Unit 3 – Linked List

A linked list is a data structure that plays a crucial role in data organization and management. It contains a series of nodes that are stored at random locations in memory, allowing for efficient memory management. Each node in a linked list contains two main components: the data part and a reference to the next node in the sequence.

Why Linked Lists?

Linked lists were created to overcome various drawbacks associated with storing data in regular lists and arrays, as outlined below:

- **Ease of insertion and deletion**

In lists, inserting or deleting an element at any position other than the end requires shifting all the subsequent items to a different position. This process has a time complexity of O(n) and can significantly degrade performance, especially as the list size grows..

Linked lists, however, operate differently. They store elements in various, non-contiguous memory locations and connect them through pointers to subsequent nodes. This structure allows linked lists to add or remove elements at any position by simply modifying the links to include a new element or bypass the deleted one.

Once the position of the element is identified and there is direct access to the point of insertion or deletion, adding or removing nodes can be achieved in O(1) time.

- **Dynamic size**

Python lists are dynamic arrays, which means that they provide the flexibility to modify size.

However, this process involves a series of complex operations, including reallocating the array to a new, larger memory block. Such reallocation is inefficient since elements are copied over to a new block, potentially allocating more space than is immediately necessary.

In contrast, linked lists can grow and shrink dynamically without the need for reallocation or resizing. This makes them a preferable option for tasks that require high flexibility.

- **Memory efficiency**

Lists allocate memory for all of its elements in a contiguous block. If a list needs to grow beyond its initial size, it must allocate a new, larger block of contiguous memory and

then copy all existing elements to this new block. This process is time-consuming and inefficient, especially for large lists. On the other hand, if the initial size of the list is overestimated, the unused memory is wasted.

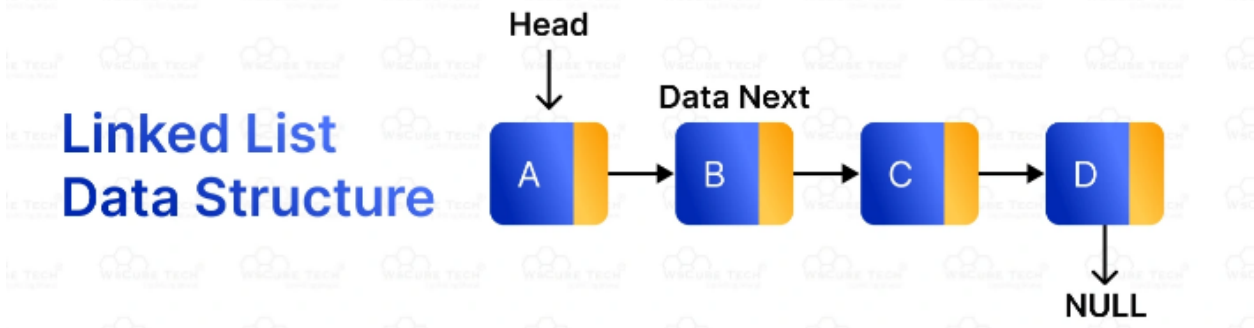
In contrast, linked lists allocate memory for each element separately. This structure leads to better memory utilization since memory for new elements can be allocated as they are added.

When Should You Use Linked Lists?

The choice between using a linked list or an array depends on the specific needs of the application. Linked lists are most useful when:

- You need to frequently insert and delete many elements
- The data size is unpredictable or likely to change frequently
- Direct access to elements is not a requirement
- The dataset contains large elements or structures

Definition : A linked list is a type of linear data structure similar to arrays. It is a collection of nodes that are linked with each other. A node contains two things first is data and second is a link that connects it with another node. Below is an example of a linked list with four nodes and each node contains character data and a link to another node.



Structure or Representation of Linked List

Node:

Each element of a linked list is called node. A node has two components:

- Data: The actual value or information that needs to be stored.
- Next: A pointer (or reference) to the next node in the linked list.

Head:

The first node in a linked list is called the head. It is the entry point to the list and used as a reference point to traverse it.

Null:

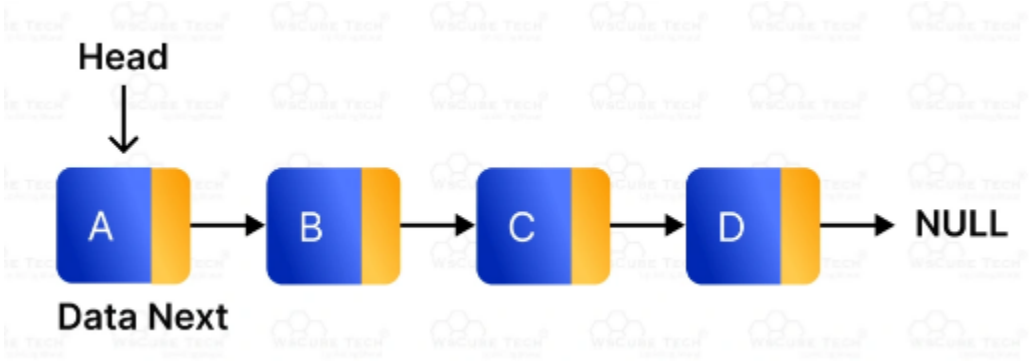
The last node of a linked list, which points to null, indicating the end of the list.

Types of Linked List in Data Structures

There are three primary types of linked lists:

1. Singly Linked List

A **singly linked list** is a collection of nodes where each node contains data and a reference to the next node in the sequence. This structure allows for a simple, one-directional traversal from the head to the end of the list.



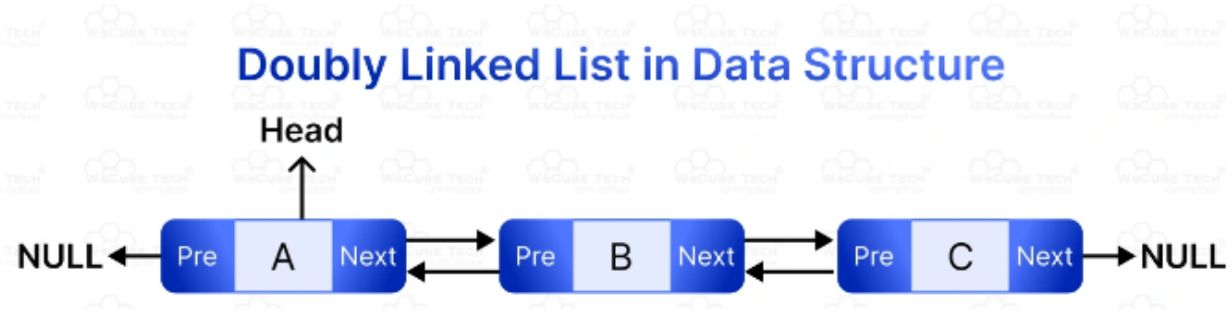
Example:
Consider managing a simple task list. Each task points to the next one until the end:

Task1 -> Task2 -> Task3 -> null

Operations like adding or removing tasks involve adjusting the 'next' reference of the preceding node, which is straightforward but only allows moving forward through the list.

2. Doubly Linked List

Doubly linked lists expand on singly linked lists by having each node contain two references: one pointing to the next node and one to the previous node. This allows nodes to be traversed in both forward and backward directions.



Example:
Imagine a playlist where you can move to the next or previous song:

null <- PrevSong <-> CurrentSong <-> NextSong -> null

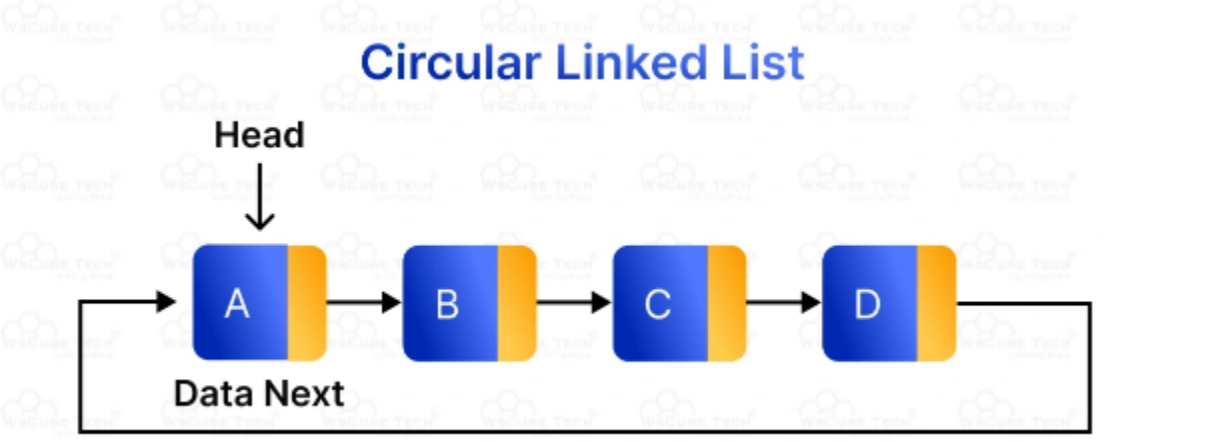
This setup is particularly useful for applications like image viewers or document viewers where users may need to go forward and backward between images or pages.

