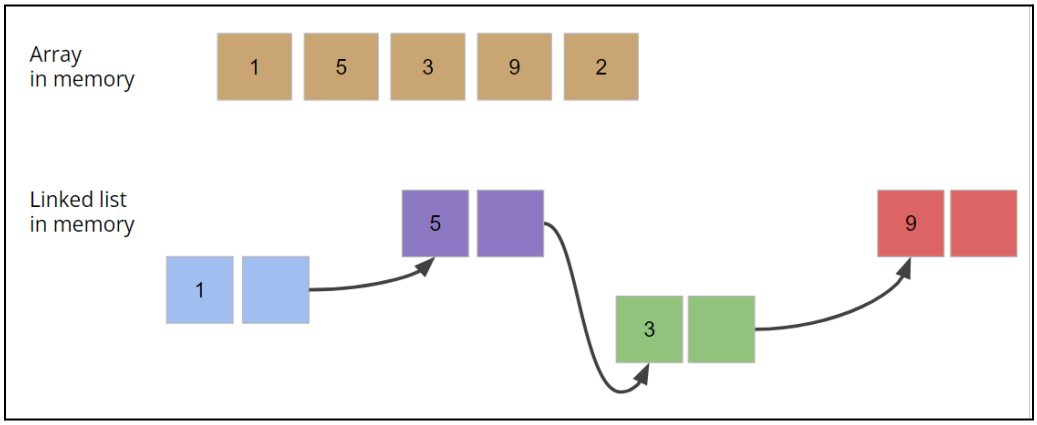
Linked List

Let's recap some of the disadvantages of an Array:

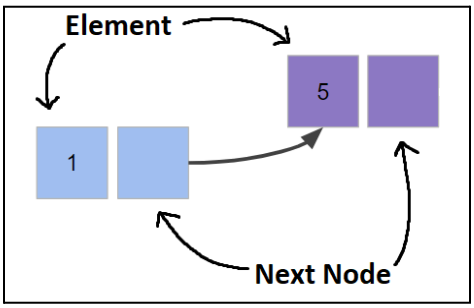
- **Fixed capacity:** Once created, an array cannot be resized. The only way to "resize" is to create a larger new array, copy the elements from the original array into the new one, and then change the reference to the new one.
- **Shifting elements in insert:** Since we do not allow gaps in the array, inserting a new element requires that we shift all the subsequent elements right to create an empty space where the new element is placed.
- **Shifting elements in removal:** Removing may also require shifting to fill up the empty space left by the removed element.

To solve these problems, Linked List comes to the rescue

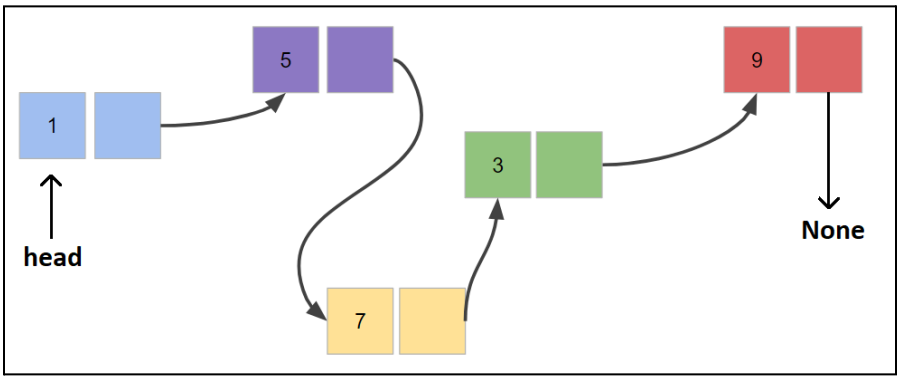


Definition: Linked List is another data structure which is a sequence of nodes connected by links. A node typically contains two fields:

