🟦 Linked Lists & Double-Linked Lists

1️⃣ Grundidee **Verkettete Listen** speichern Elemente in **Knoten** (Nodes). Jeder Knoten hält einen **Wert** und mindestens **einen Zeiger** auf den nächsten Knoten.

- **Singly Linked List (einfach verkettet)**: jeder Knoten zeigt auf **next**.
- **Doubly Linked List (doppelt verkettet)**: jeder Knoten zeigt auf **prev** und **next**. ➡️ Ziel: **flexibles Einfügen/Entfernen** ohne Verschieben vieler Elemente.

2️⃣ Voraussetzungen

- Definition einer **Knotenklasse** (Node)
- Verwaltung von **head** (und bei doppelter Verkettung zusätzlich **tail**)
- Optionale **Hilfsmethoden**: append, prepend, insert, remove

3️⃣ Laufzeiten & Eigenschaften

Eigenschaft	Singly Linked List	Doubly Linked List
Zugriff per Index	O(n)	O(n)
Suche	O(n)	O(n)
Einfügen/Löschen am Kopf	O(1)	O(1)
Einfügen/Löschen am Ende	O(n) (ohne tail), O(1) (mit tail*)	O(1)
Einfügen/Löschen bei bekanntem Knoten	O(1) (Löschen benötigt Vorgängerzeiger)	O(1)
Speicherbedarf	O(n)	O(n) (etwas mehr wegen prev)
In-place	ja	ja
Stabil	–	–

Hinweis: Bei der einfach verketteten Liste ist **Löschen in O(1)** nur möglich, wenn **Vorgänger** bereits bekannt ist; sonst muss er gesucht werden (**O(n)**).

4️⃣ Struktur & Knotenklassen **Singly Linked Node**

```
class SinglyNode:
    def __init__(self, value):
        self.value = value
        self.next = None
```

Doubly Linked Node

```
class DoublyNode:
    def __init__(self, value):
        self.value = value
```

```
        self.prev = None
        self.next = None
```

5️⃣ Operationen (klassischer Prüfungsstoff)

1. **Einfügen am Kopf (prepend)**
   - Singly/Doubly: neuer Knoten zeigt auf bisherigen Kopf; Kopf wird aktualisiert. (O(1))
2. **Anhängen am Ende (append)**
   - Singly ohne tail: Liste durchlaufen bis Ende. (O(n))
   - Singly mit tail: direkt verknüpfen. (O(1))
   - Doubly: mit tail immer O(1).
3. **Einfügen nach einem bekannten Knoten**
   - Zeiger setzen (next und ggf. prev). (O(1))
4. **Löschen eines bekannten Knotens**
   - Singly: Vorgänger.next = aktueller.next (Vorgänger muss bekannt sein). (O(1))
   - Doubly: prev.next = next und next.prev = prev; Kopf/Tail anpassen. (O(1))
5. **Löschen nach Wert**
   - Erst suchen (O(n)), dann wie oben verknüpfen. Gesamtkosten **O(n)**.

6️⃣ Besonderheiten / Prüfungsrelevante Hinweise

- **Iteration vs. allgemeines Entfernen**: Während einer Iteration befindet sich der Iterator bereits auf dem Knoten → Entfernen **O(1)**. Allgemein muss der Knoten erst gefunden werden → **O(n)**.
- **Edge Cases**: leere Liste; Löschen des **head**; Löschen des **tail**; Einzelknotenliste.
- **Richtungslose Traversierung**: Singly nur vorwärts; Doubly vorwärts und rückwärts.

7️⃣ Vor- und Nachteile **Singly Linked List**

- Vorteile: einfach, geringer Speicherbedarf, schnelles Einfügen/Löschen am Kopf.
- Nachteile: kein direkter Indexzugriff, Löschen erfordert meist Vorgänger, langsame Suche.

Doubly Linked List

- Vorteile: effizientes Löschen/Einfügen überall (bei bekanntem Knoten), bequeme Rückwärtsiteration, O(1) am Kopf und Ende.
- Nachteile: höherer Speicherbedarf, komplexere Zeigerpflege.

🧠 Merksatz für die Prüfung *„Löschen ist nur dann O(1), wenn die **Position** des Knotens bekannt ist. Sonst dominiert die **Suche** mit O(n).“*

8️⃣ Python-Implementierung (einfach verkettet)

```
class SinglyNode:
    def __init__(self, value):
        self.value = value
        self.next = None
```

```python
class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None  # optional, für O(1) append

    def prepend(self, value):
        node = SinglyNode(value)
        node.next = self.head
        self.head = node
        if self.tail is None:
            self.tail = node
        return node

    def append(self, value):
        node = SinglyNode(value)
        if self.head is None:
            self.head = self.tail = node
            return node
        # mit tail O(1)
        self.tail.next = node
        self.tail = node
        return node

    def remove_by_value(self, value):
        prev, cur = None, self.head
        while cur is not None:
            if cur.value == value:
                if prev is None:
                    # Kopf löschen
                    self.head = cur.next
                    if cur is self.tail:
                        self.tail = cur.next  # wird None
                else:
                    prev.next = cur.next
                    if cur is self.tail:
                        self.tail = prev
                cur.next = None
                return True
            prev, cur = cur, cur.next
        return False

    def __iter__(self):
        cur = self.head
        while cur:
            yield cur
            cur = cur.next
```

Python-Implementierung (doppelt verkettet)

```python
class DoublyNode:
    def __init__(self, value):
        self.value = value
```

```
        self.prev = None
        self.next = None

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

    def prepend(self, value):
        node = DoublyNode(value)
        node.next = self.head
        if self.head is not None:
            self.head.prev = node
        else:
            self.tail = node
        self.head = node
        return node

    def append(self, value):
        node = DoublyNode(value)
        if self.tail is None:
            self.head = self.tail = node
            return node
        node.prev = self.tail
        self.tail.next = node
        self.tail = node
        return node

    def remove(self, node):
        if node is None:
            return False
        if node.prev is not None:
            node.prev.next = node.next
        else:
            self.head = node.next
        if node.next is not None:
            node.next.prev = node.prev
        else:
            self.tail = node.prev
        node.prev = node.next = None
        return True

    def remove_by_value(self, value):
        cur = self.head
        while cur is not None:
            if cur.value == value:
                return self.remove(cur)
            cur = cur.next
        return False

    def __iter__(self):
        cur = self.head
```

```
        while cur:
            yield cur
            cur = cur.next
```

🔟 Beispiel

```
# Singly
sll = SinglyLinkedList()
sll.append(1); sll.append(2); sll.prepend(0)
sll.remove_by_value(2)

# Doubly
dll = DoublyLinkedList()
a = dll.append(1)
b = dll.append(2)
c = dll.append(3)
dll.remove(b)        # O(1), da Knoten bekannt
_ = dll.remove_by_value(3)  # O(n) Suche + O(1) Entfernen

# Iteration und Entfernen während Iteration (Doubly)
for node in list(dll):  # defensive Kopie
    if node.value == 1:
        dll.remove(node)
```