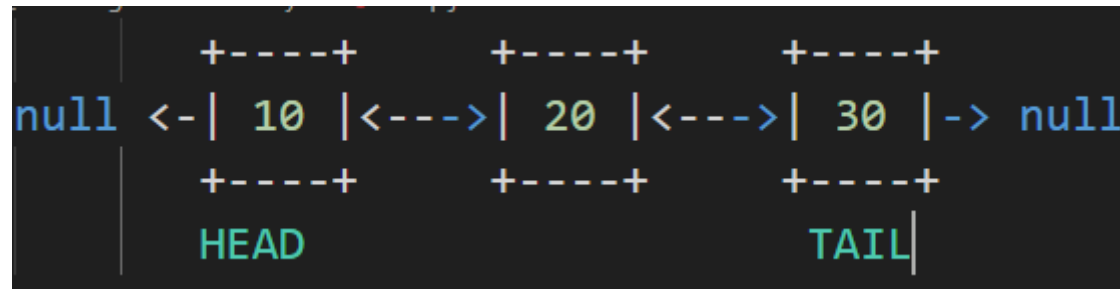


Introduction to Doubly Linked list

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



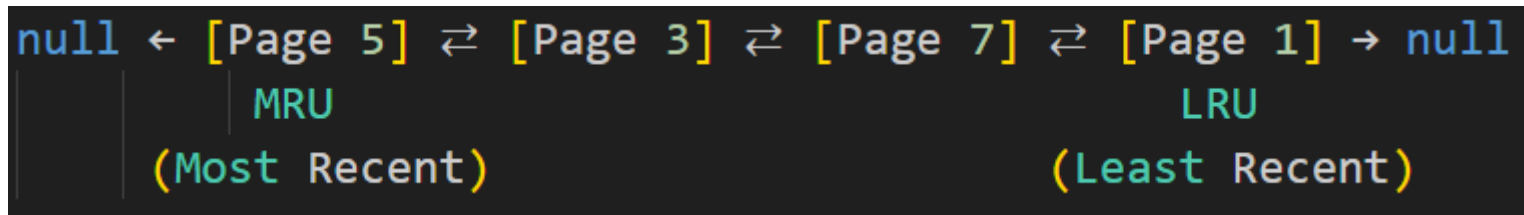
DLL : Node structure in Java

```
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



- **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
```

```
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
```

```
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

```
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)

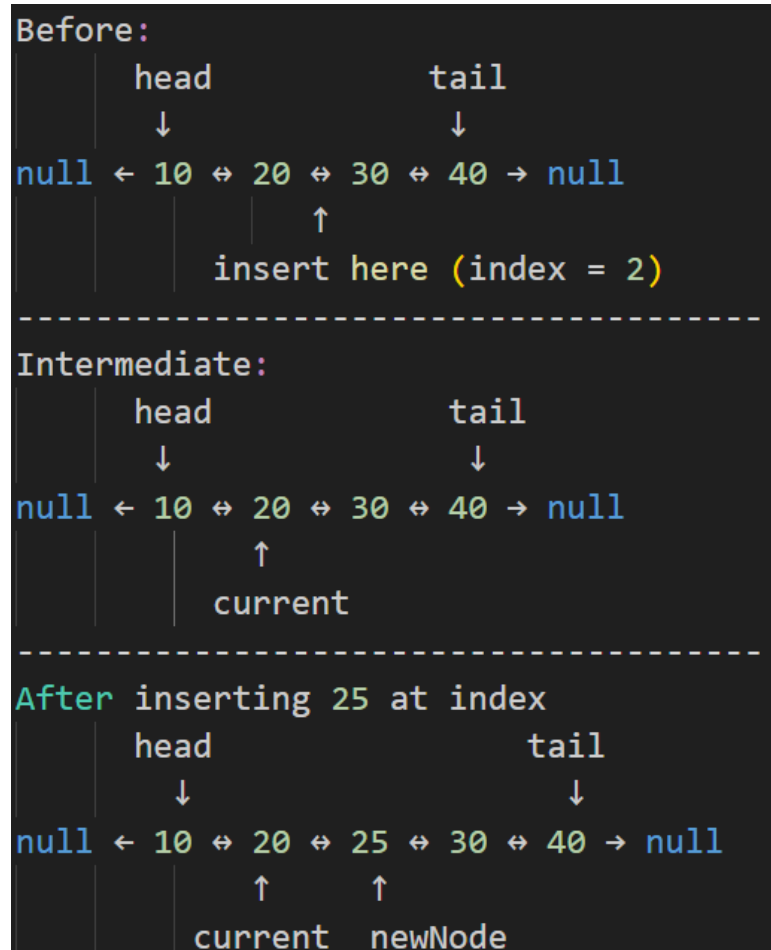
(30.next = 40)
(tail = 40)
-----
After inserting 40 at tail:
      head      tail
      ↓        ↓
null ← 10 ↔ 20 ↔ 30 ↔ 40 → null
```