- Element: Stores the value
- Next: Reference to the next node

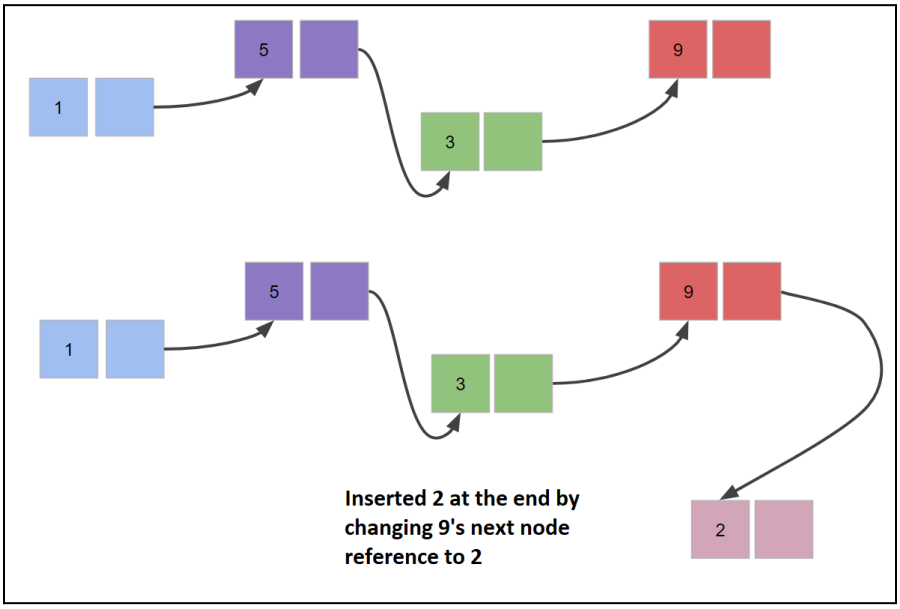


A "head" pointer is assigned to the first node. The next node reference of the last node is set to None

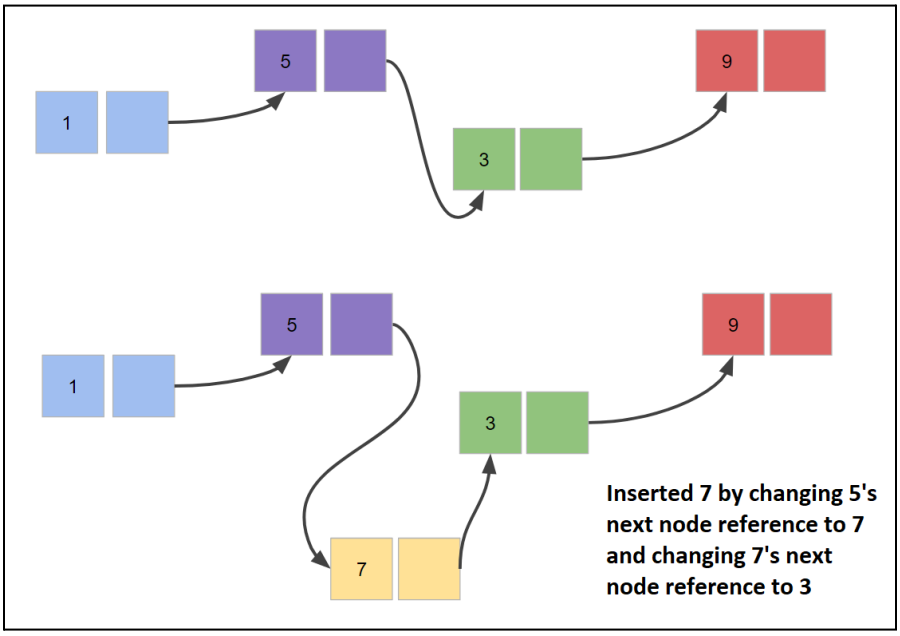


Now let's see how a linked list can easily solve some problems faced by arrays:

