Linked Lists

Linked Lists

- A linked list is a data structure that consists of a sequence of elements, each of which contains a reference (or "link") to the next element in the sequence.

- The elements are typically stored in non-contiguous memory locations, and the links allow the elements to be efficiently accessed in a specific order.

  - When traversing the list, the program can follow the references from one node to the next, in order, to access the data stored in each node.
  - It's important to note that the last node in the list usually contains a reference to a null value or a special value indicating the end of the list.

Linked Lists - Applications

- Dynamic memory allocation: Linked lists can be used to allocate memory dynamically, which is useful when the amount of memory required is not known in advance.
- Linked lists can be used to implement various types of data structures, such as stacks, queues, and hash tables.
- They also allow for efficient insertions and deletions, as elements can be easily added or removed without the need to move other elements around in memory.
- Linked lists can be used to implement **sparse matrices**, which are matrices with a large number of zero elements.

Linked Lists – Pros

- Dynamic size: Linked lists can grow or shrink in size as needed, while arrays have a fixed size.
- Efficient memory usage: Linked lists use memory more efficiently than arrays, as they only allocate memory for the actual data elements, rather than for the entire array.
- Insertion and deletion: Linked lists allow for efficient insertion and deletion of elements, while arrays require shifting elements to make room for new elements or fill the gap left by deleted elements.
- Flexibility: Linked lists can be used to create more complex data structures, such as multi-linked lists and circular linked lists.
- Cache efficiency: Linked lists can be more cache efficient than arrays, as elements in a linked list are scattered in memory, and so are less likely to cause cache misses.

Linked Lists – Cons

- Random access: Linked lists do not allow for random access to elements, while arrays do.
  - Access time: Linked lists have a longer access time than arrays, as elements need to be traversed one by one, rather than being accessed directly by index.

- More memory overhead: Linked lists require more memory overhead than arrays, as they need to store both the data element and the next pointer.

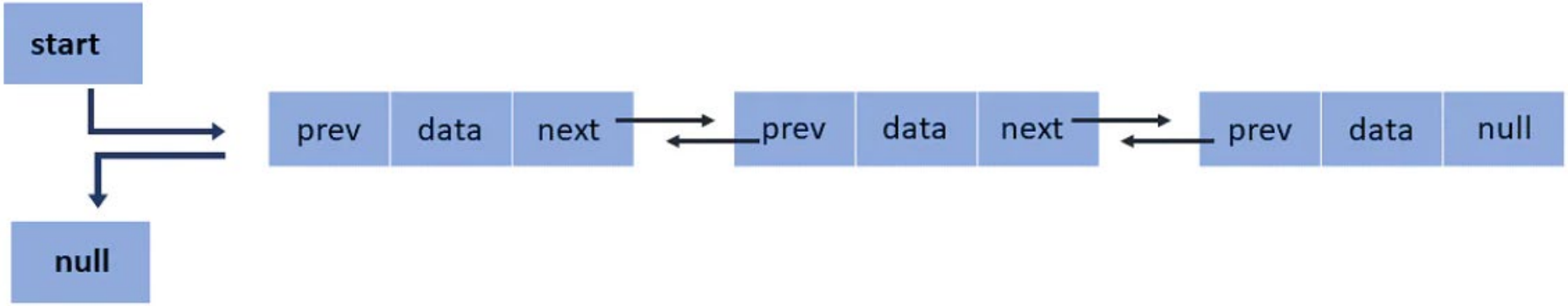
- More complex: Linked lists are more complex than arrays and may require more code to implement.

