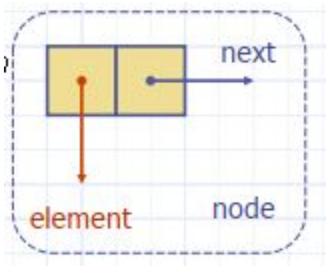
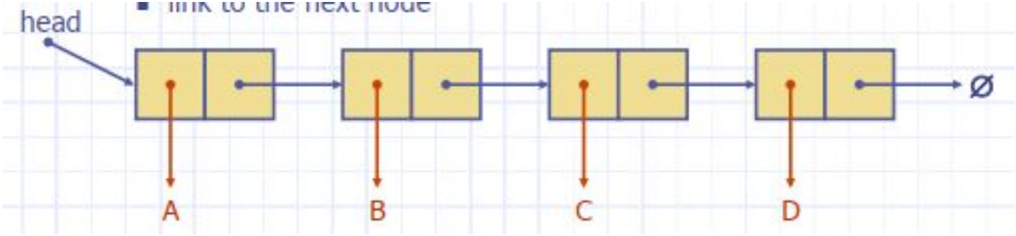
Linked lists

Linked list

- Linear data structures where elements are stored in nodes, and each node points to the next one.
  - Singly, double, circular
- A concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - Element
  - Link to the next node





Node

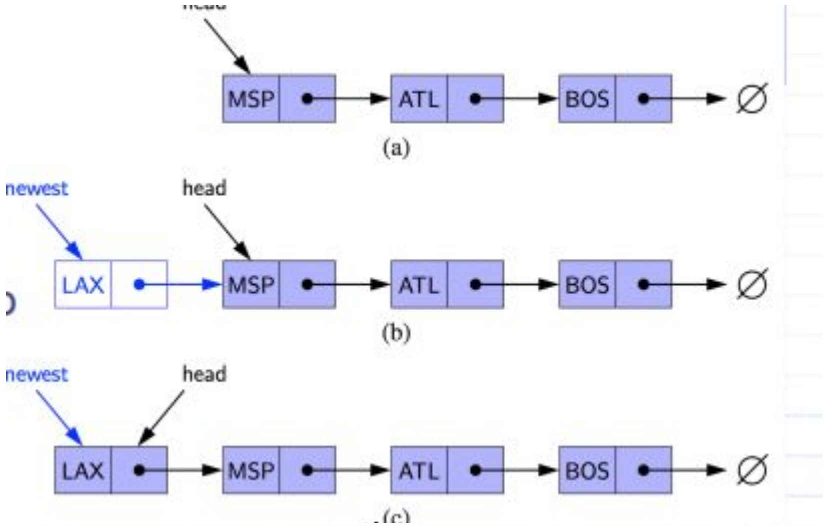
```
public class Node {
    int data;
    Node next;

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


}
```

Singly Linked List

Inserting at the head (prepend)

- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node

- In Singly, we have a reference to the **head** node.




```java
// Method to prepend a new node at the beginning
public void prepend(int data) {
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode;
}
```

Inserting at the tail (append)

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node

```java
// Method to append a new node at the end
public void append(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        return;
    }
    Node last = head;
    while (last.next != null) {
        last = last.next;
    }
    last.next = newNode;
}
```

Removing at the head

- Update head to point to next node in list
- Allow garbage collector to reclaim the former first node
    - If there is no garbage collector, you need to free the memory.
    - (malloc, free)

```
public void removeAtHead() {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }
    head = head.next;
}
```

Removing at the tail

- Not efficient
- You need to start from the top and traverse to the last node.
  - Takes O(n) time
  - No short way
- Then, you need to update the next of the (last - 1) node to null.

```java
// Remove at Tail
public void removeAtTail() {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }
    if (head.next == null) {
        head = null;
        return;
    }
    Node secondLast = head;
    while (secondLast.next.next != null)
        secondLast = secondLast.next;
    secondLast.next = null;
}
```

Details

- Very important
- Implementing data structures
  - Stacks, queues.
  - In other terms, to implement *dynamic* arrays.
- There are usually implementations in programming languages, but they are implemented like this under the hood.
- Dynamic memory management