3. Circular Linked List

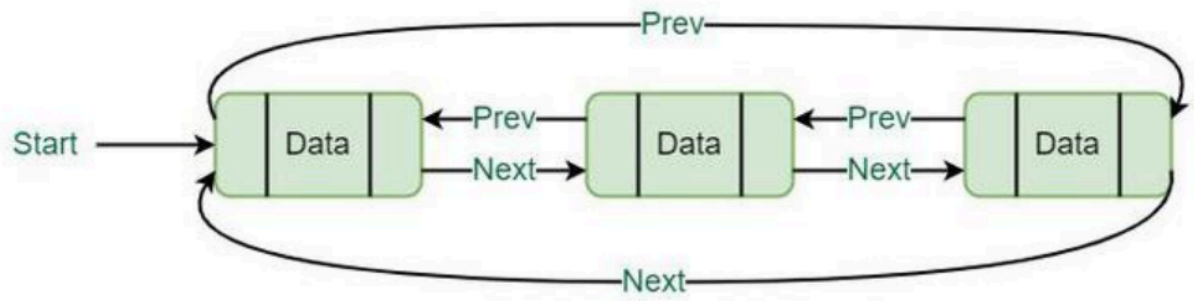
A **circular linked list** is a **type of data structure** where each node points to the next node, and the last node points back to the first, forming a circle.

This setup allows for an endless cycle of node traversal, which can be particularly useful in applications where the end of the list naturally reconnects to the beginning.

Singly Circular :



Doubly Circular :



Circular doubly linked list

Example:
In multiplayer games, a circular linked list manages player turns, cycling back to the first player after the last one finishes.

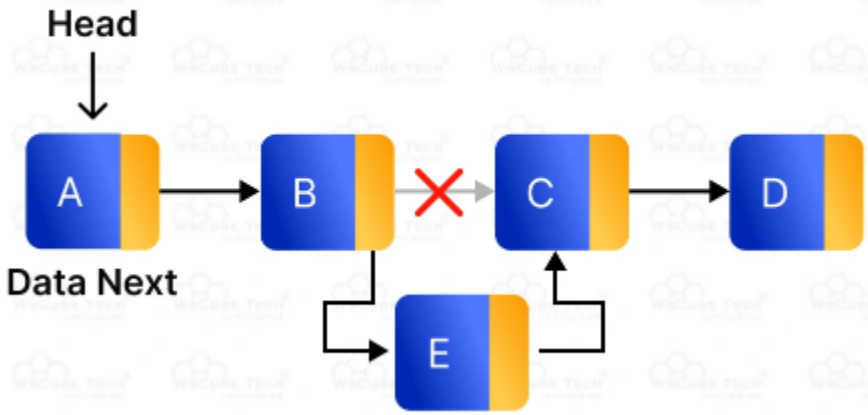
Linked List Operations

Understanding how to perform operations on linked lists is crucial for using them effectively in programming.

1. Insertion

Insertion in a linked list includes adding a new node to the list. You can insert a node at the beginning, in the middle, or at the end of the list.

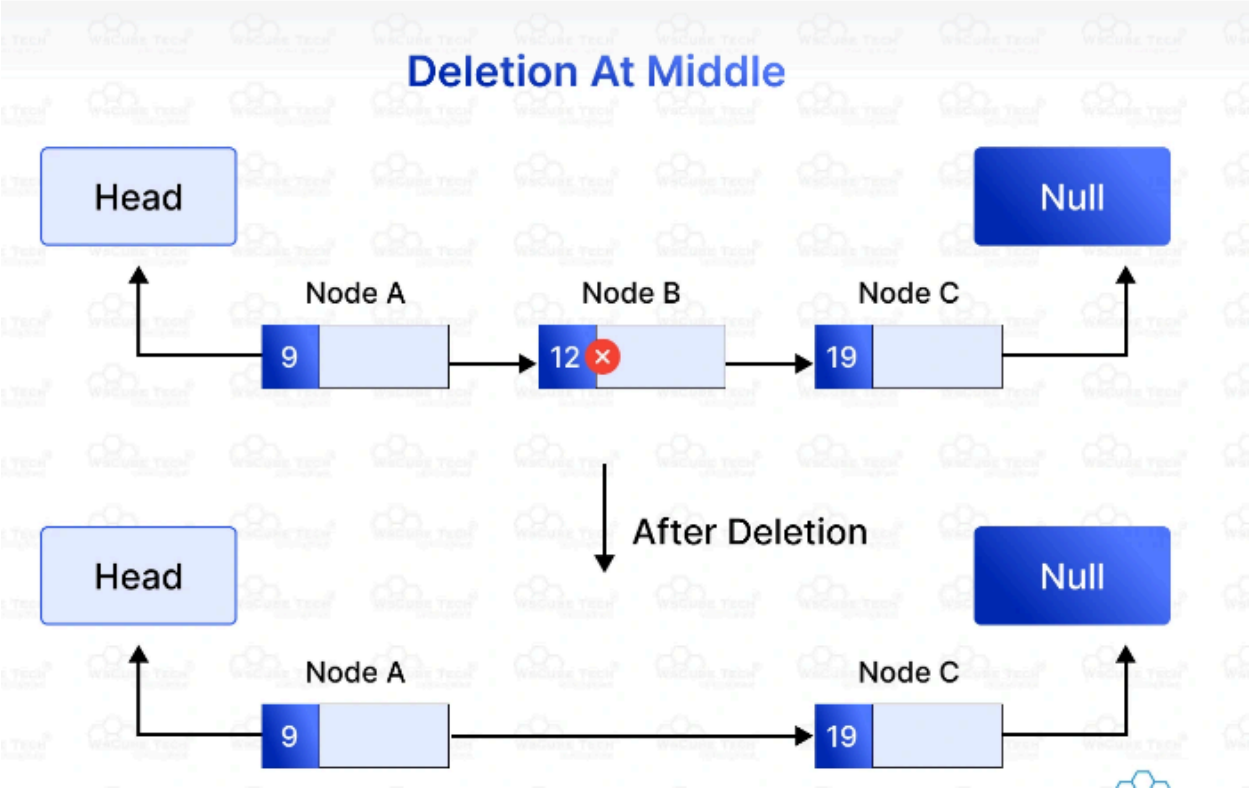
Insertion in a linked list



2. Deletion

Deletion removes a node from the linked list. This can also be done at the beginning, at a specific position, or at the end.

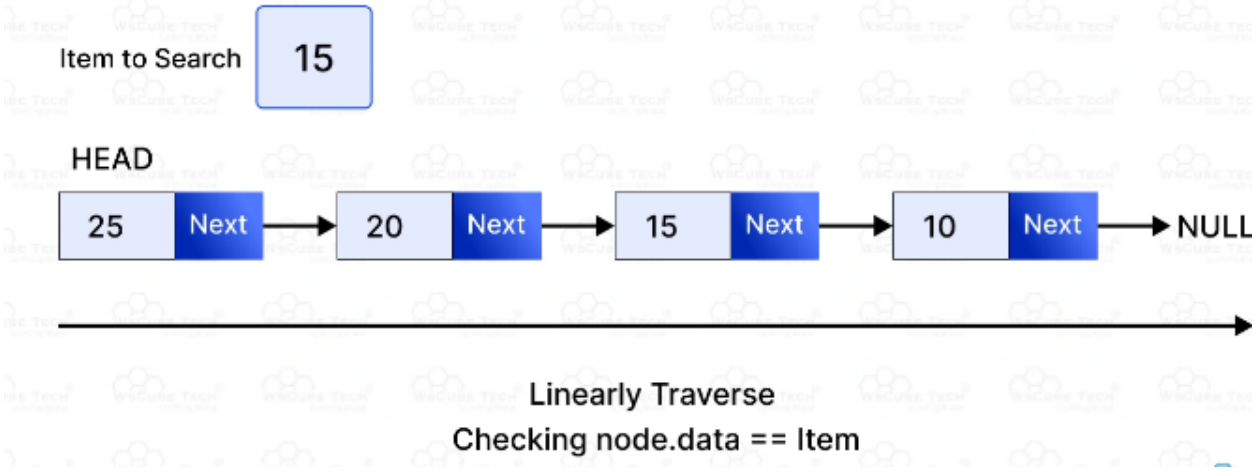
Deletion at middle



3. Search

Searching in a linked list involves traversing through the list from the head to find a node containing a specific value.