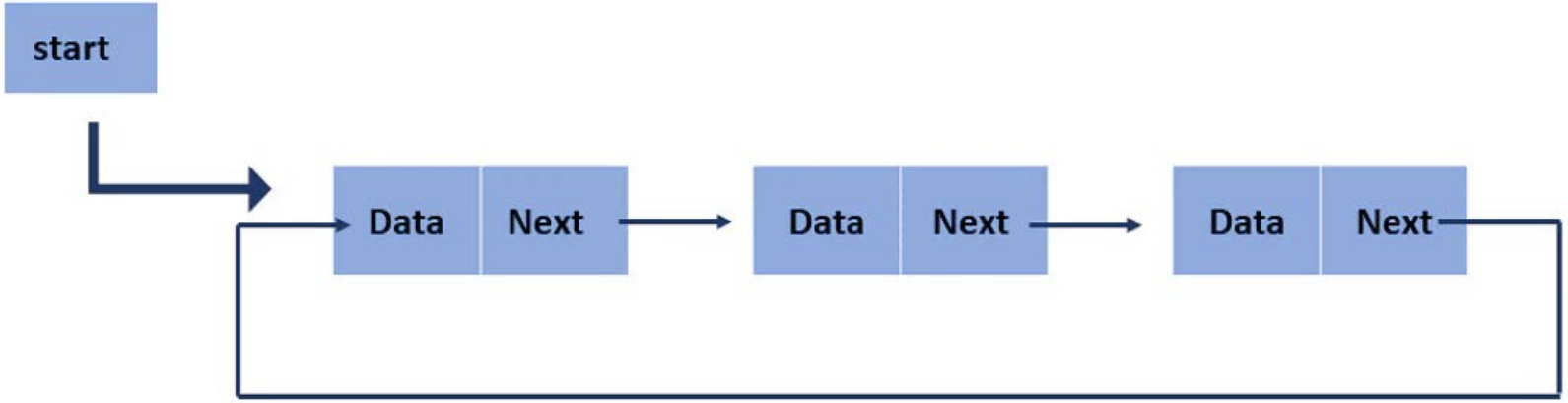
Linked Lists

1.Singly Linked List

2.Doubly Linked List

3.Circular Linked List





Operations on Linked Lists

Traversing: To traverse all nodes one by one.

Insertion: To insert new nodes at specific positions.

Deletion: To delete nodes from specific positions.

Searching: To search for an element from the linked list.

Linked List: Insert

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

self.head is an instance variable that represents the first node in the linked list.

When a new node is added to the list, the **self.head** variable is updated to point to the new node.

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

    def insert_front(self, data):
        new_node = Node(data, self.head)
        self.head = new_node

    def insert_back(self, data):
        new_node = Node(data, None)
        if self.head is None:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
```

```python
# Test the linked list
ll = LinkedList()
ll.insert_front(5)
ll.insert_front(3)
ll.insert_back(7)
ll.insert_back(9)
ll.insert_middle(4, 2)
ll.print_list()
```

```
3
5
4
7
9
```

Linked List: Insert

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

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

    def insert_front(self, data):
        new_node = Node(data, self.head)
        self.head = new_node

    def insert_back(self, data):
        new_node = Node(data, None)
        if self.head is None:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
```

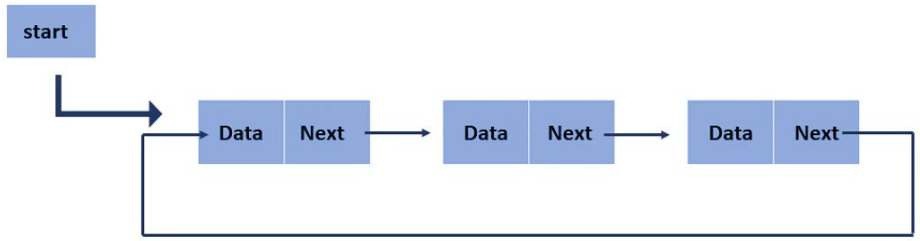
```python
def insert_middle(self, data, position):
    new_node = Node(data)
    current = self.head
    for i in range(position - 1):
        current = current.next
        if current is None:
            return
    new_node.next = current.next
    current.next = new_node
```

```python
def print_list(self):
    if self.head is None:
        print("Linked list is empty")
    current = self.head
    while current:
        print(current.data)
        current = current.next
```

```python
# Test the linked list
ll = LinkedList()
ll.insert_front(5)
ll.insert_front(3)
ll.insert_back(7)
ll.insert_back(9)
ll.insert_middle(4, 2)
ll.print_list()
```

```
3
5
4
7
9
```