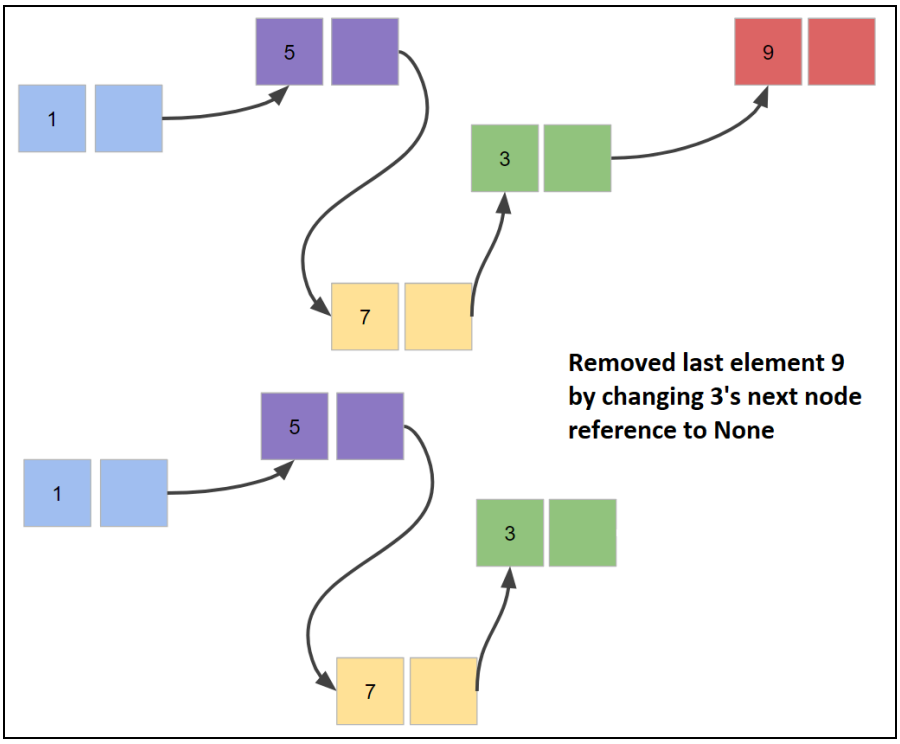
- Insert End:



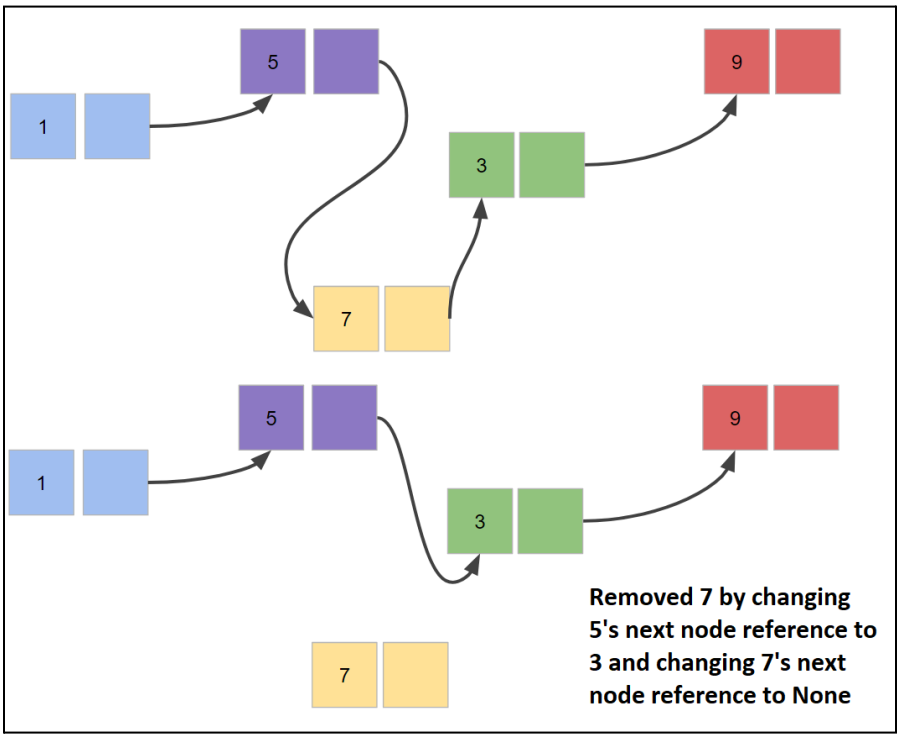
- Insert Anywhere Else:



- Remove End:



- Remove Anywhere Else:



