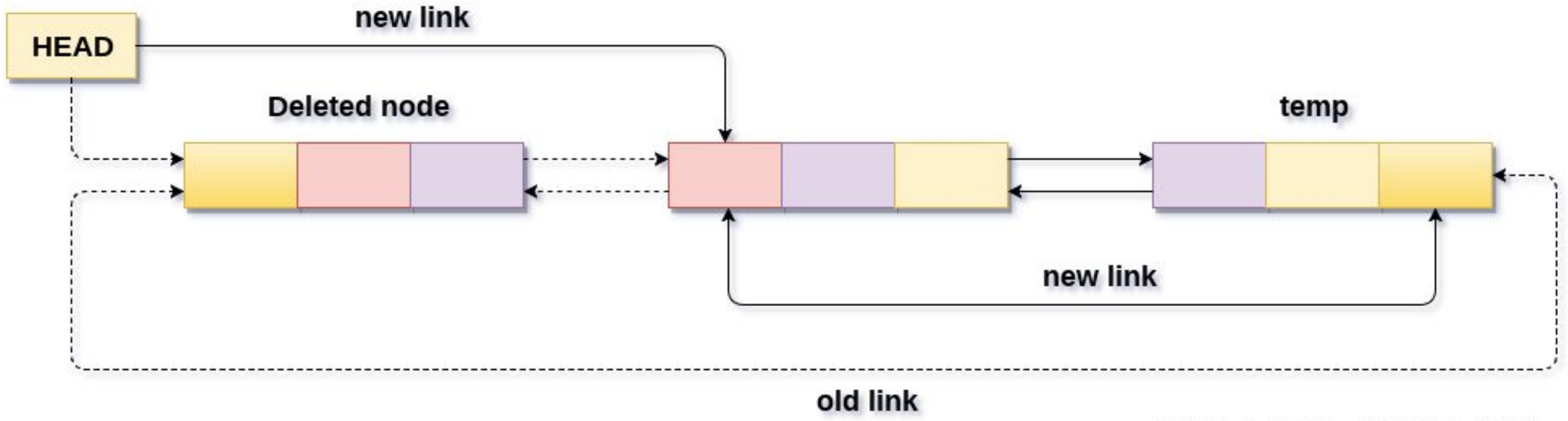
Circular Doubly Linked List - Insertion



temp -> next = ptr
ptr -> prev = temp
head -> prev = ptr
ptr -> next = head

Insertion into circular doubly linked list at end

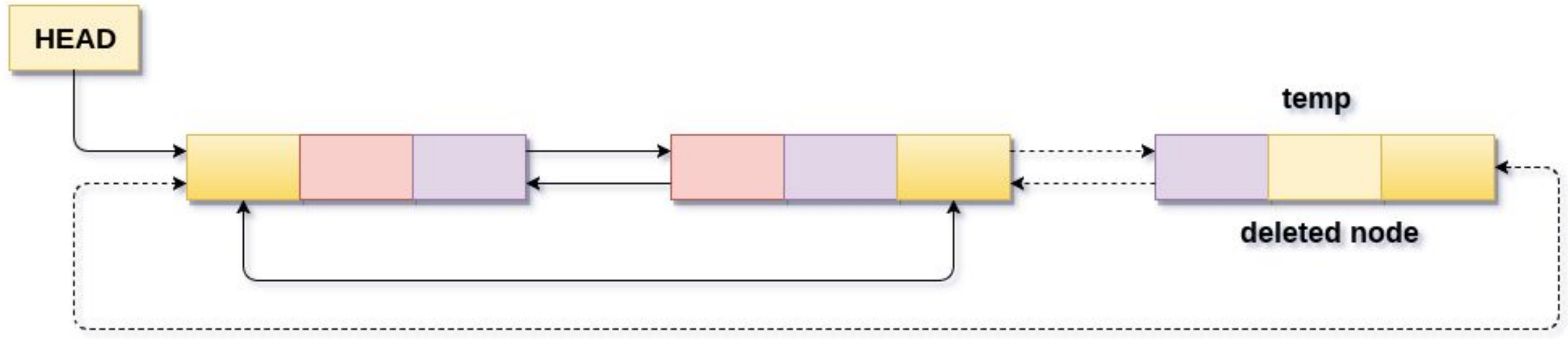
Circular Doubly Linked List - Deletion



temp -> next = head -> next
head -> next -> prev = temp
free head
head = temp -> next

Deletion in circular doubly linked list at beginning

Circular Doubly Linked List - Deletion



```
temp -> prev -> next = HEAD
HEAD -> prev = temp -> prev
free temp
```

Deletion in circular doubly linked list at beginning