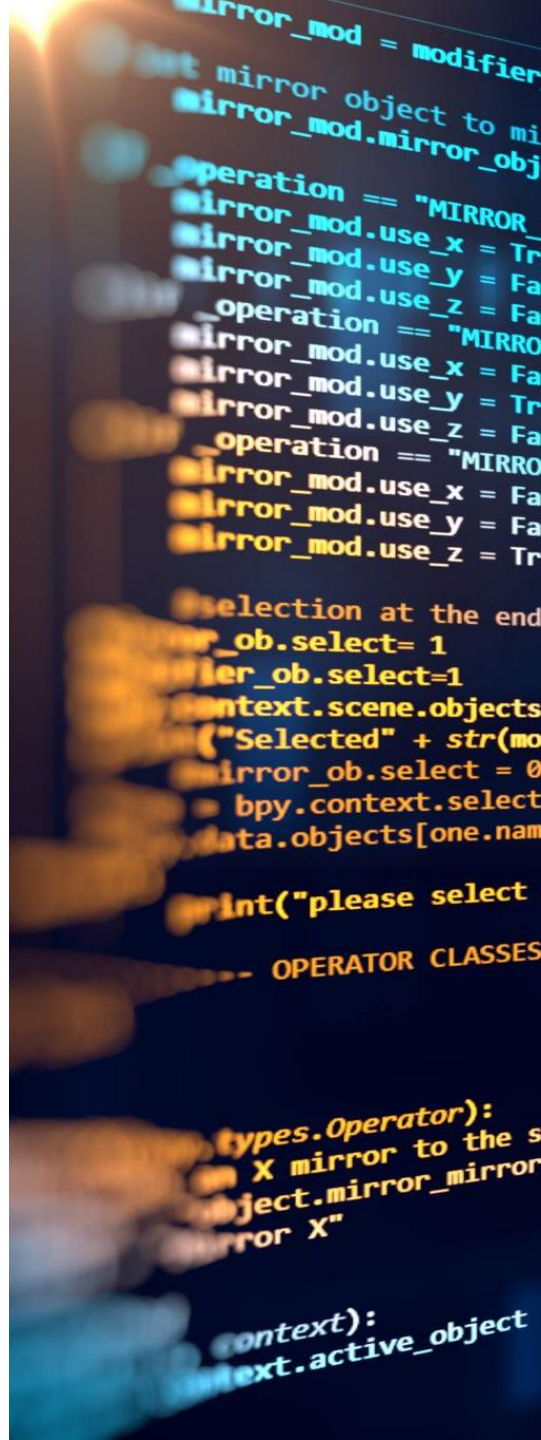
Please use the following QR code to check in and record your attendance.

CS 1027
Fundamentals of Computer Science II

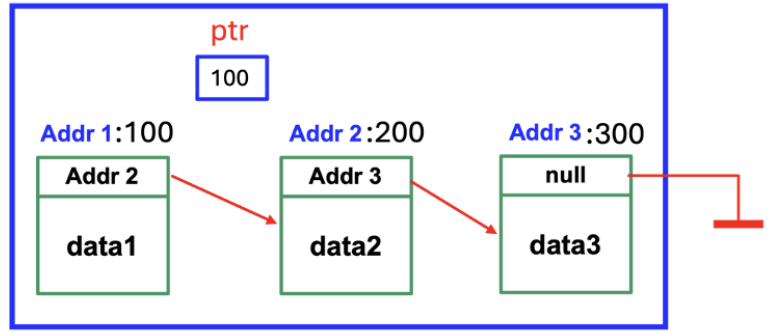
Doubly Linked List

Ahmed Ibrahim

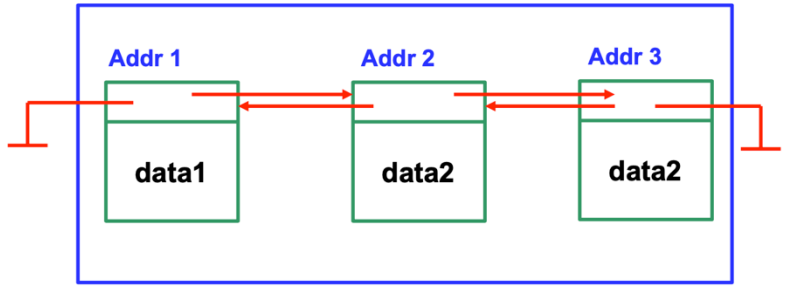


Recap

- Array Limitations
  - Fixed-size and consecutive memory locations limit array flexibility.
  - Insertions and deletions often require shifting elements, which is inefficient.
- Linked Data Structures Overview
  - A linked data structure consists of items linked together by pointers.
  - Each node has a data field and a next pointer that points to the next node in the sequence.