Searching in a linked list



4. Traversal

Traversal means going through each node in the list from the beginning to the end to access or modify data.

5. Update

Updating involves changing the data in one of the nodes without altering the structure of the list.

Advantages of Linked Lists

- **Dynamic Sizing:** Linked lists can grow and shrink during runtime as they do not have a fixed size, unlike arrays.
- **Efficient Insertions and Deletions:** Adding or removing nodes does not require the elements to be contiguous in memory, so operations can be performed without reallocating or reorganizing the entire structure.
- **No Memory Waste:** Since there's no need to pre-allocate memory, linked lists can manage data more efficiently without wasting memory on unused space.
- **Ease of Implementation:** Can easily be implemented and manipulated, particularly when it comes to complex data structures such as stacks, queues, and other abstract data types.
- **Flexible Structure:** Nodes can easily be rearranged by changing their next pointers without the need for data movement, which is beneficial for applications that require frequent reordering of data.

Disadvantages of Linked Lists (Limitations)

- **Higher Memory Use:** Each node in a linked list requires additional memory for a pointer to the next (and possibly previous) node, which is more memory-intensive than the tight data packing in arrays.

- **Slower Access Time:** Direct access is not feasible with linked lists. To access an element at a specific index, you have to traverse the list from the head to that position, which takes more time than array access.
- **Complexity in Navigation:** Navigating a linked list can be more complex compared to navigating an array, as it requires pointer manipulation, which can be prone to errors like pointer corruption or memory leaks.
- **Extra Memory for Pointers:** The requirement for storing pointers typically means using extra memory, which can be significant in systems with a large number of elements.

Array vs Linked List Difference

Let's know about the difference between linked list and **array in data structures**:

Feature	**Linked List**	**Array**
Memory Allocation	Dynamic allocation. Nodes are allocated as needed.	Static or dynamic allocation but fixed size once created.
Element Access	Sequential access. Must traverse from the head.	Direct access via indices.
Insertion/Deletion	Fast operations as they only require pointer changes.	Slower operations, especially in the middle, because elements must be shifted.
Memory requirement	Less memory-efficient due to extra space for pointers.	More memory-efficient; only stores the data.
Implementation	More complex implementation, especially for beginners.	Simpler implementation and easier to manage.
Speed of Access	Slower to read due to traversal needs.	Fast and immediate access to any element.
Data Structure Size	Can grow or shrink dynamically without a high cost.	Requires resizing and copying to grow, which is costly.
Cache Locality	Poor cache locality due to non-contiguous storage.	Better cache locality due to contiguous memory layout.

Usage Scenarios	Good for scenarios with frequent insertions and deletions without a predetermined size of data.	Best when the size is known in advance and fast element access is needed.
Manipulation Overhead	Low overhead for adding/removing elements.	Higher overhead for adding/removing elements due to shifting.
Structural Overhead	Nodes carry extra overhead due to storing pointers.	No extra overhead beyond the data itself.

Uses of Linked Lists

Data structure linked lists offer several practical uses across different applications:

1. Dynamic Memory Allocation

Linked lists are ideal for applications where the amount of data is unknown in advance and can change dynamically. They are extensively used in implementing memory management algorithms.

2. Implementing Stacks and Queues

Linked lists provide a natural way to implement other abstract data types like stacks and queues.

For example, a singly linked list can serve as a stack or queue by adding or removing nodes from one end.

3. Browser History Management

The functionality of back and forward navigation in web browsers can be effectively managed using a doubly linked list where each node points to the previous and next web pages visited.

4. Undo Functionality in Applications

Linked lists are used in implementing undo mechanisms in software applications, where operations are stored in a list and can be reversed by traversing backwards.

5. Music Playlists and Image Viewers

Applications that require sequential access to elements, such as media players or image viewers, often use doubly linked lists to facilitate easy navigation between items (next and previous).

6. Graphs Representation

Adjacency lists, which are used to represent graphs, can be implemented using linked lists, where each node represents a vertex and its linked list represents the adjacent vertices.

7. Polynomial Arithmetic

Linked lists are useful in representing and manipulating polynomials. Each node can represent a term of the polynomial, allowing for dynamic adjustments to the polynomial as terms are added or subtracted.

Singly Linked List using Python

Below is a detailed explanation with Python code and diagrammatic representations of operations on a singly linked list:

---

1. Creating a Node in a Singly Linked List

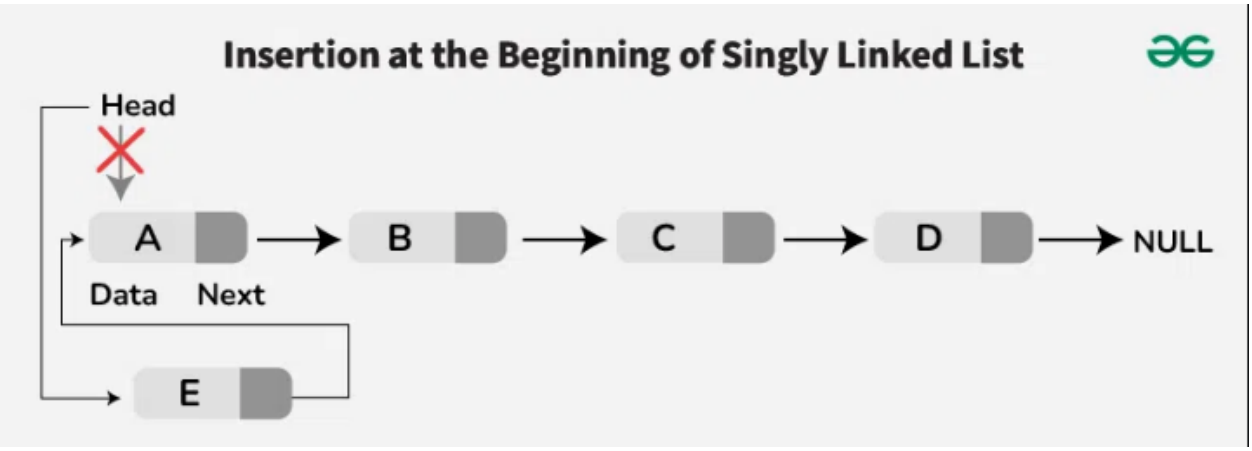
A node consists of two parts:

1. **Data**: Stores the value.
2. **Next**: Stores the reference to the next node.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

2. Insert Node at the Beginning

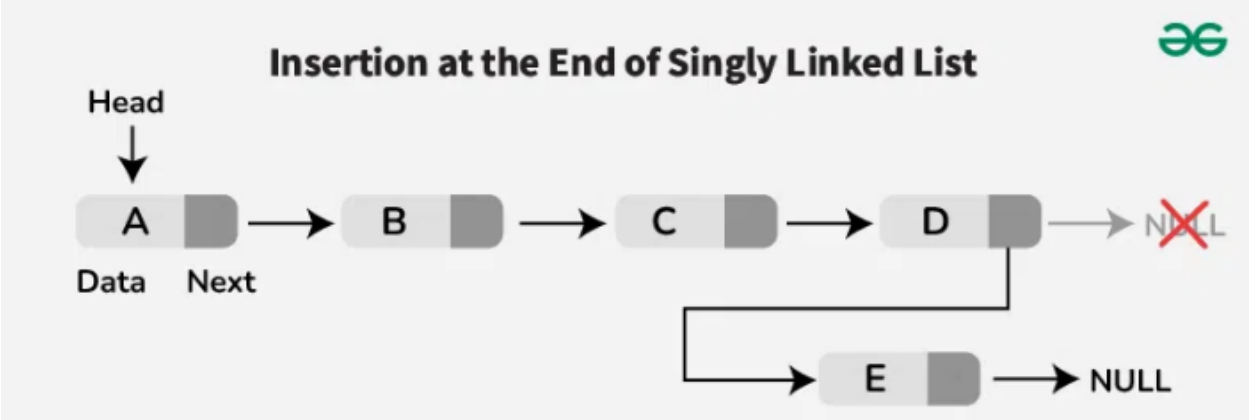
- Create a new node.
- Point the new node's `next` to the current head.
- Update the head to the new node.




```
class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
```

Insert Node at the End

- Traverse to the last node.
- Update the last node's `next` to the new node.