Complexities

- Access
  - Get Head: O(1)
  - Get Tail: O(1)
  - Access at Index: O(n)
  - Search for Value: O(n)
- Insertion
  - Insert at Head: O(1)
  - Insert at Tail: O(1)
  - Insert at Index: O(n)
- Deletion
  - Delete at Head: O(1)
  - Delete at Tail: O(1)
  - Delete at Index: O(n)
  - Delete by Value: O(n)
- Traversal
  - Traverse: O(n)

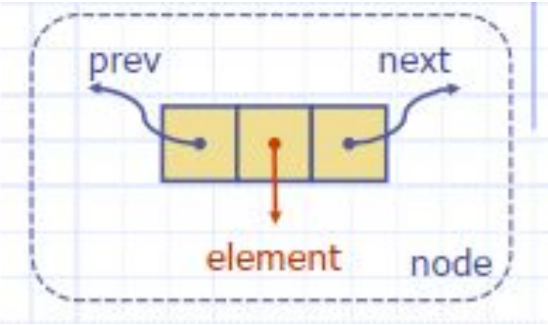
Complexities (Singly Linked List)

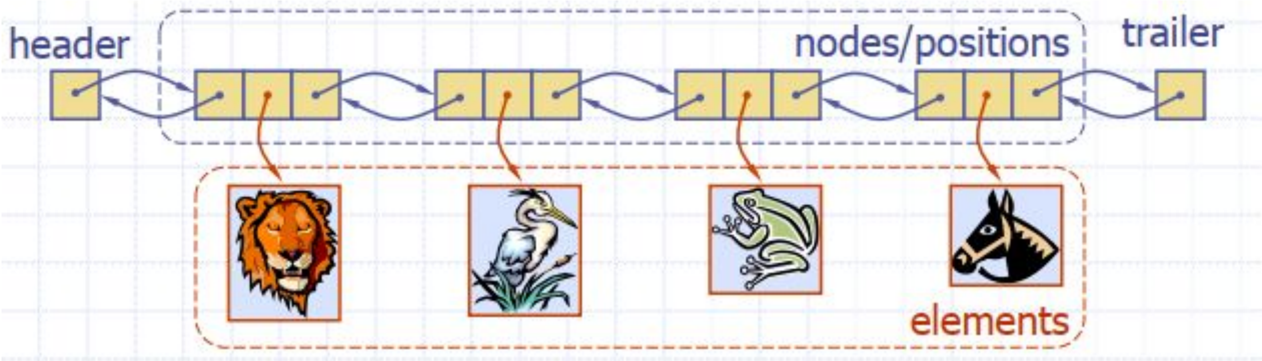
- Access
  - Get Head: O(1)
  - Get Tail: O(n)
  - Access at Index: O(n)
  - Search for value: O(n)
- Insertion
  - Insert at head: O(1)
  - Insert at tail: O(n)
  - Insert at index: O(n)
- Deletion
  - Delete at head: O(1)
  - Delete at tail: O(n)
  - Delete at index: O(n)
  - Delete by value: O(n)
- Traversal
  - O(n)

Doubly linked lists

Doubly Linked List

- Can be traversed forward and backward
- Nodes store:
  - element
  - link to previous node
  - link to next node
- Special trailer and header nodes
- We have access to **head** and **tail**