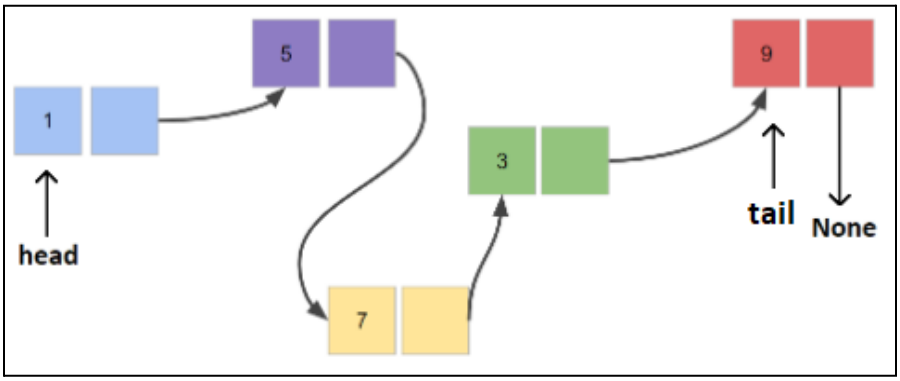
We can see how the links allow the insertion of new nodes anywhere in the list or the removal of an existing node from the list without having to disturb the rest of the list (the only nodes affected are the ones adjacent to the node being inserted or removed). Since we can extend the list one node at a time, we can also resize a list until we run out of resources.

Linked List Properties:

- **Nodes**: contains nodes, taking extra memory for the address of each node
- **Dynamic Size**: can grow and shrink in size by allocating or deallocating memory at runtime
- **Pointers**: contains head (and tail) pointers for traversal
- **No Random Access**: element cannot be accessed directly, it has to be iterated from the head node
- **Non-Contiguous Memory**: elements are placed in different places of memory and linked using addresses

- **Mutable**: elements and next node references of a link can be changed after initialization

Often, we use another pointer "tail" that points to the last element of the linked list. This pointer is generally used to traverse a list. We also sometimes name this pointer as "temp".



Linked List Declaration/Initialization:

```
#Node Class Design
class Node:
        def __init__(self, elem, next):
                self.elem = elem
                self.next = next
```

```
#Linked List from an Array
def list_from_array(arr):
        head = Node(arr[0], None)
        tail = head
        for i in range(1, len(arr)):
                n = Node(arr[i], None)
                tail.next = n
                tail = tail.next
        return head
```

Follow-Up Question:

Q) Why did we use tail.next = n and not head.next = n?

Ans: Because head is fixed in the first position whereas tail is iterating. Using tail, we are getting the previous node.

Linked List Operation:

- Iteration:

```
def iteration(head):
        temp = head
        while temp != None:
                print(temp.elem)
                temp = temp.next
```

- Count:

```
def count(head):
        count = 0
        temp = head
        while temp != None:
                count += 1
                temp = temp.next
        return count
```

- Retrieve element using index

```
def retrieve_elem(head, idx):
        count = 0
        temp = head
        result = None
        while temp != None:
                if count == idx:
                        result = temp.elem
                        break
                temp = temp.next
                count += 1
        if result == None:
                return "Invalid Index"
        else:
                return result
```

- Retrieve node using index

```
def retrieve_node(head, idx):
        count = 0
        temp = head
        while temp != None:
                if count == idx:
                        return temp
                temp = temp.next
                count += 1
        return None      # Invalid Index
```

- Update value of specific index:

```
def update_elem(head, idx, elem):
        count = 0
        temp = head
        while temp != None:
                if count == idx:
                        temp.elem = elem
                        break
                temp = temp.next
                count += 1
```

- Search index of an element:

```
def search_elem(head, elem):
        count = 0
        temp = head
        while temp != None:
                if elem == temp.elem
                        return count
                        break
                temp = temp.next
                count += 1
        return -1: # If element is not found
```

- Insert:

```
def insert(head, elem, idx):
        total_nodes = count(head)
        if idx == 0: # Inserting at the beginning
                n = Node(elem, head)
                head = n
```

```
    elif idx >= 1 and idx < total_nodes - 1: # Inserting at the middle
        n = Node(elem, None)
        prev_n = retrieve_node(head, idx - 1)
        current_n = prev_n.next
        n.next= current_n
        prev_n.next = n
    elif idx == total_nodes - 1: # Inserting at the end
        n = Node(elem, None)
        last_n = retrieve_node(head, total_nodes-1)
        last_n.next = n
    else:
        print("Invalid Index")
    return head
```