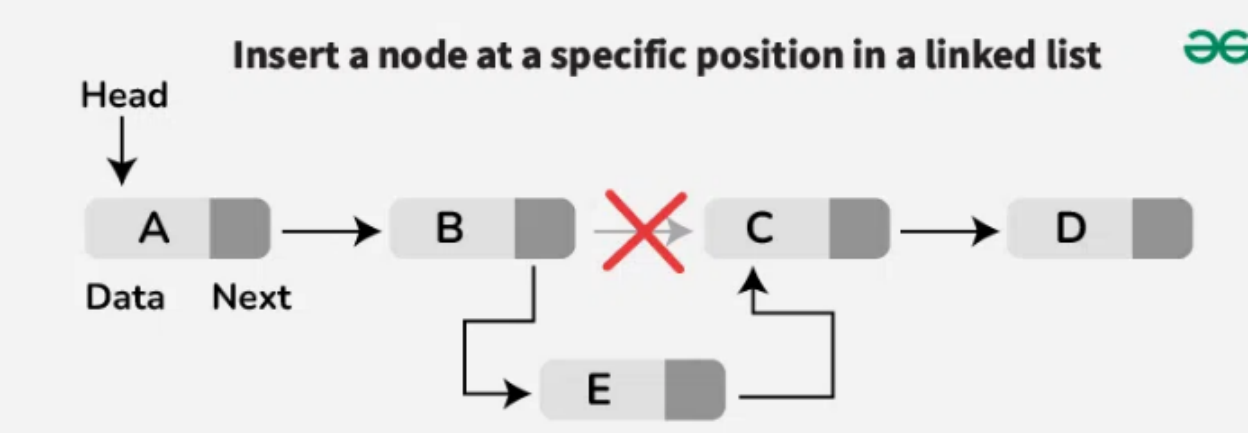



```
    def insert_at_end(self, data):
```

```
new_node = Node(data)
if self.head is None:
    self.head = new_node
    return

current = self.head
while current.next:
    current = current.next
current.next = new_node
```

Insert Node at Any Position

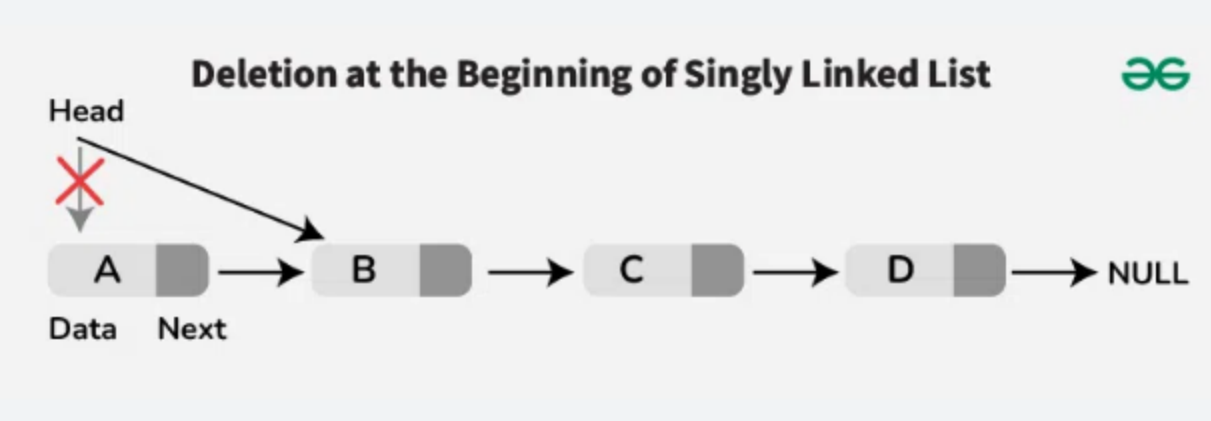
- Traverse to the node just before the desired position.
- Point the new node's `next` to the `next` of the previous node.
- Update the previous node's `next` to the new node.




```
def insert_at_position(self, data, position):
    if position == 0:
        self.insert_at_beginning(data)
        return

    new_node = Node(data)
    current = self.head
    for _ in range(position - 1):
        if current is None:
            raise IndexError("Position out of range")
        current = current.next

    new_node.next = current.next
    current.next = new_node
```

. Delete the First Node

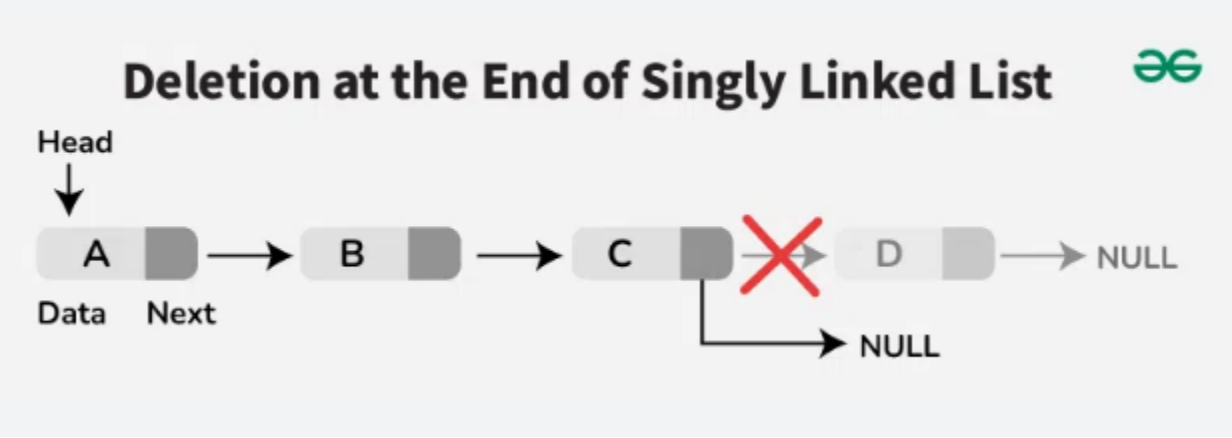
- Update the head to the next node.




```
def delete_first(self):
    if self.head is None:
        print("List is empty")
        return
    self.head = self.head.next
```

Delete the Last Node

- Traverse to the second-last node.
- Set its `next` to `None`.

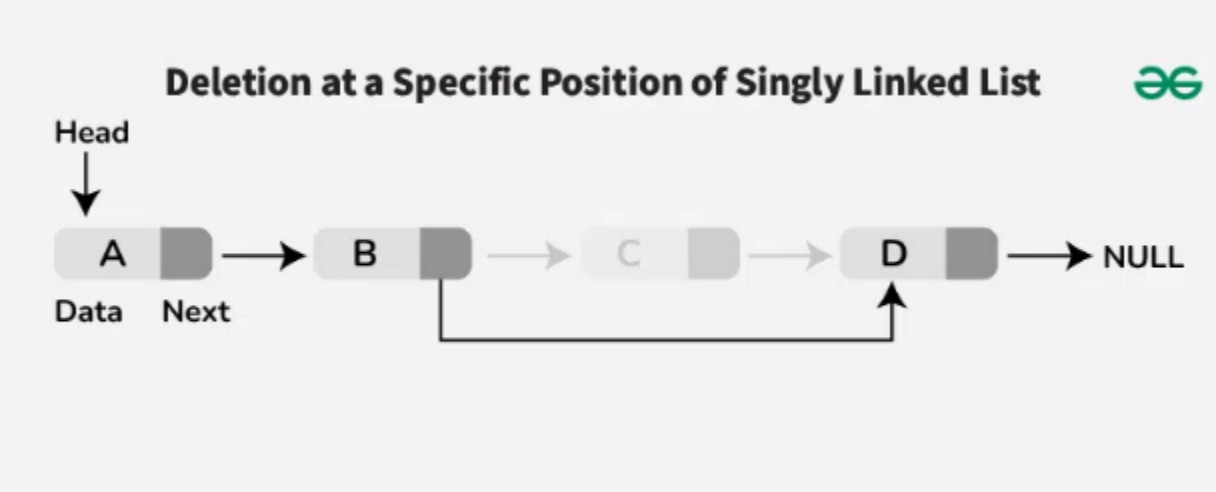



```
def delete_last(self):
    if self.head is None:
        print("List is empty")
        return
    if self.head.next is None:
```

```
        self.head = None
        return

    current = self.head
    while current.next and current.next.next:
        current = current.next
    current.next = None
```