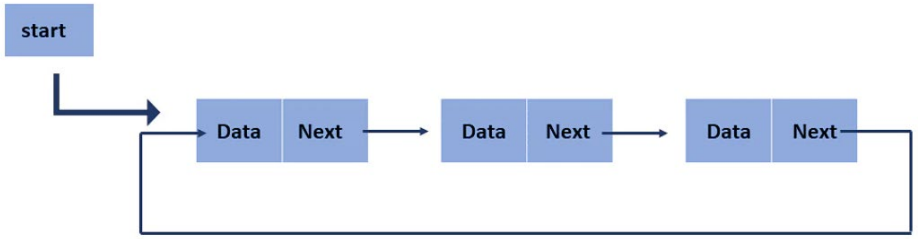
Circular Linked List: Insert




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

    def insert_front(self, data):
        new_node = Node(data, self.head)
        self.head = new_node

    def insert_back(self, data):
        new_node = Node(data, None)
        if self.head is None:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
```

```python
def insert_front(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.head.next = self.head
        return
    current = self.head
    while current.next != self.head:
        current = current.next
    current.next = new_node
    new_node.next = self.head
    self.head = new_node
```

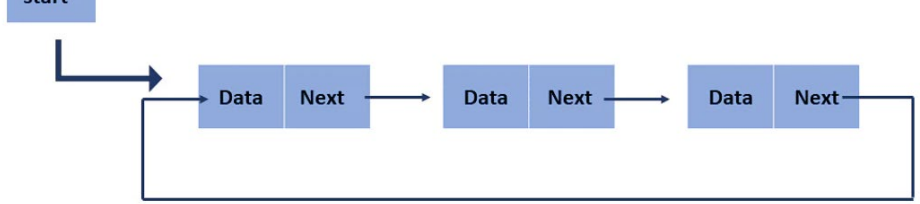
Circular Linked List: Insert




```
def insert_front(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.head.next = self.head
        return
    current = self.head
    while current.next != self.head:
        current = current.next
    current.next = new_node
    new_node.next = self.head
    self.head = new_node
```

```
def insert_back(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.head.next = self.head
        return
    current = self.head
    while current.next != self.head:
        current = current.next
    current.next = new_node
    new_node.next = self.head
```

Circular Linked List: Insert



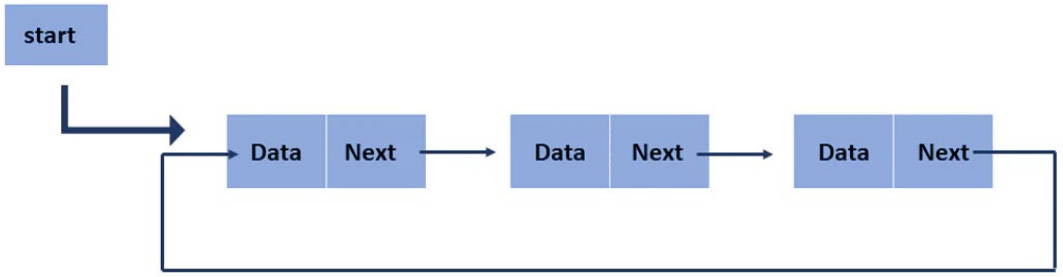

```
24    def insert_middle(self, data, position):
25        new_node = Node(data)
26        current = self.head
27        for i in range(position - 1):
28            current = current.next
29            if current is None:
30                return
31        new_node.next = current.next
32        current.next = new_node
```

```
def insert_middle(self, data, position):
    new_node = Node(data)
    current = self.head
    for i in range(position - 1):
        current = current.next
        if current.next == self.head:
            return
    new_node.next = current.next
    current.next = new_node
```

Circular Linked List: Print

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

```python
def print_list(self):
    if self.head is None:
        print("Linked list is empty")
    current = self.head
    while current:
        print(current.data)
        current = current.next
```




```python
def print_list(self):
    current = self.head
    while current:
        print(current.data)
        current = current.next
        if current == self.head:
            break
```

Linked list – delete