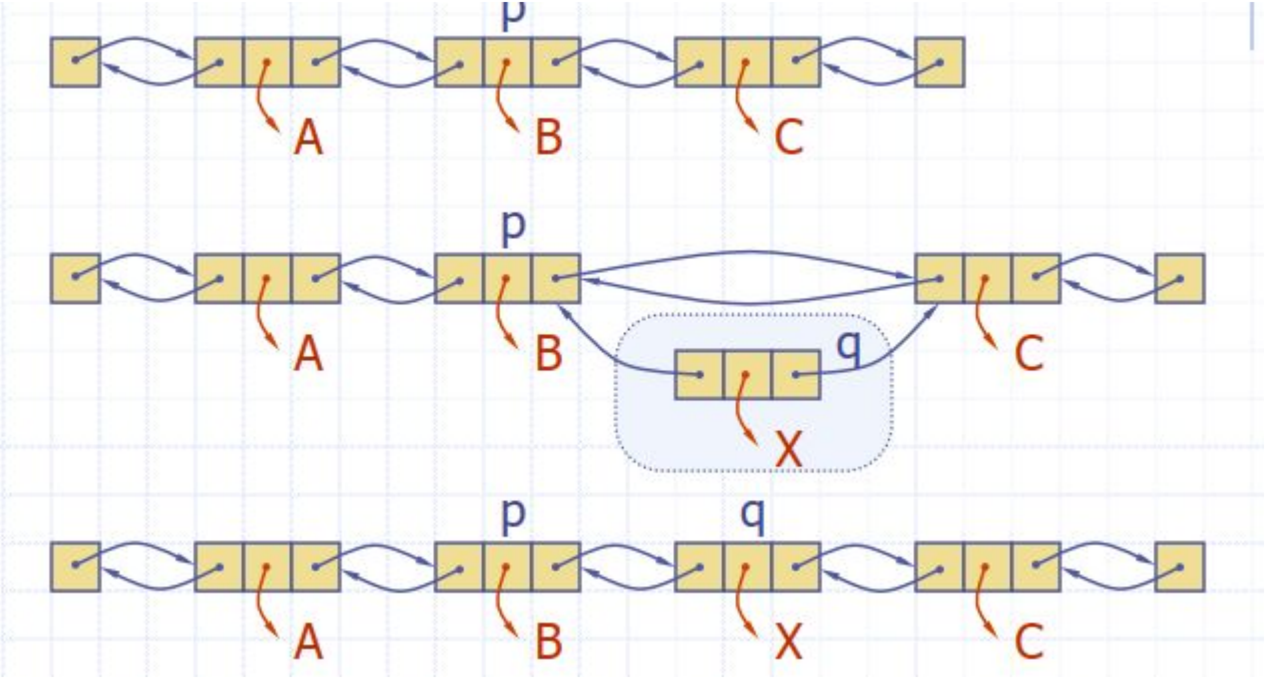
Details

- Node class will be different now.
  - We need to have prev and next.

Details

- Like a lift (elevator)
- In order to go to the top floor, you need to go through all floors.
- You cannot go to a random floor directly.
  - Have to go through other floors.
- Cannot go *beyond* the top floor.
  - Next to the tail node is assigned null
- Cannot go *beyond* the ground floor.
  - Previous to the head node is assigned null

Insertion



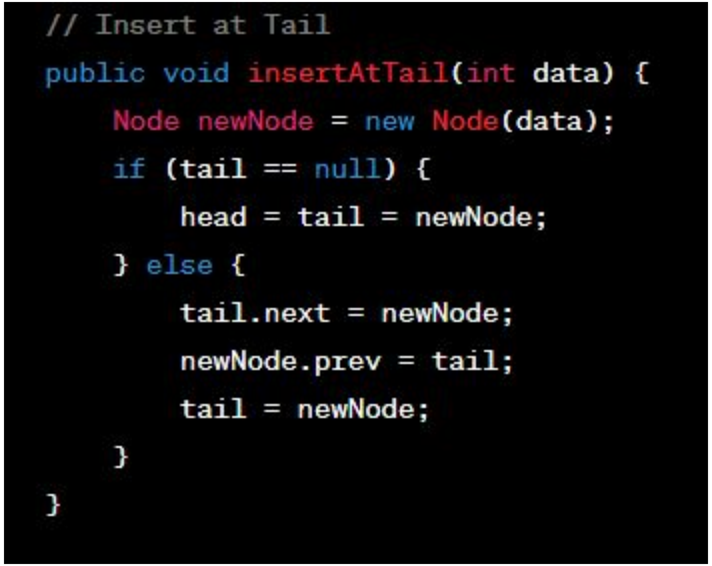
Insertion at the head

- Create a new node
- Update previous pointer
- Update next pointer
    - Because it has a **previous**
- Update head.
- Since we have direct access to **head**, it is O(1).

```java
// Insert at Head
public void insertAtHead(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = tail = newNode;
    } else {
        newNode.next = head;
        head.prev = newNode;
        head = newNode;
    }
}
```