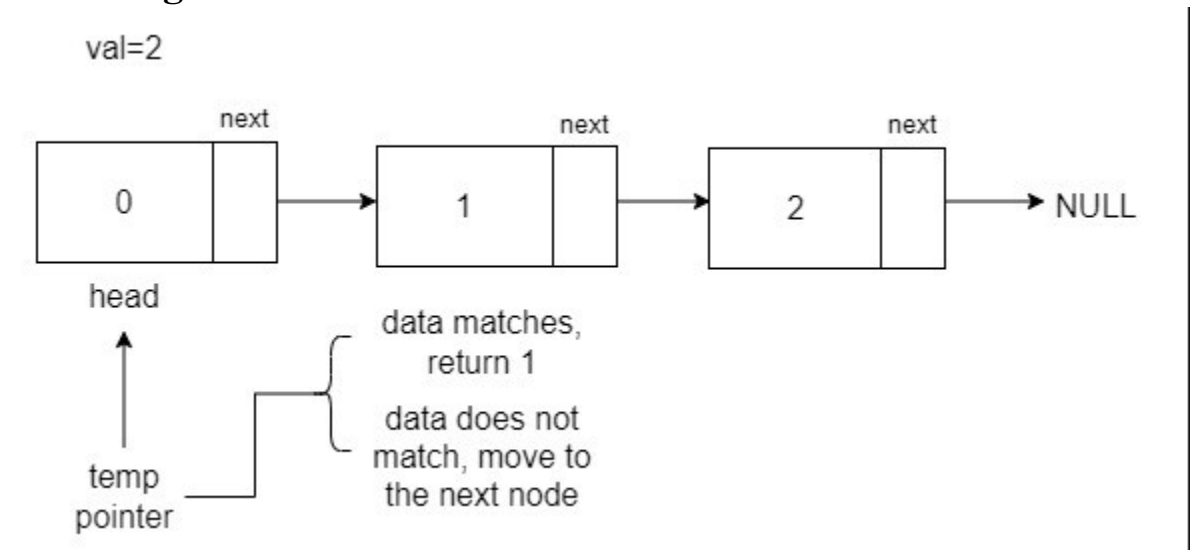
Delete a Node by Searching Data

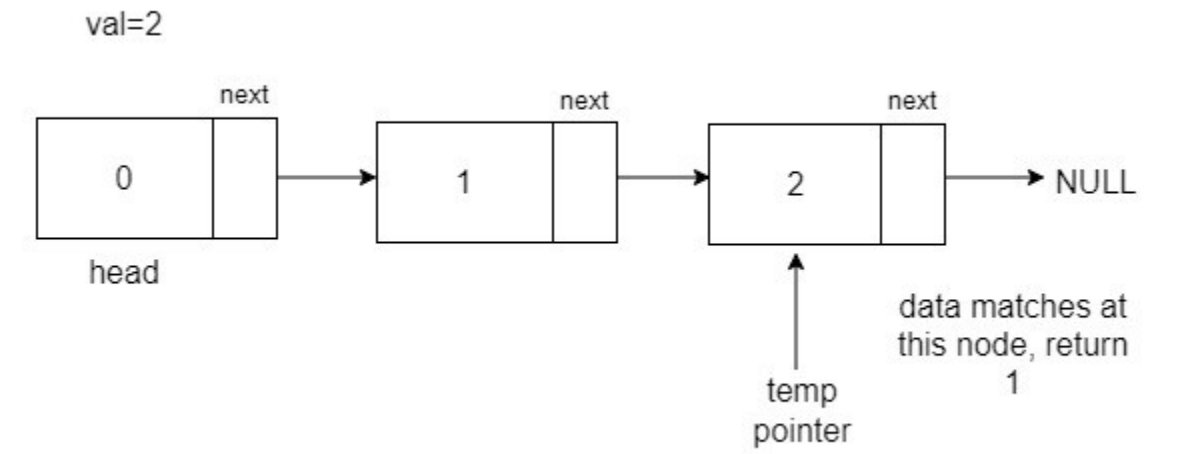
- Traverse to find the node with the target data.
- Update the previous node's `next` to skip the target node.




```
def delete_by_value(self, value):
    if self.head is None:
        print("List is empty")
        return

    if self.head.data == value:
        self.head = self.head.next
        return

    current = self.head
    while current.next and current.next.data != value:
        current = current.next

    if current.next is None:
        print("Value not found in the list")
        return

    current.next = current.next.next
```

Searching Element in list

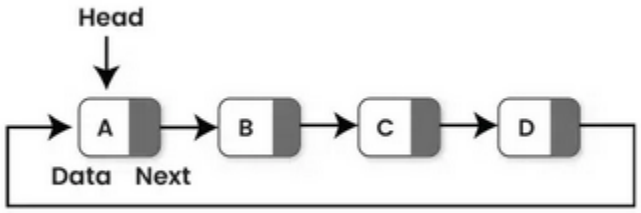





```
def search(self, key):
        current = self.head
        while current:
            if current.data == key:
                return True
            current = current.next
        return False
```

Singly Circular Linked List:

A circular linked list is a type of linked list in which the last node of the list points back to the first node (head), forming a loop or circle. Unlike a linear linked list, where the last node points to **NULL**, in a circular linked list, the next pointer of the last node points back to the first node.



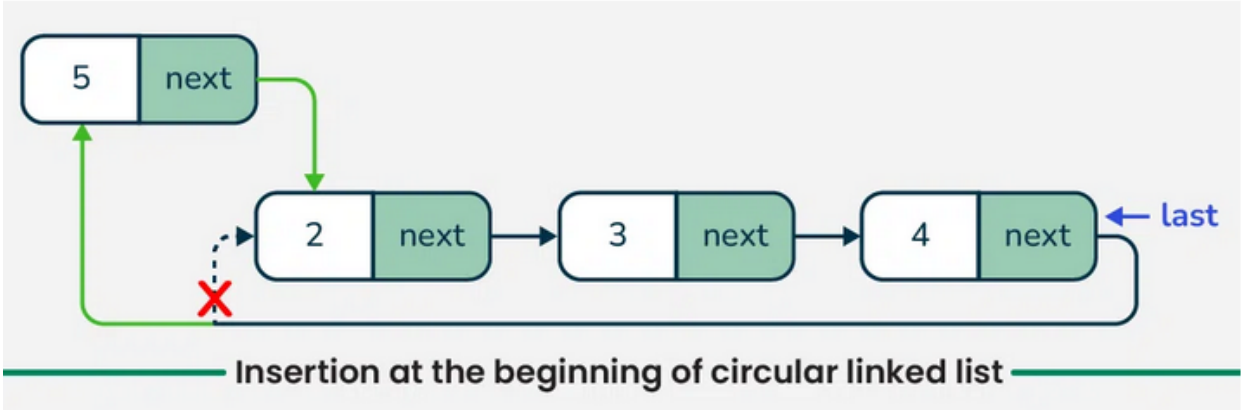
Circular linked lists can be singly linked or doubly linked, meaning each node may have one or two pointers respectively (one pointing to the next node and, in the case of doubly linked lists, another pointing to the previous node). They can be used in various scenarios, such as representing circular buffers, round-robin scheduling algorithms, and as an alternative to linear linked lists when operations involve wrapping around from the end to the beginning of the list.

