Linked Lists Cheat Sheet (Theory Only)

---

1. Introduction to Linked Lists

A **linked list** is a linear data structure consisting of **nodes** where each node contains:

- **Data** (Value)
- **Pointer (Reference) to the next node**

Types of Linked Lists:

- **Singly Linked List (SLL)** → Each node points to the next node.
- **Doubly Linked List (DLL)** → Each node points to both previous and next nodes.
- **Circular Linked List (CLL)** → The last node points back to the first node.

Advantages of Linked Lists Over Arrays:

- **Dynamic Size** → No pre-allocation of memory required.
- **Efficient Insertions/Deletions** → No shifting needed like in arrays.
- **Memory Utilization** → Memory allocated as needed.

Disadvantages:

- **Extra memory for pointers**.
- **Slower access time (O(n))** compared to arrays (O(1) for indexed access).

---

2. Singly Linked List (SLL)

Nodes are connected in one direction using a pointer.
The last node **points to NULL (None in Python)**.
Memory Representation of SLL:

```
[10 | *] → [20 | *] → [30 | *] → None
```

---

3. Operations on Singly Linked List

(A) Traversing a Linked List

Algorithm: Start from the head and move through each node until NULL.

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

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

    def traverse(self):
        temp = self.head
        while temp:
            print(temp.data, end=" → ")
            temp = temp.next
        print("None")

# Example Usage
ll = LinkedList()
ll.head = Node(10)
ll.head.next = Node(20)
ll.head.next.next = Node(30)

ll.traverse()  # Output: 10 → 20 → 30 → None
```

(B) Searching in a Linked List

Algorithm: Traverse through the list and compare each node's value with the target.

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

# Example:
print(ll.search(20))  # Output: True
print(ll.search(50))  # Output: False
```

(C) Insertion in a Linked List

Three Cases:

1st **Insert at Beginning**
2nd**Insert at End**
3rd**Insert at Specific Position**

```
# Insert at Beginning
def insert_at_beginning(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Insert at End
def insert_at_end(self, new_data):
    new_node = Node(new_data)
    if self.head is None:
        self.head = new_node
        return
    temp = self.head
    while temp.next:
        temp = temp.next
    temp.next = new_node
```

(D) Deletion in a Linked List

Three Cases:

1st **Delete First Node**
2nd**Delete Last Node**
3rd**Delete a Node with Given Key**

```
# Delete a Node by Key
def delete_node(self, key):
    temp = self.head

    if temp is not None and temp.data == key:
        self.head = temp.next
        temp = None
        return

    prev = None
    while temp is not None and temp.data != key:
        prev = temp
        temp = temp.next

    if temp is None:
```

```
        return  # Key not found

    prev.next = temp.next
    temp = None
```

---

4. Linked Representation of Stack & Queue

(A) Stack Using Linked List (LIFO - Last In, First Out)

Push (Insert at Head), **Pop (Remove from Head)**.

```
class StackLL:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if self.top is None:
            return "Stack Underflow"
        popped_data = self.top.data
        self.top = self.top.next
        return popped_data
```

(B) Queue Using Linked List (FIFO - First In, First Out)

Enqueue (Insert at Tail), **Dequeue (Remove from Head)**.

```
class QueueLL:
    def __init__(self):
        self.front = self.rear = None

    def enqueue(self, data):
        new_node = Node(data)
        if self.rear is None:
            self.front = self.rear = new_node
            return
        self.rear.next = new_node
        self.rear = new_node

    def dequeue(self):
        if self.front is None:
            return "Queue Underflow"
        dequeued_data = self.front.data
        self.front = self.front.next
        return dequeued_data
```

---

5. Doubly Linked List (DLL)

Each node contains **two pointers** → prev (previous node) and next (next node).

Allows **both forward & backward traversal**.

Memory Representation of DLL:

```
None ← [10 | * | *] → [20 | * | *] → [30 | * | None]
```

Operations on DLL:

- **Insertion at Beginning, End, or Specific Position**
- **Deletion of a Node**

- **Traversal in Both Directions**

Example of Insertion at Beginning:

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

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

    def insert_at_beginning(self, data):
        new_node = DNode(data)
        new_node.next = self.head
        if self.head is not None:
            self.head.prev = new_node
        self.head = new_node
```

---

6. Circular Linked List (CLL)

The last node points back to the first node, forming a circular structure.

Types:

- **Singly Circular Linked List** → Only next pointer forms a loop.
- **Doubly Circular Linked List** → Both next and prev pointers form loops.

Operations in CLL:

- **Insertion at Beginning or End**
- **Deletion of a Node**
- **Traversal in a Circular Manner**

Example of Traversal in CLL:

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

    def traverse(self):
        if self.head is None:
            return "Empty List"
        temp = self.head
        while True:
            print(temp.data, end=" → ")
            temp = temp.next
            if temp == self.head:
                break
        print()
```

---

7. Comparison Between SLL, DLL & CLL

Feature	**Singly Linked List**	**Doubly Linked List**	**Circular Linked List**
Pointers	Only next pointer	prev & next pointers	next (Singly), prev & next (Doubly)
Traversal	Forward only	Forward & Backward	Circular Traversal
Memory Usage	Less	More (Extra prev pointer)	Similar to DLL

Feature	**Singly Linked List**	**Doubly Linked List**	**Circular Linked List**
Complexity	Moderate	Faster due to prev pointer	Faster for continuous operations

---

Key Takeaways

- **Singly Linked List (SLL):** Uses a single pointer, supports basic traversal.
- **Doubly Linked List (DLL):** Allows bidirectional traversal, but uses more memory.
- **Circular Linked List (CLL):** Provides continuous navigation, useful in round-robin scheduling.

This **Linked List Cheat Sheet** covers **SLL, DLL, CLL operations, representation in memory, stacks & queues using linked lists**. Let me know if you need further explanations!