Introduction to Doubly Linked list

Doubly Linked List

A linear data structure where each node contains 3 components:

- **Data** : The actual value stored
- **Next pointer** : Reference to next node
- **Previous pointer** : Reference to the previous node

```
       +----+      +----+      +----+
null <-| 10 |<--->| 20 |<--->| 30 |-> null
       +----+      +----+      +----+
       HEAD                    TAIL
```

DLL : Node structure in Java

```
class Node {
    int data;
    Node next;
    Node prev;

    Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}
```

- Bidirectional traversal (forward and backward)
- Extra memory overhead for the previous pointer
- More flexible than singly linked lists for certain operations

DLL : Real-World Applications

- **LRU Cache** : DLL's shine here – Used in databases, OS & Web Servers

```
null ← [Page 5] ⇄ [Page 3] ⇄ [Page 7] ⇄ [Page 1] → null
          MRU                                 LRU
     (Most Recent)                       (Least Recent)
```

- **Music playlist editors**: Insert/delete songs anywhere without shifting
- **Text editors:** Cursor navigation with efficient insertion/deletion at any position
- **Operating System task schedulers**: Efficiently move processes between priority queues

Why do we need the tail reference ?

- Insert at end– Without tail, traversal from head to find last node
- Delete tail – Without tail, must traverse to find second-last node
- Access last element – Traverse entire list without tail
- Start backward traversal – First traverse to head and then backwards

With a tail reference all of the above become O(1) instead of O(n)

DLL – Forward Traversal

```
Before:
      head                tail
        ↓                   ↓
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
--------------------------------------
Intermediate:
current = head

10 → 20 → 30 → 40 → null
↑
current
current = current.next
...
current = 40
10 → 20 → 30 → 40
                ↑
             current
--------------------------------------
After:
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
```

```java
public void traverseForward() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
}
```

DLL – Backward Traversal

```
Before:
        head              tail
        ↓                 ↓
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
--------------------------------------
Intermediate:

current = tail
current = current.prev

...
40 ← 30 ← 20 ← 10
               ↑
            current
--------------------------------------
After:
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
```

```java
public void traverseBackward() {
    Node current = tail;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.prev;
    }
}
```

DLL : Insert at Head

```
Before:

      head       tail
       ↓          ↓
null ← 10 ↔ 20 ↔ 30 → null
--------------------------------------
Place newNode(5) at the head position:

           newNode(5)   head
             ↓           ↓
           null ← 5 → 10
                     ↑
                     |  10.prev = 5
--------------------------------------
After:
     head            tail
      ↓               ↓
null ← 5 ↔ 10 ↔ 20 ↔ 30 → null
```

```java
public void insertAtHead(int data) {
    Node newNode = new Node(data);

    if (head == null) {     // empty list
        head = tail = newNode;
        return;
    }

    newNode.next = head;    // 5.next = 10
    newNode.prev = null;    // 5.prev = null
    head.prev = newNode;    // 10.prev = 5
    head = newNode;         // update head
}
```

DLL : Insert at Tail

```
Before:
      head       tail
       ↓          ↓
null ← 10 ⇔ 20 ⇔ 30 → null
--------------------------------------
null ← 10 ⇔ 20 ⇔ 30 → null
                      ↑
                      | (newNode.prev = 30)

                  30 → 40 → null
                      ↑
                      | (newNode.next = null)

(30.next = 40)
(tail = 40)
--------------------------------------
After inserting 40 at tail:
      head           tail
       ↓              ↓
null ← 10 ⇔ 20 ⇔ 30 ⇔ 40 → null
```

```java
public void insertAtTail(int data) {
    Node newNode = new Node(data);

    if (tail == null) {         // empty list
        head = tail = newNode;
        return;
    }

    newNode.prev = tail;        // 40.prev = 30
    newNode.next = null;        // 40.next = null
    tail.next = newNode;        // 30.next = 40
    tail = newNode;             // update tail
}
```

DLL : Insert at Index (ex : 2)

```
Before:
              head              tail
               ↓                 ↓
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
                      ↑
                insert here (index = 2)
--------------------------------------
Intermediate:
              head               tail
               ↓                  ↓
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
                   ↑
                current
--------------------------------------
After inserting 25 at index
              head                   tail
                ↓                     ↓
null ← 10 ↔ 20 ↔ 25 ↔ 30 ↔ 40 → null
                   ↑      ↑
               current  newNode
```

```java
public void insertAtIndex(int index, int data) {
    if (index == 0) {
        insertAtHead(data);
        return;
    }

    Node current = head;
    for (int i = 0; i < index - 1; i++) {
        current = current.next;  // current stops at 20
    }

    if (current == tail) {
        insertAtTail(data);
        return;
    }

    Node newNode = new Node(data);

    newNode.prev = current;           // 25.prev = 20
    newNode.next = current.next;      // 25.next = 30
    current.next.prev = newNode;      // 30.prev = 25
    current.next = newNode;           // 20.next = 25
}
```