Insertion in Circular Linked List in Python:

1. Insertion at the Beginning:

To insert a node at the beginning of a circular linked list, you need to follow these steps:

- Create a new node with the given data.
- If the list is empty, make the new node the head and point it to itself.
- Otherwise, set the next pointer of the new node to point to the current head.
- Update the head to point to the new node.
- Update the next pointer of the last node to point to the new head (to maintain the circular structure).



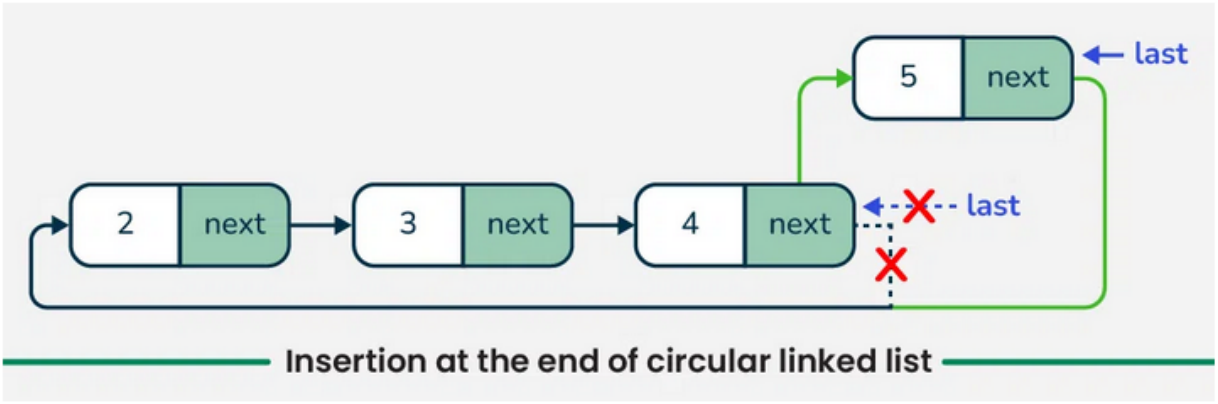

```
def insert_at_beginning(self, data):
    # Insert a new node with data at the beginning of the linked list
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        new_node.next = self.head
    else:
        new_node.next = self.head
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        temp.next = new_node
        self.head = new_node
```

Insertion at the End:

To insert a node at the end of a circular linked list, you need to follow these steps:

- Create a new node with the given data.

- If the list is empty, make the new node the head and point it to itself.
- Otherwise, traverse the list to find the last node.
- Set the next pointer of the last node to point to the new node.
- Set the next pointer of the new node to point back to the head (to maintain the circular structure).

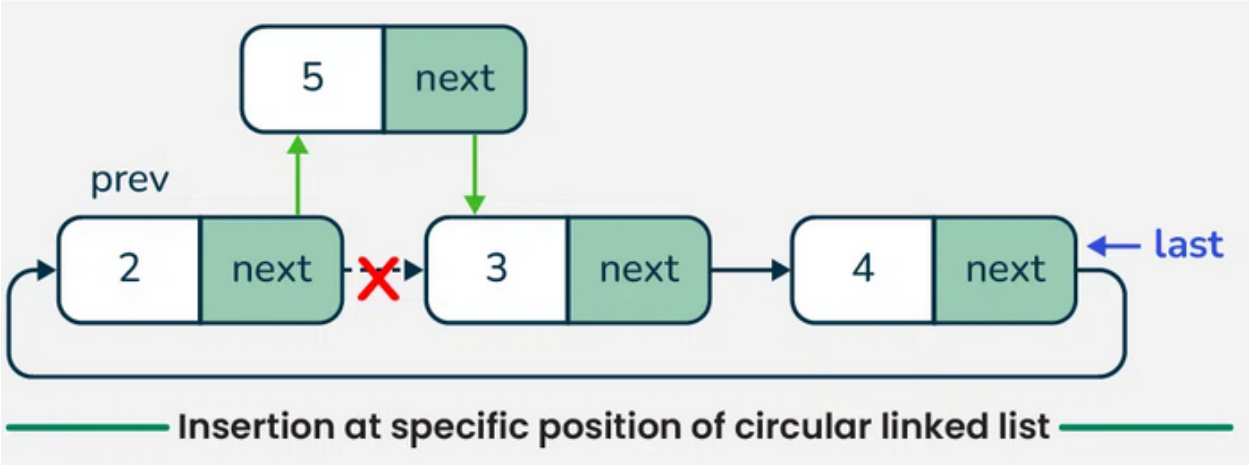



```
def InsertatEnd(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        new_node.next = self.head
        return
    current = self.head
    while current.next != self.head:
        current = current.next
    current.next = new_node
    new_node.next = self.head
```

Insertion at a Particular Position:

To insert a node at a particular position (index) in a circular linked list, you need to follow these steps:

- Create a new node with the given data.
- Traverse the list to find the node at the desired position (index - 1).
- Update the next pointer of the new node to point to the next node of the current node.
- Update the next pointer of the current node to point to the new node.




```
def insert_at_position(self, position, data):
    if position <= 0:
        print("Position should be greater than 0.")
        return
    new_node = Node(data)
    if position == 1:
        self.insert_first(data)
        return
    current = self.head
    for _ in range(position - 2):
        if current.next == self.head:
            print("Position out of bounds.")
            return
```

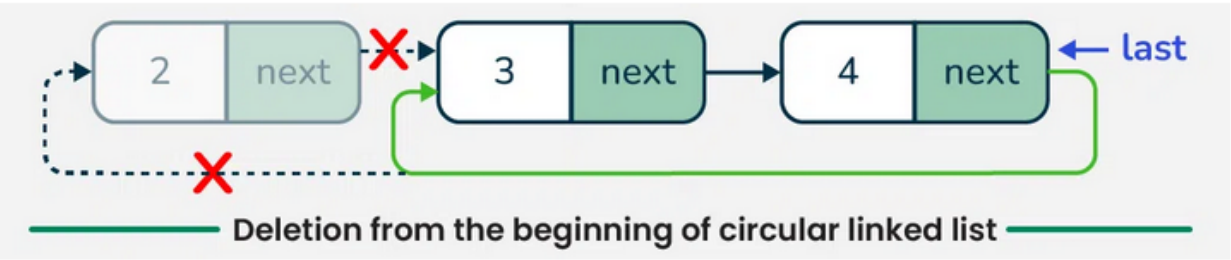
```
        current = current.next
    new_node.next = current.next
    current.next = new_node
```

Deletion in Circular Linked List in Python:

1. Deletion at the beginning of the circular linked list:

To delete the node at the beginning of a circular linked list in Python, you need to follow these steps:

- Check if the list is empty. If it is empty, there is nothing to delete.
- If the list has only one node, set the head to None to delete the node.
- Otherwise, find the last node of the list (the node whose next pointer points to the head).
- Update the next pointer of the last node to point to the second node (head's next).
- Update the head to point to the second node.
- Optionally, free the memory allocated to the deleted node.



Delete the first node in circular linked list

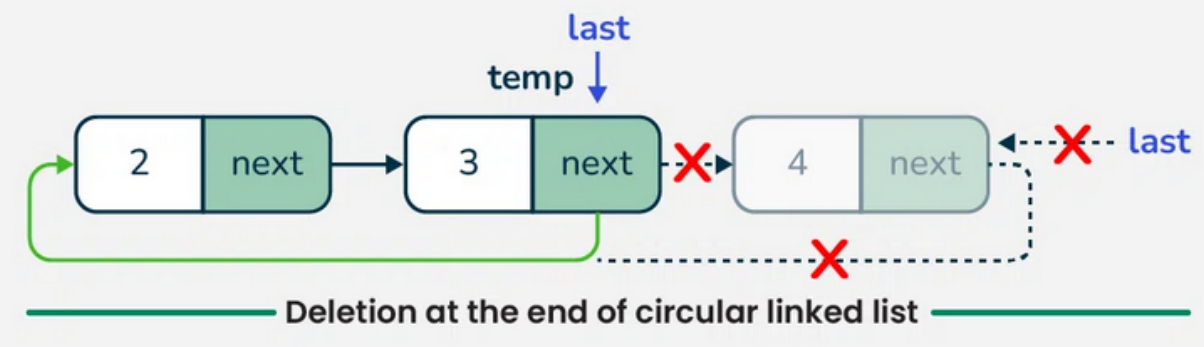
```
def delete_first(self):
    if not self.head:
        print("List is empty. Cannot delete.")
        return
```

```
self.head = self.head.next
```

Deletion at the end of a Circular Linked List:

To delete the node at the beginning of a circular linked list in Python, you need to follow these steps:

- Check if the list is empty. If it is empty, there is nothing to delete.
- If the list has only one node, set the head to None to delete the node.
- Otherwise, find the last node of the list (the node whose next pointer points to the head).
- Update the next pointer of the last node to point to the second node (head's next).
- Update the head to point to the second node.
- Optionally, free the memory allocated to the deleted node.




```
def delete_last(self):
    if not self.head:
        print("List is empty. Cannot delete.")
        return
    if not self.head.next:
        self.head = None
```

```
        return
    current = self.head
    while current.next and current.next.next:
        current = current.next
    current.next = None
```

Delete a specific node in circular linked list

To delete a specific node from a circular linked list, we first check if the list is empty.