Singly-linked list in memory



Doubly-linked list in memory

Recap cont.

- Linked List Class Design
- Insertion operation:
  - At the front, The new node becomes the head.
  - In the middle, Insert the node between two existing nodes.
  - At the end, The new node is linked after the last node.
- Deletion operation:
  - From the front, The head is removed, and the second node becomes the new head.
  - From the middle, Remove a node and link the previous node to the next one.
  - From the end, Remove the last node and set the second-to-last node's pointer to null.
- Traversal: Traverse through all the list nodes, typically from the head to the tail.
- Search: Search for a node containing a specific value in the list.

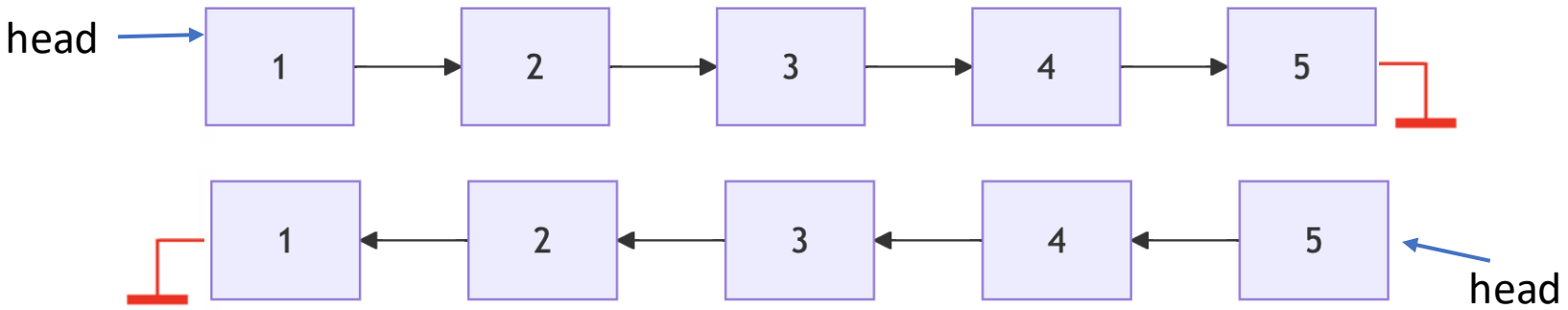
Basic Operations

Advanced Operations: **Reverse a List**

Reverse: Reverse the order of nodes in the list.

```
// Function to reverse the linked list
public Node reverseLinkedList(Node head)
{
        Node prevNode = null;
        Node nextNode = null;
        Node current = head;
while (current != null) {
        nextNode = current.next;
        current.next = prevNode;
        prevNode = current;
        current = nextNode;
 }
 return prevNode;
}
```

Animation source: https://www.youtube.com/watch?v=g_uxlc8C6HE



Advanced Operations: **Merge two Lists**

Merging Two Lists:

Combine two singly linked lists into one.

```
public Node mergeLists(Node head1, Node head2) {
if (head1 == null) return head2;
if (head2 == null) return head1;
Node dummy = new Node(0);
Node current = dummy;
while (head1 != null && head2 != null) {
if (head1.data <= head2.data) {
          current.next = head1;
          head1 = head1.next;
         } else {
          current.next = head2;
          head2 = head2.next;
         }
         current = current.next;
    }
 if (head1 != null) {
  current.next = head1;
 } else if (head2 != null) {
  current.next = head2;
 }
return dummy.next;
}
```