The code above is missing null check for head,current & checking if index is out of bounds for given doubly linked list – add that as an exercise

DLL : Delete head

```
Before deleting 10 at index 0:
      head           tail
       ↓              ↓
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
--------------------------------------
Intermediate:

temp = head
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
       ↑
      temp

temp (10)   null ← 20 ↔ 30 ↔ 40 → null
   ↓              ↑
 (removed)      new head
--------------------------------------
After:
      head    tail
       ↓       ↓
null ← 20 ↔ 30 ↔ 40 → null
```

```java
public void deleteHead() {
    if (head == null) return;

    if (head == tail) {   // only one node
        head = tail = null;
        return;
    }

    Node temp = head;      // temp = 10
    head = head.next;      // head = 20
    head.prev = null;      // 20.prev = null
    temp.next = null;      // cleanup
}
```

DLL : Delete tail

```
Before deleting tail(40):
          head                 tail
            ↓                    ↓
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
--------------------------------------
Intermediate:

temp = tail

          head                 tail
            ↓                    ↓
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
                                 ↑
                                temp
--------------------------------------
After:
          head          tail
            ↓             ↓
null ← 10 ↔ 20 ↔ 30 → null
```

```java
public void deleteTail() {
    if (tail == null) return;

    if (head == tail) {   // single node
        head = tail = null;
        return;
    }

    Node temp = tail;      // temp = 40
    tail = tail.prev;      // tail = 30
    tail.next = null;      // 30.next = null
    temp.prev = null;      // cleanup
}
```

DLL : Delete at Index

```
Before deleting:
      head                 tail
       ↓                    ↓
null ← 10 ↔ 20 ↔ 30 ↔ 40 ↔ 50 → null
                  ↑
           delete this (index = 2)
--------------------------------------
Intermediate:
current = 20
toDelete = current.next = 30

null ← 10 ↔ 20 ↔ 30 ↔ 40 ↔ 50 → null
             ↑     ↑
         current  toDelete
--------------------------------------
After:
       head            tail
       ↓                ↓
null ← 10 ↔ 20 ↔ 40 ↔ 50 → null
```

```java
public void deleteAtIndex(int index) {
    if (index == 0) {
        deleteHead();
        return;
    }

    Node current = head;
    for (int i = 0; i < index - 1; i++) {
        current = current.next;     // stop BEFORE the target
    }

    //Exercise: What if index is the last element or beyond?

    Node toDelete = current.next;    // node being removed

    if (toDelete == tail) {
        deleteTail();
        return;
    }

    current.next = toDelete.next;        // 20.next = 40
    toDelete.next.prev = current;        // 40.prev = 20

    toDelete.next = toDelete.prev = null; // cleanup
}
```

DLL : Delete by value

```
Before deleting:
       head                tail
        ↓                   ↓
null ← 10 ⇔ 20 ⇔ 30 ⇔ 40 ⇔ 50 → null
                 ↑
               delete 30
--------------------------------------
Intermediate:
1.Find node to delete(temp)
2.Get a reference to previous node(prevNode)
3.Get a reference to next node(nextNode)
       head                tail
        ↓                   ↓
null ← 10 ⇔ 20 ⇔ 30 ⇔ 40 ⇔ 50 → null
            ↑     ↑     ↑(nextNode)
         prevNode temp (node to delete)
--------------------------------------
After:
      head           tail
null ← 10 ⇔ 20 ⇔ 40 ⇔ 50 → null
```

```java
public void deleteByValue(int value) {
    if (head == null) return;

    // if deleting head
    if (head.data == value) {
        deleteHead();
        return;
    }

    Node temp = head;

    while (temp != null && temp.data != value) {
        temp = temp.next;
    }

    if (temp == null) return; // not found

    // if deleting tail
    if (temp == tail) {
        deleteTail();
        return;
    }

    Node prevNode = temp.prev;    // node before temp
    Node nextNode = temp.next;    // node after temp

    prevNode.next = nextNode;     // 20.next = 40
    nextNode.prev = prevNode;     // 40.prev = 20

    temp.next = temp.prev = null; // cleanup
}
```