```python
# Node class for linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

```python
# Function to print the linked list
def print_list(self):
    current = self.head
    while current:
        print(current.data, end=' ')
        current = current.next
    print('')
```


```python
# Function to delete a node from the begi
def delete_from_beginning(self):
    if self.head is None:
        return
    self.head = self.head.next

# Function to delete a node from the end
def delete_from_end(self):
    if self.head is None:
        return
    if self.head.next is None:
        self.head = None
        return
    current = self.head
    while current.next.next:
        current = current.next
    current.next = None
```

Linked list – delete

```python
# Node class for linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

```python
# Function to delete a node from the begi
def delete_from_beginning(self):
    if self.head is None:
        return
    self.head = self.head.next

# Function to delete a node from the end
def delete_from_end(self):
    if self.head is None:
        return
    if self.head.next is None:
        self.head = None
        return
    current = self.head
    while current.next.next:
        current = current.next
    current.next = None
```

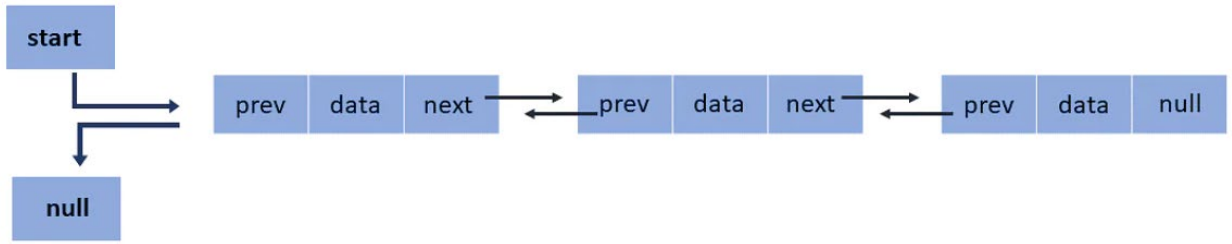
```python
# Function to delete a node from a specific position
def delete_at_position(self, position):
    if self.head is None:
        return
    if position == 0:
        self.head = self.head.next
        return
    current = self.head
    for i in range(position-1):
        if current.next is None:
            return
        current = current.next
    current.next = current.next.next
```

Doubly Linked List: Insert

```python
class DoublyNode:
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev

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

```python
def insert_front(self, data):
    new_node = DoublyNode(data, self.head, None)
    if self.head:
        self.head.prev = new_node
    self.head = new_node
    if not self.tail:
        self.tail = self.head
```

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

    def insert_front(self, data):
        new_node = Node(data, self.head)
        self.head = new_node

    def insert_back(self, data):
        new_node = Node(data, None)
        if self.head is None:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
```

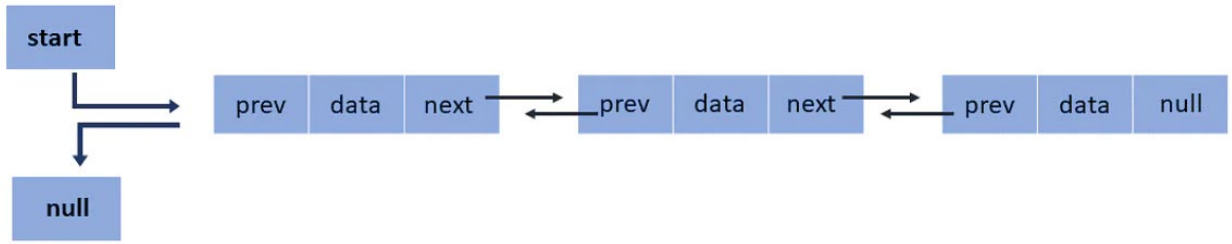


Doubly Linked List: Insert

```python
class DoublyNode:
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev

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

```python
def insert_back(self, data):
    new_node = DoublyNode(data, None, self.tail)
    if self.tail:
        self.tail.next = new_node
    self.tail = new_node
    if not self.head:
        self.head = self.tail
```