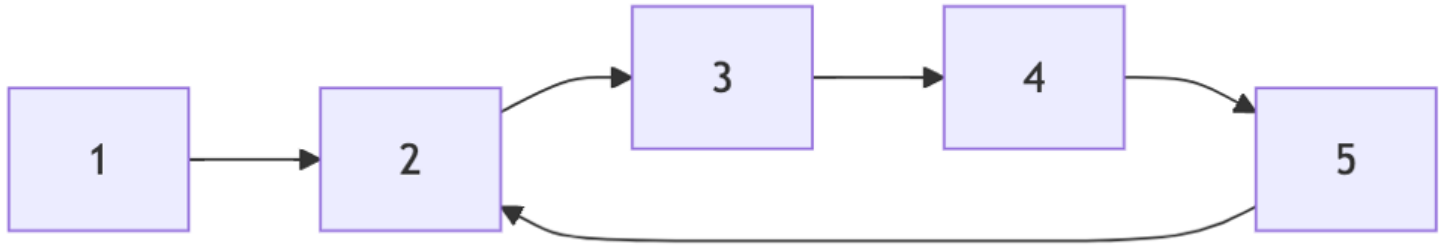
Animation source: https://www.youtube.com/watch?v=g_uxlc8C6HE

More Advanced Operations

- **Removing Duplicates**: Eliminate duplicate nodes from the list.
- **Splitting the List**: Split the linked list into two sub-lists based on some criteria (e.g., by index or value).
- **Clone the List**: Create an exact replica of the linked list.
- **Detecting a Cycle**: Check if the list contains a cycle using algorithms like Floyd's Cycle Detection (Tortoise and Hare).
- **Find Middle Node**: Identify the middle node in the list using two-pointer techniques.
- **Nth Node from the End**: Find the nth node from the end of the list using two-pointer techniques.

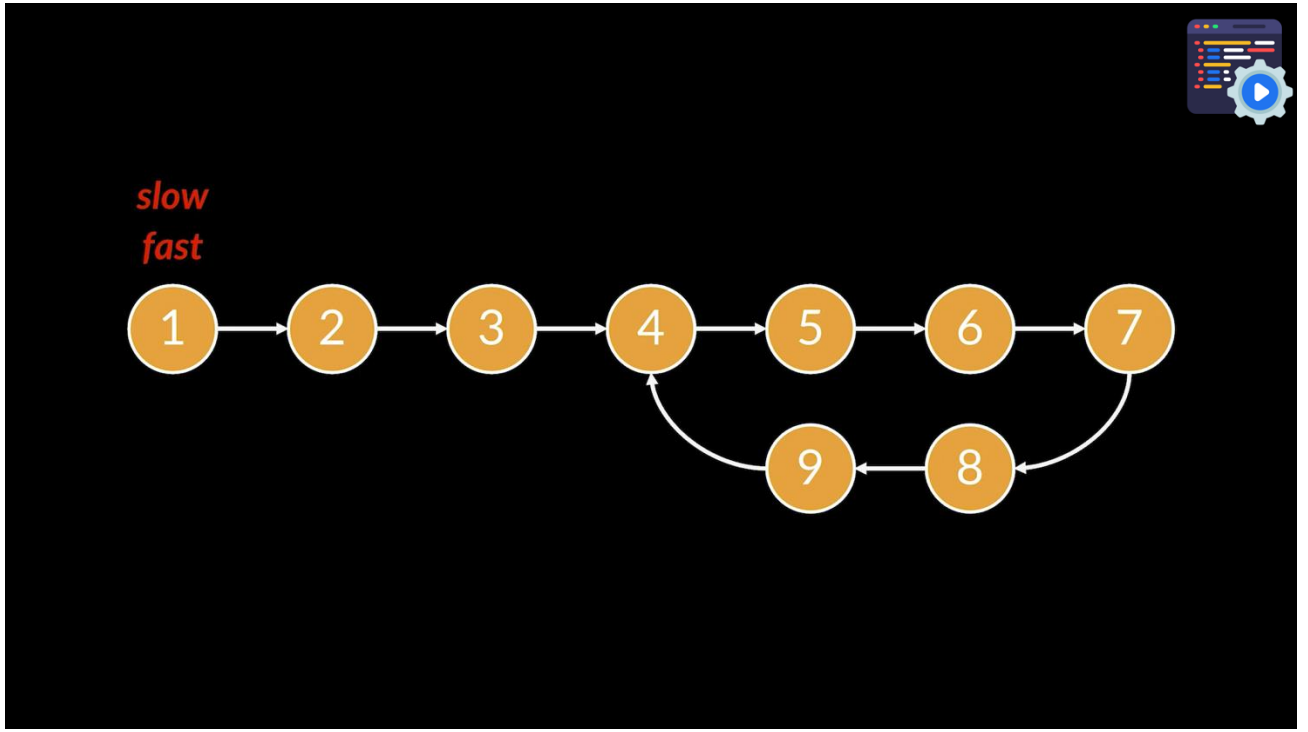
Detecting a Cycle in a Singly Linked List