If it is then we print a message and return **nullptr**.

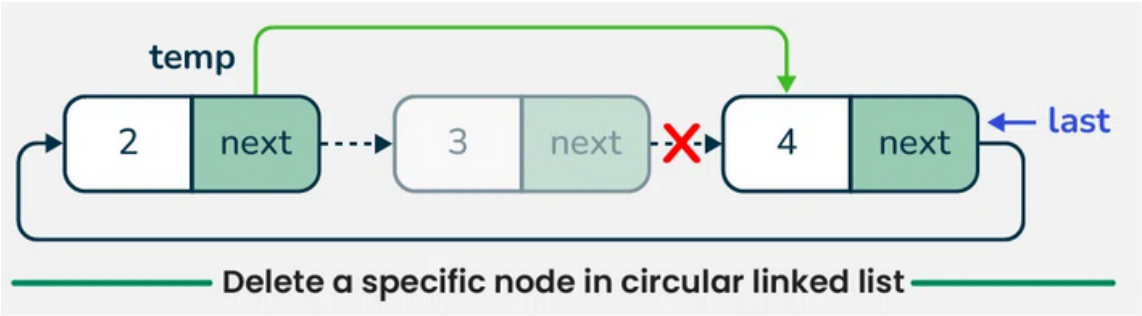
If the list contains only one node and it matches the **key** then we delete that node and set **last** to **nullptr**.

If the node to be deleted is the first node then we update the **next** pointer of the **last** node to skip the **head** node and delete the **head**.

For other nodes, we traverse the list using two pointers: **curr** (to find the node) and **prev** (to keep track of the previous node).

If we find the node with the matching key then we update the next pointer of **prev** to skip the **curr** node and delete it.

If the node is found and it is the last node, we update the **last** pointer accordingly. If the node is not found then do nothing and **tail** or **last** as it is. Finally, we return the updated **last** pointer.



Delete a specific node in circular linked list

```
def delete_by_value(self, value):
    if not self.head:
        print("List is empty. Cannot delete.")
        return
```

```
    if self.head.data == value:
        self.head = self.head.next
        return
    current = self.head
    while current.next and current.next.data != value:
        current = current.next
    if current.next:
        current.next = current.next.next
    else:
        print(f"Value {value} not found in the list.")
```

Searching in Circular Linked list

Searching in a circular linked list is similar to searching in a regular linked list. We start at a given node and traverse the list until you either find the target value or return to the starting node. Since the list is circular, make sure to keep track of where you started to avoid an infinite loop.

```
  def search_by_value(self, value):
    if not self.head:
        print("List is empty.")
        return
    current = self.head
    position = 1
    while True:
        if current.data == value:
            print(f"Value {value} found at position {position}.")
            return
        current = current.next
```

```
        position += 1
        if current == self.head:
            break
    print(f"Value {value} not found in the list.")
```

Advantages of Circular Linked Lists

- In circular linked list, the last node points to the first node. There are no null references, making traversal easier and reducing the chances of encountering null pointer exceptions.
- We can traverse the list from any node and return to it without needing to restart from the head, which is useful in applications requiring a circular iteration.
- Circular linked lists can easily implement circular queues, where the last element connects back to the first, allowing for efficient resource management.
- In a circular linked list, each node has a reference to the next node in the sequence. Although it doesn't have a direct reference to the previous node like a doubly linked list, we can still find the previous node by traversing the list.

Disadvantages of Circular Linked Lists

- Circular linked lists are more complex to implement than singly linked lists.
- Traversing a circular linked list without a clear stopping condition can lead to infinite loops if not handled carefully.
- Debugging can be more challenging due to the circular nature, as traditional methods of traversing linked lists may not apply.

Applications of Circular Linked Lists

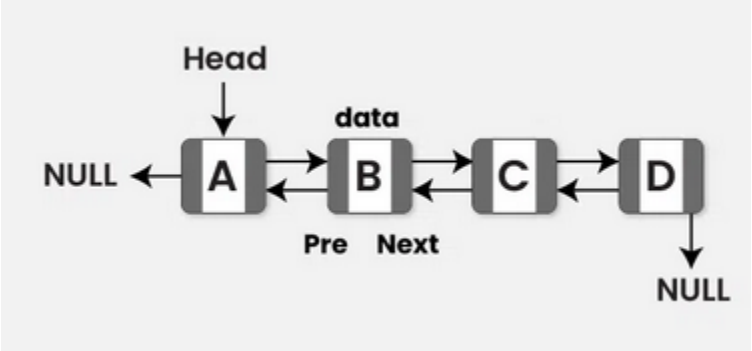
- It is used for time-sharing among different users, typically through a **Round-Robin scheduling mechanism.**
- In multiplayer games, a circular linked list can be used to switch between players. After the last player's turn, the list cycles back to the first player.
- Circular linked lists are often used in buffering applications, such as streaming data, where data is continuously produced and consumed.

- In media players, circular linked lists can manage playlists, this allowing users to loop through songs continuously.

- Browsers use circular linked lists to manage the cache. This allows you to navigate back through your browsing history efficiently by pressing the BACK button.

Doubly Linked List

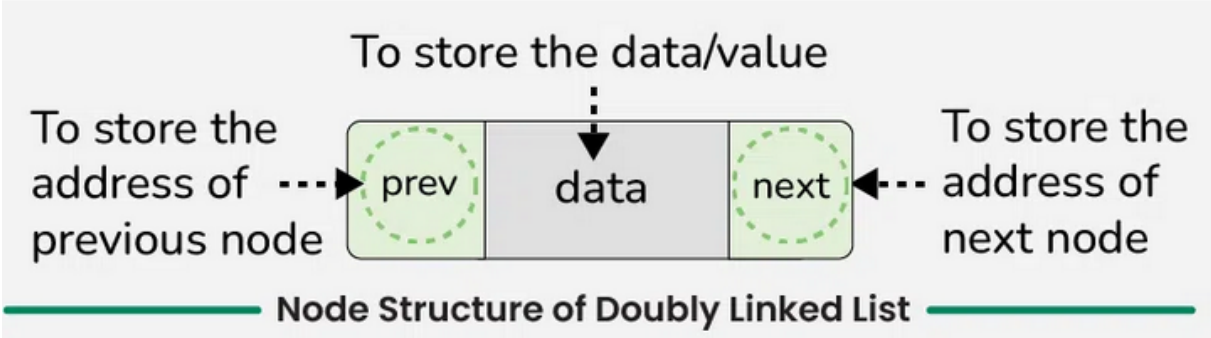
A **doubly linked list** is a data structure that consists of a set of nodes, each of which contains a **value** and **two pointers**, one pointing to the **previous node** in the list and one pointing to the **next node** in the list. This allows for efficient traversal of the list in **both directions**, making it suitable for applications where frequent **insertions** and **deletions** are required.