- Delete:

```
def delete(head, idx):
    total_nodes = count(head)
    if idx == 0: # Removing first element
        temp = head
        head = head.next
        temp.next = None
    elif idx >= 1 and idx < total_nodes: # Removing middle/last elem
        prev_n = retrieve_node(head, idx - 1)
        current_n = prev_n.next
        prev_n.next = current_n.next
        current_n.next = None
    else:
        print("Invalid Index")
    return head
```

- Reverse:

```
def reverse_out_of_place(head):
        new_head = Node(head.elem, None)
        temp = head.next
        while temp != None:
                n = Node(temp.elem, new_head)
                new_head = n
                temp = temp.next
        return new_head
```

```
def reverse_in_place(head):
        new_head = None
        temp = head
        while temp != None:
                n = temp.next
                temp.next = new_head
                new_head = temp
                temp = n
        return new_head
```

- Rotate:

```
def rotate_left(head):
        first_n = head
        last_n = head.next
        while last_n.next != None:
                last_n = last_n.next
        last_n.next = head
        head = first_n.next
```

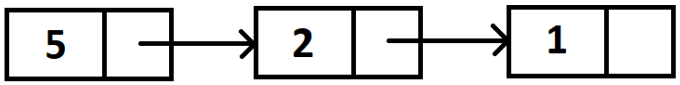
```
        first_n.next = None
        return head
```

```
def rotate_right(head):
        last_n = head.next
        second_last_n = head
        while last_n.next != None:
                last_n = last_n.next
                second_last_n = second_last_n.next
        last_n.next = head
        second_last_n.next = None
        head = last_n
        return head
```

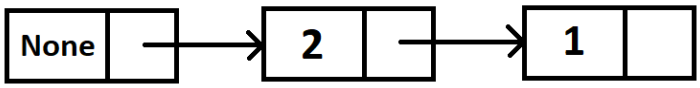
Types of Linked Lists:

Category 1	Category 2	Category 3
Non-Dummy Headed	Singly	Linear
Dummy Headed	Doubly	Circular

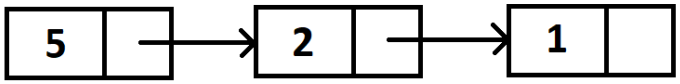
- **Non-Dummy Headed:** It means that the linked list's first node (the head) contains an item along with the information of the next node.



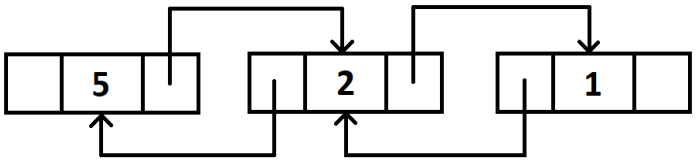
- **Dummy-Headed:** This refers to the linked list where a head node is a node-type object but it does not store any element on its own. Rather it stores the information of the next node where the starting value is stored. The benefit of this is that we do not need to handle the first item of the data carefully now.



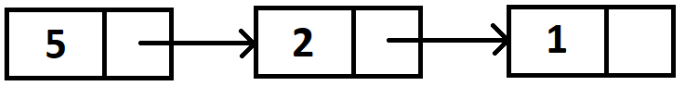
- **Singly:** It means every node has only the information of its next node but not the previous node.



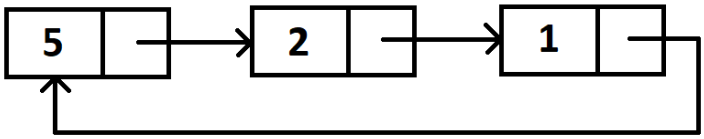
- **Doubly:** It means a node knows its next node and its previous node's information.



- **Linear:** It refers to the linked list where the linked list is linear in structure. The last node of the linked list will refer to None.



- **Circular:** This refers to a linked list where the linked list is circular in structure. The last node of the linked list will refer to the starting node.