- A singly linked list can potentially have a cycle, where a node's next pointer references a previous node, creating an infinite loop in the list. The goal is to detect whether such a cycle exists.
- Why?
  - **Avoid Infinite Loops**: lead to an infinite loop if not detected.
  - **Efficiency**: Detecting cycles early prevents unnecessary computations.



Floyd’s Cycle Detection Algorithm (Tortoise and Hare)

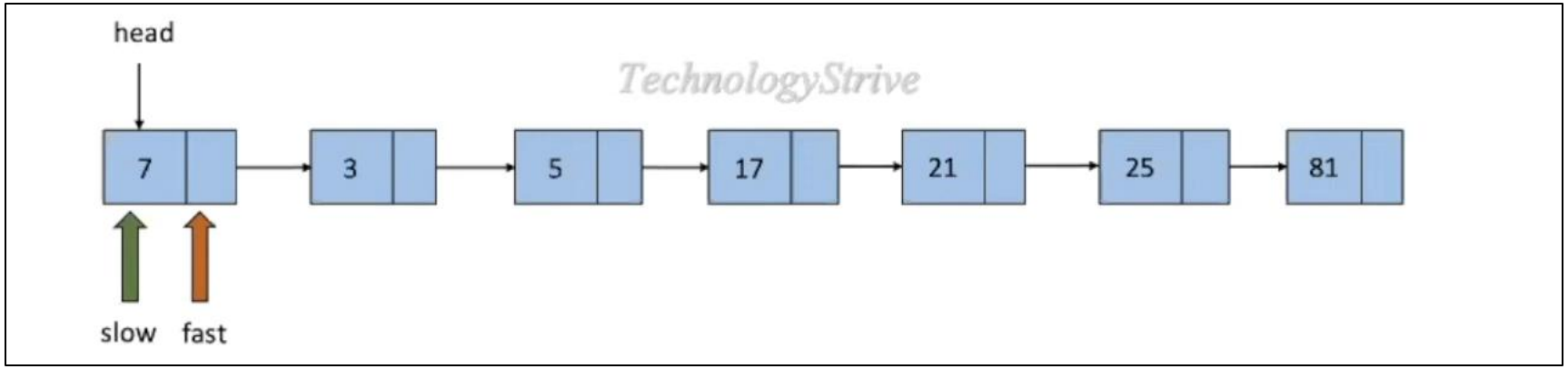
- **Floyd’s Cycle Detection algorithm** uses two pointers:
  - **Tortoise** (slow pointer): Moves one step at a time.
  - **Hare** (fast pointer): Moves two steps at a time.
- The idea is that if there is a cycle, the fast pointer (Hare) will eventually "**lap**" the slow pointer (Tortoise).
- If there is **no cycle**, the Hare will reach the end of the list.