Representation of Doubly Linked List in Data Structure

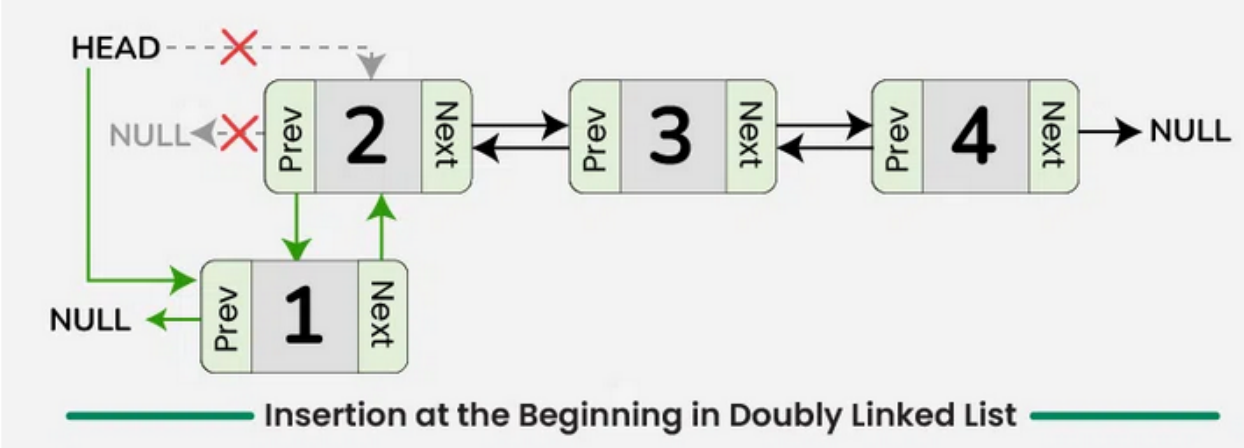
In a data structure, a doubly linked list is represented using nodes that have three fields:

1. Data
2. A pointer to the next node (**next**)
3. A pointer to the previous node (**prev**)




```
class Node:

    def __init__(self, data):
        # To store the value or data.
        self.data = data

        # Reference to the previous node
        self.prev = None

        # Reference to the next node
        self.next = None
```

Insertion at the Beginning in Doubly Linked List

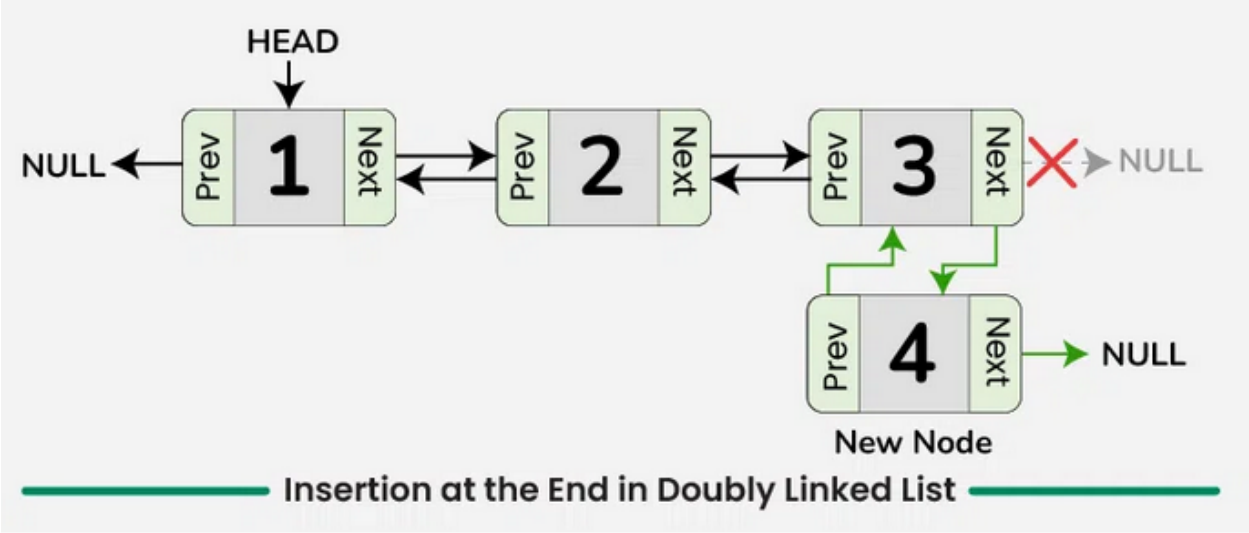


To insert a node at the beginning of a doubly linked list in Python, you need to follow these steps:

- Create a new node with the given data.
- Set the "**next**" pointer of the new node to point to the current head (if any).
- Set the "**previous**" pointer of the new node to None (as it will become the new head).
- If the list is not empty, update the "**previous**" pointer of the current head to point to the new node.
- Update the head of the list to point to the new node.

```
def insert_first(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        return
    new_node.next = self.head
    self.head.prev = new_node
    self.head = new_node
```

Insertion at the End of Doubly Linked List



To insert a node at the end of a doubly linked list in Python, you need to follow these steps:

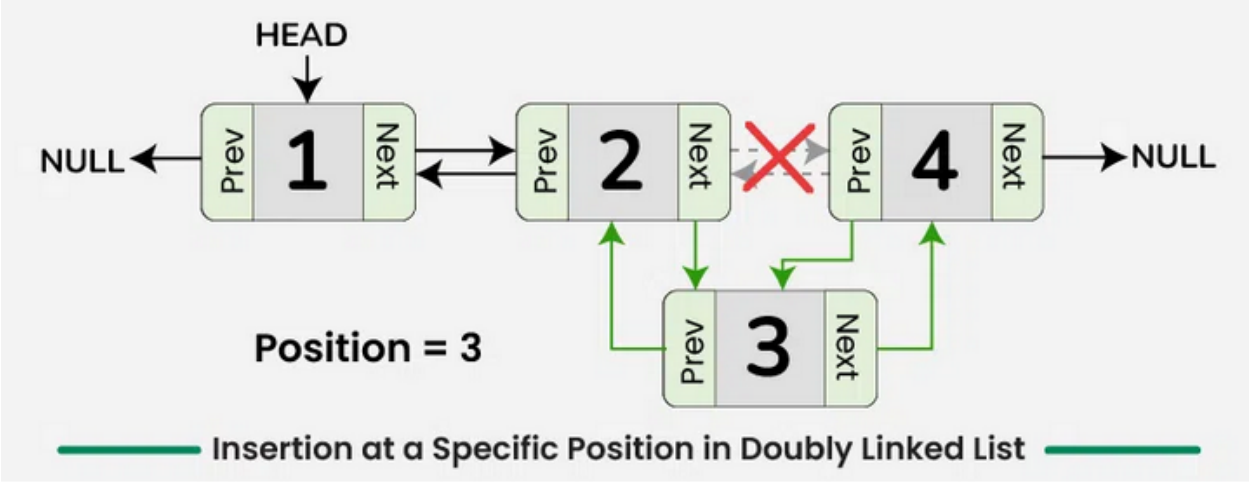
- Create a new node with the given data.
- If the list is empty (head is None), make the new node the head of the list.
- Otherwise, traverse the list to find the last node.
- Set the "**next**" pointer of the last node to point to the new node.
- Set the "**previous**" pointer of the new node to point to the last node.
- Optionally, update the head of the list to point to the new node if it's the first node in the list.

```
def insert_last(self, data):
    self.append(data)
def append(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
```

```
        return
    current = self.head
    while current.next:
        current = current.next
    current.next = new_node
    new_node.prev = current
```

Insertion at a Specific Position in Doubly Linked List

To insert a node at a specific Position in doubly linked list, we can use the following steps:



Insertion at a Specific Position in Doubly Linked List

to insert a new node at a specific position,

- If position = 1, create a new node and make it the head of the linked list and return it.
- Otherwise, traverse the list to reach the node at position – 1, say **curr**.
- If the position is valid, create a new node with given data, say **new_node**.
- Update the next pointer of new node to the next of current node and prev pointer of new node to current node, **new_node->next = curr->next** and **new_node->prev = curr.**
- Similarly, update next pointer of current node to the new node, **curr->next = new_node**.
- If the new node is not the last node, update prev pointer of new node's next to the new node, **new_node->next->prev = new_node**
- def insert_at_position(self, position, data):

```
        if position <= 0:
            print("Position should be greater than 0.")
            return
        new_node = Node(data)
        if position == 1:
            self.insert_first(data)
            return
        current = self.head
        for _ in range(position - 2):
            if not current:
                print("Position out of bounds.")
                return
            current = current.next
        if not current:
            print("Position out of bounds.")
            return
        new_node.next = current.next
        if current.next:
            current.next.prev = new_node
        current.next = new_node
        new_node.prev = current
```

```
current.next = new_node

new_node.prev = current

    new_node.next = current.next

    if current.next:

        current.next.prev = new_node

    current.next = new_node

    new_node.prev = current
```