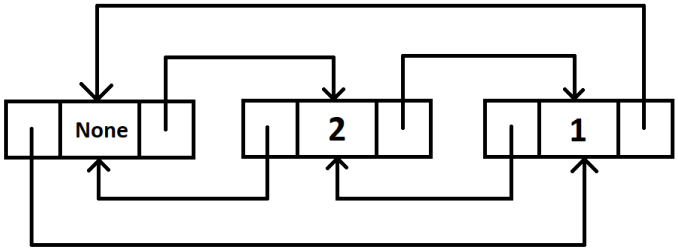


Follow-Up Question:

Q) Draw Non-Dummy Headed Doubly Circular Linked List
Q) Draw Dummy Headed Singly Linear Linked List
Q) Draw Dummy Headed Doubly Linear Linked List

Q) Draw Dummy Headed Doubly Circular Linked List

Ans:



Dummy Headed Doubly Circular Linked List Declaration/Initialization:

```
class DoublyNode:
        def __init__(self, elem, next, prev):
                self.elem = elem
                self.next = next
                self.prev = prev
```

```
def list_from_array(arr):
        dummy_head = DoublyNode(None, None, None)
        dummy_head.next= dummy_head
        dummy_head.prev = dummy_head
        tail = dummy_head
        for i in range(len(arr)):
                n = DoublyNode(a[i], dummy_head, tail)
                tail.next = n
                tail = tail.next
                dummy_head.prev = tail
        return dummy_head
```

Dummy Headed Doubly Circular Linked List Operations:

- Iteration:

```
def iteration(dummy_head):
    temp = dummy_head.next
    while temp != dummy_head:
        print(temp.elem)
        temp = temp.next
```

- Retrieve node using index:

```
def retrieve_node(dummy_head, idx):
    temp = dummy_head.next
    count = 0
    while temp != dummy_head:
        if count == idx:
            return temp
        temp = temp.next
        count += 1
    return None # Invalid Index
```

- Insert:

```
def insert(dummy_head, elem, idx)
    new_n = Doubly(elem, None, None)
    current_n = retrieve_node(dummy_head, idx)
    prev_n = current_n.prev
    new_n.next = current_n
    new_n.prev = prev_n
    prev_n.next = new_n
    current_n.prev = new_n
    return dummy_head
```

- Delete:

```
def delete(dummy_head, idx):
        current_n = retrieved_node(dummy_head, idx)
        prev_n = current_n.prev
        next_n = current_n.next
        prev_n.next = next_n
        next_n.prev = prev_n
        current_n.next = None
        current_n.prev = None
        return dummy_head
```

Follow-Up Question:

Q) Write a function that moves the last node to the front in a given Singly Linked List.

Input: 1 → 2 → 3 → 4 → 5, Output: 5 → 1 → 2 → 3 → 4
Input: 3 → 8 → 1 → 5 → 7 → 12, Output: 12 → 3 → 8 → 1 → 5 → 7

Ans:

```
last = head
second_last = None
while last.next != None:
        second_last = last
        last = last.next
second_last.next = None
last.next = head
head = last
```

Q) Given a dummy-headed doubly linear linked list, rotate it right.

Ans:

```
last = dummy_head
second_last = None
while last.next != None:
        second_last = last
        last = last.next
```

```
second_last.next = None
last.next = dummy_head.next
dummy_head.next.prev = last
dummy_head.next = last
last.prev = dummy_head
```

Q) Given a dummy-headed doubly circular linked list, rotate it right by k node (where k is a positive integer)

Ans:

```
# Calculate the length of the list
length = 0
temp = dummy_head.next
while temp != dummy_head:
        length += 1
        temp = temp.next
# Adjust k to be within the bounds of the list length
k = k % length
if k == 0:
        return

# Find the new head after rotating
temp = dummy_head.next
for i in range(length - k):
        temp = temp.next

# Update the pointers to rotate the list
dummy_head.next.prev = dummy_head.prev
dummy_head.prev.next = dummy_head.next

dummy_head.next = temp
temp.prev = dummy_head

dummy_head.prev = temp.prev
temp.prev.next = dummy_head
```