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

    def insert_front(self, data):
        new_node = Node(data, self.head)
        self.head = new_node

    def insert_back(self, data):
        new_node = Node(data, None)
        if self.head is None:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
```

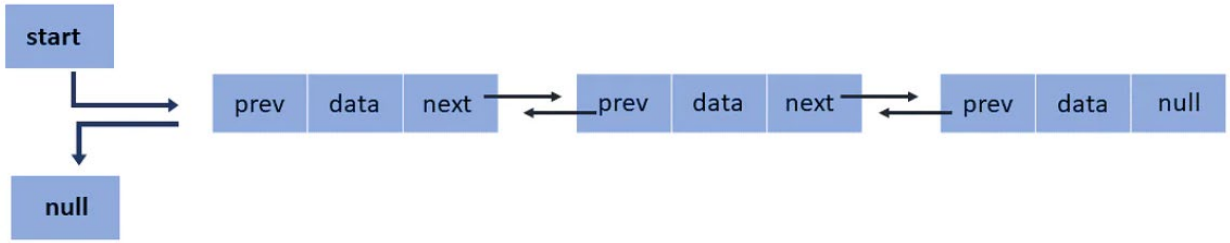


Doubly Linked List: Insert

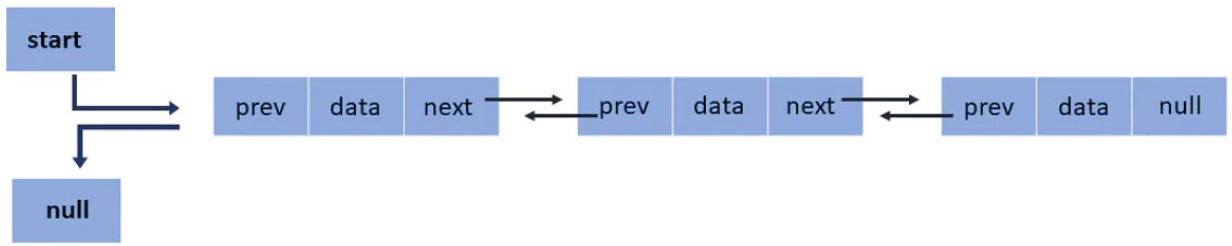
```python
def insert_middle(self, data, position):
    new_node = DoublyNode(data)
    current = self.head
    for i in range(position - 1):
        current = current.next
        if current is None:
            return
    new_node.next = current.next
    new_node.prev = current
    if current.next:
        current.next.prev = new_node
    current.next = new_node
```

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

```python
    def insert_middle(self, data, position):
        new_node = Node(data)
        current = self.head
        for i in range(position - 1):
            current = current.next
            if current is None:
                return
        new_node.next = current.next
        current.next = new_node
```



```python
class DoublyNode:
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev
```




```python
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert_front(self, data):
        new_node = DoublyNode(data, self.head, Non
        if self.head:
            self.head.prev = new_node
        self.head = new_node
        if not self.tail:
            self.tail = self.head
```

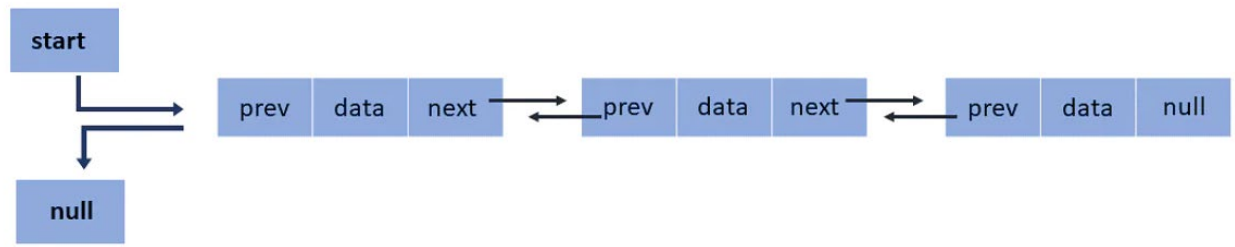
```python
def insert_back(self, data):
    new_node = DoublyNode(data, None, self.tail)
    if self.tail:
        self.tail.next = new_node
    self.tail = new_node
    if not self.head:
        self.head = self.tail
```

```python
class DoublyNode:
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev
```