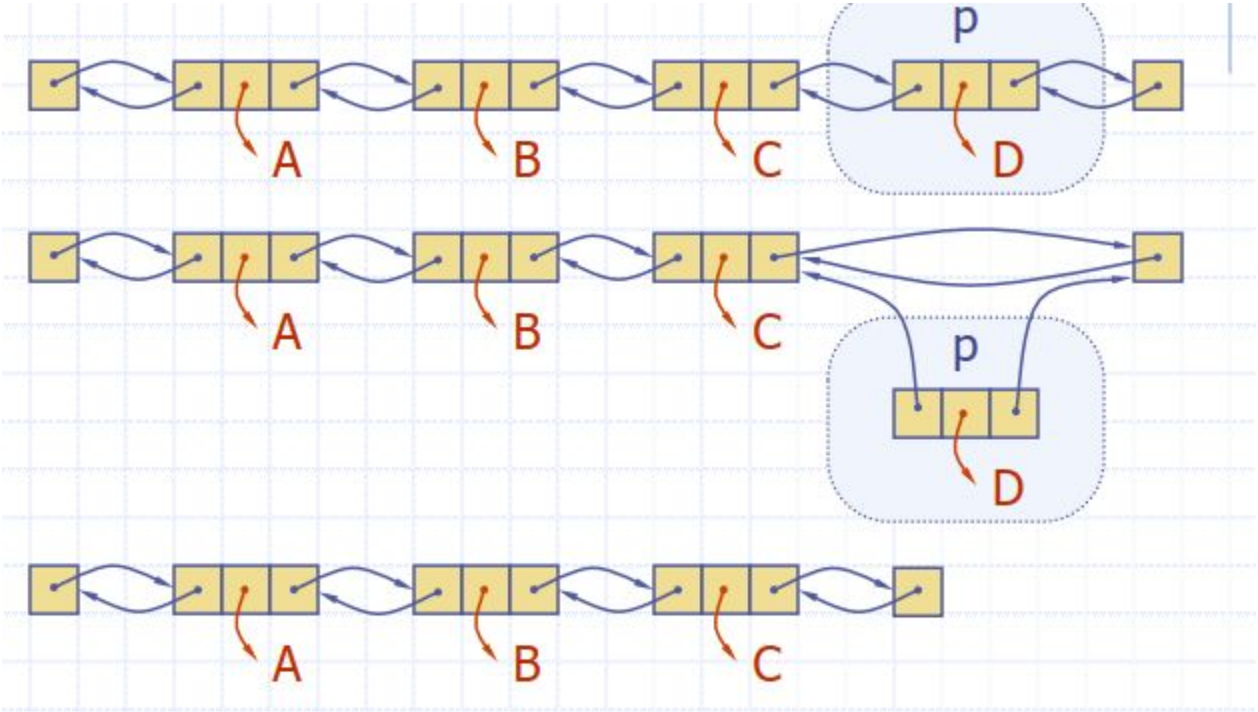
Insertion at the tail

- Create a new node
- Update previous pointer
- Update next pointer
- Update tail
- In a doubly linked list, we also keep a reference of the **tail**, therefore we have direct access to the last node.
    - Therefore, inserting is O(1).



```
// Insert at Tail
public void insertAtTail(int data) {
    Node newNode = new Node(data);
    if (tail == null) {
        head = tail = newNode;
    } else {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
    }
}
```

Deletion



Removing

```java
// Remove at Head
public void removeAtHead() {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }
    head = head.next;
    if (head != null) {
        head.prev = null;
    } else {
        tail = null;
    }
}
```

```java
    // Remove at Tail
    public void removeAtTail() {
        if (tail == null) {
            System.out.println("List is empty");
            return;
        }
        tail = tail.prev;
        if (tail != null) {
            tail.next = null;
        } else {
            head = null;
        }
    }
}
```

Comparison

- ❌ Needs a **previous** in addition to singly.
  - overhead
- ❌ Implementation is more complex.
- ✅ Bidirectional movement.
- ✅ Insertion at both ends is O(1).

Display from head to tail

```java
// Display the content of the list from head to tail
public void displayForward() {
    Node current = head;
    if (head == null) {
        System.out.println("List is empty");
        return;
    }
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```

Display from tail to head

```java
// Display the content of the list from tail to head
public void displayBackward() {
    Node current = tail;
    if (tail == null) {
        System.out.println("List is empty");
        return;
    }
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.prev;
    }
    System.out.println();
}
```

doubly linked list

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

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

    Node head, tail = null;

    // Add a node to the list
```

singly linked list

```
public class SinglyLinkedList {
    class Node {
        int data;
        Node next;

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

    Node head = null;

    // Add a node to the end of the list
    public void addNode(int data) {
        Node newNode = new Node(data);

        if (head == null) {
```