```
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)



```
1 public void insertAtIndex(int index, int data) {
2     if (index == 0) {
3         insertAtHead(data);
4         return;
5     }
6
7     Node current = head;
8     for (int i = 0; i < index - 1; i++) {
9         current = current.next; // current stops at 20
10    }
11
12    if (current == tail) {
13        insertAtTail(data);
14        return;
15    }
16
17    Node newNode = new Node(data);
18
19    newNode.prev = current; // 25.prev = 20
20    newNode.next = current.next; // 25.next = 30
21    current.next.prev = newNode; // 30.prev = 25
22    current.next = newNode; // 20.next = 25
23 }
```

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
```

```
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
```

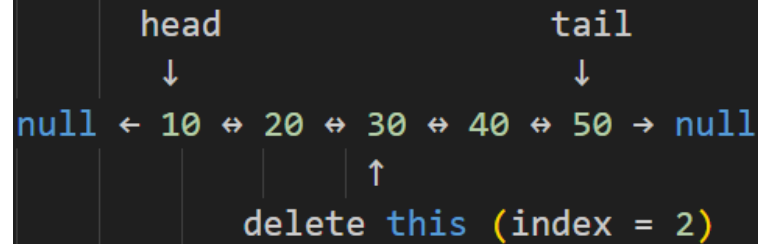
```
public void deleteTail() {
    if (tail == null) return;

    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
}
```

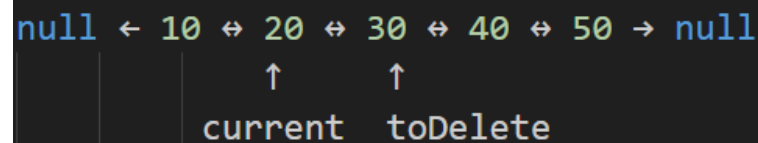
DLL : Delete at Index

Before deleting:

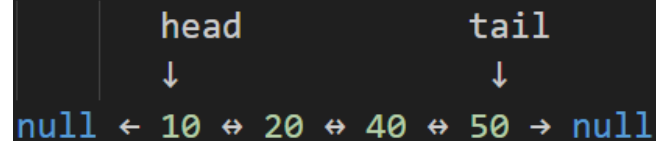


Intermediate:

current = 20
toDelete = current.next = 30



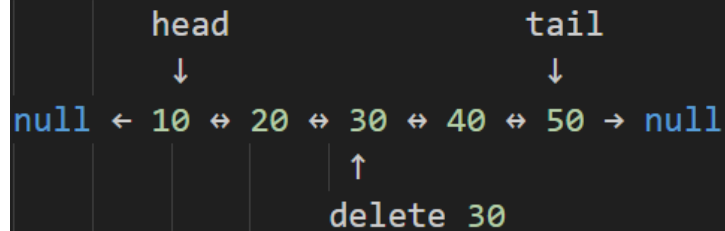
After:



```
1 public void deleteAtIndex(int index) {
2     if (index == 0) {
3         deleteHead();
4         return;
5     }
6
7     Node current = head;
8     for (int i = 0; i < index - 1; i++) {
9         current = current.next;    // stop BEFORE the target
10    }
11
12    //Exercise: What if index is the last element or beyond?
13
14    Node toDelete = current.next;    // node being removed
15
16    if (toDelete == tail) {
17        deleteTail();
18        return;
19    }
20
21    current.next = toDelete.next;    // 20.next = 40
22    toDelete.next.prev = current;    // 40.prev = 20
23
24    toDelete.next = toDelete.prev = null; // cleanup
25 }
26
```

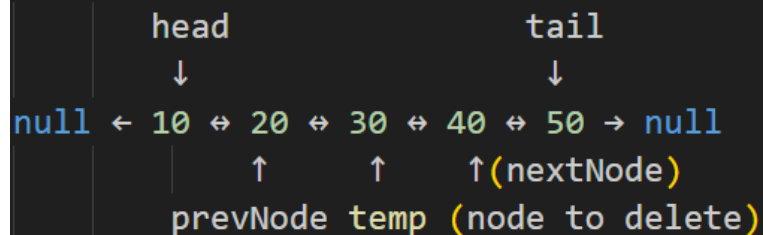
DLL : Delete by value

Before deleting:

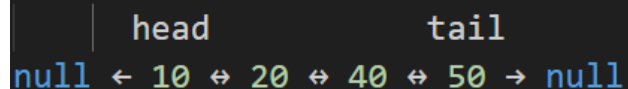


Intermediate:

1. Find node to delete(temp)
2. Get a reference to previous node(prevNode)
3. Get a reference to next node(nextNode)



After:



```
1 public void deleteByValue(int value) {
2     if (head == null) return;
3
4     // if deleting head
5     if (head.data == value) {
6         deleteHead();
7         return;
8     }
9
10    Node temp = head;
11
12    while (temp != null && temp.data != value) {
13        temp = temp.next;
14    }
15
16    if (temp == null) return; // not found
17
18    // if deleting tail
19    if (temp == tail) {
20        deleteTail();
21        return;
22    }
23
24    Node prevNode = temp.prev; // node before temp
25    Node nextNode = temp.next; // node after temp
26
27    prevNode.next = nextNode; // 20.next = 40
28    nextNode.prev = prevNode; // 40.prev = 20
29
30    temp.next = temp.prev = null; // cleanup
31 }
```