```python
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert_front(self, data):
        new_node = DoublyNode(data, self.head, None)
        if self.head:
            self.head.prev = new_node
        self.head = new_node
        if not self.tail:
            self.tail = self.head
```

```python
def insert_middle(self, data, position):
    new_node = DoublyNode(data)
    current = self.head
    for i in range(position - 1):
        current = current.next
        if current is None:
            return
    new_node.next = current.next
    new_node.prev = current
    if current.next:
        current.next.prev = new_node
    current.next = new_node
```

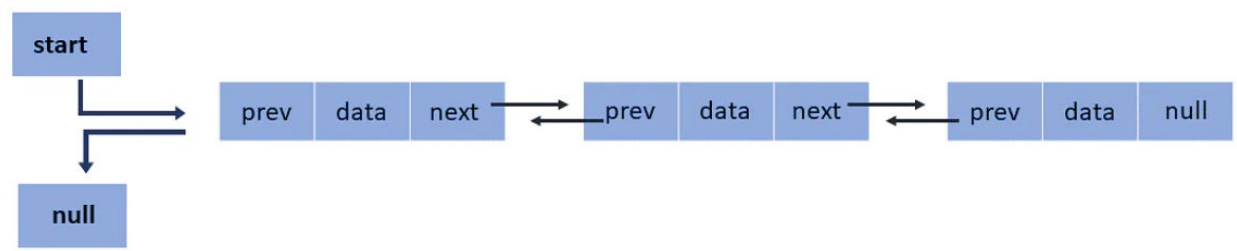
```python
def print_list(self):
    current = self.head
    while current:
        print(current.data)
        current = current.next
```



```python
class DoublyNode:
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev
```

```python
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert_front(self, data):
        new_node = DoublyNode(data, self.head, None)
        if self.head:
            self.head.prev = new_node
        self.head = new_node
        if not self.tail:
            self.tail = self.head
```

```python
def insert_middle(self, data, position):
    new_node = DoublyNode(data)
    current = self.head
    for i in range(position - 1):
        current = current.next
        if current is None:
            return
    new_node.next = current.next
    new_node.prev = current
    if current.next:
        current.next.prev = new_node
    current.next = new_node
```



Linked list – Insert

```python
# Node class for linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

```python
# Function to insert a node at the beginning
def insert_at_beginning(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Function to insert a node at the end
def insert_at_end(self, new_data):
    new_node = Node(new_data)
    if self.head is None:
        self.head = new_node
        return
    last = self.head
    while last.next:
        last = last.next
    last.next = new_node
```

```python
# Function to insert a node at a specific position
def insert_at_position(self, new_data, position):
    new_node = Node(new_data)
    if self.head is None:
        self.head = new_node
        return
    if position == 0:
        new_node.next = self.head
        self.head = new_node
        return
    current = self.head
    for i in range(position-1):
        if current.next is None:
            break
        current = current.next
    new_node.next = current.next
    current.next = new_node
```

Linked list – delete

```python
# Node class for linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

```python
# Function to print the linked list
def print_list(self):
    current = self.head
    while current:
        print(current.data, end=' ')
        current = current.next
    print('')
```


```python
# Function to delete a node from the begi
def delete_from_beginning(self):
    if self.head is None:
        return
    self.head = self.head.next

# Function to delete a node from the end
def delete_from_end(self):
    if self.head is None:
        return
    if self.head.next is None:
        self.head = None
        return
    current = self.head
    while current.next.next:
        current = current.next
    current.next = None
```

Linked list – delete

```python
# Node class for linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

```python
# Function to delete a node from the begi
def delete_from_beginning(self):
    if self.head is None:
        return
    self.head = self.head.next

# Function to delete a node from the end
def delete_from_end(self):
    if self.head is None:
        return
    if self.head.next is None:
        self.head = None
        return
    current = self.head
    while current.next.next:
        current = current.next
    current.next = None
```

```python
# Function to delete a node from a specific position
def delete_at_position(self, position):
    if self.head is None:
        return
    if position == 0:
        self.head = self.head.next
        return
    current = self.head
    for i in range(position-1):
        if current.next is None:
            return
        current = current.next
    current.next = current.next.next
```