Source: https://youtu.be/S5TcPmTl6ww

Finding the Middle Node

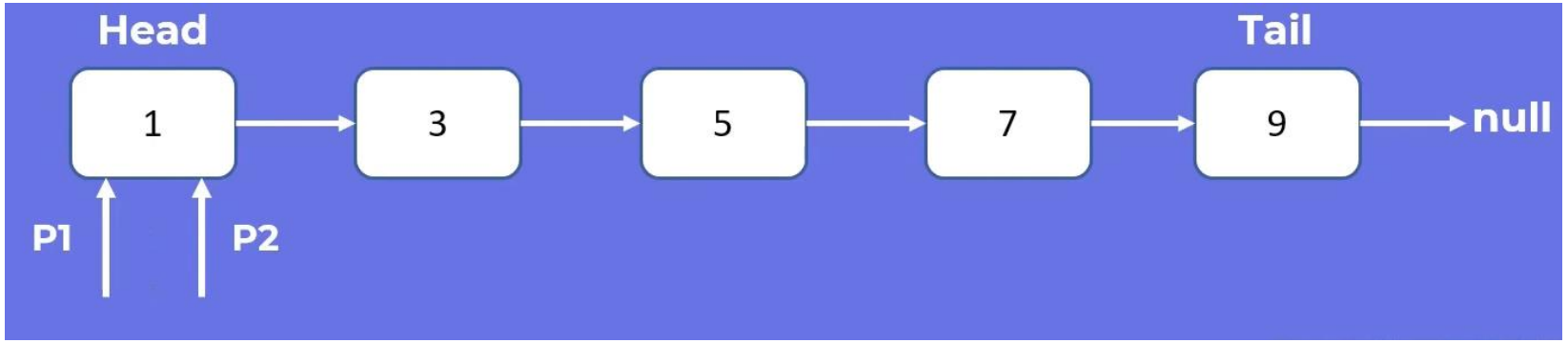
- Given a singly linked list, the task is to find the **middle node** in a single traversal without knowing the list's length in advance.
- Finding the middle of a list is a common problem in algorithms, and optimizing performance requires performing it in a **single traversal using minimal memory**.



Source: https://youtu.be/sqK_gZlecJU

Finding the Nth Node from the End

- Given a singly linked list, the task is to find the **nth node** from the end of the list.
- The challenge is to do this in a **single traversal** without knowing the list's length in advance.
- If the fast pointer reaches the end of the list before completing n steps, this means the list has fewer than n nodes, and the task cannot be completed.



Source: https://www.youtube.com/watch?v=dKFvYm3P6QY

Question!

Consider the following Java method for traversing a linked list:

```
public void traverse(Node head) {
    Node current = head;
    while (current != null) {
        System.out.println(current.data);
        current = current.next;
    }
}
```

Which of the following inputs can lead to a NullPointerException?

A) A list with one node where the next pointer is null.

B) An empty list (head == null).

C) A list with an odd number of nodes.

D) None of the above

Question!

Which of the following scenarios could cause a problem in other parts of the code that rely on the ***traverse*** method if the head is null?

A) The traverse method attempts to print the data of each node, but the list is empty (head == null).

B) Another method assumes that the head is non-null and calls traverse, but the list is empty (head == null).

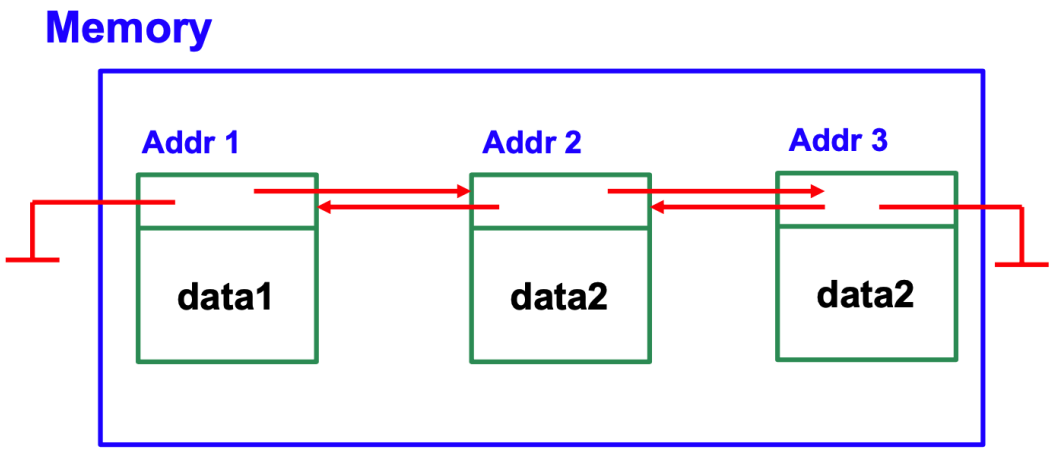
C) The list contains only one node; traverse is called on this single-node list.

D) A linked list is passed to traverse with no null head.

```
public void traverse(Node head) {
    Node current = head;
    while (current != null) {
        System.out.println(current.data);
        current = current.next;
    }
}
```

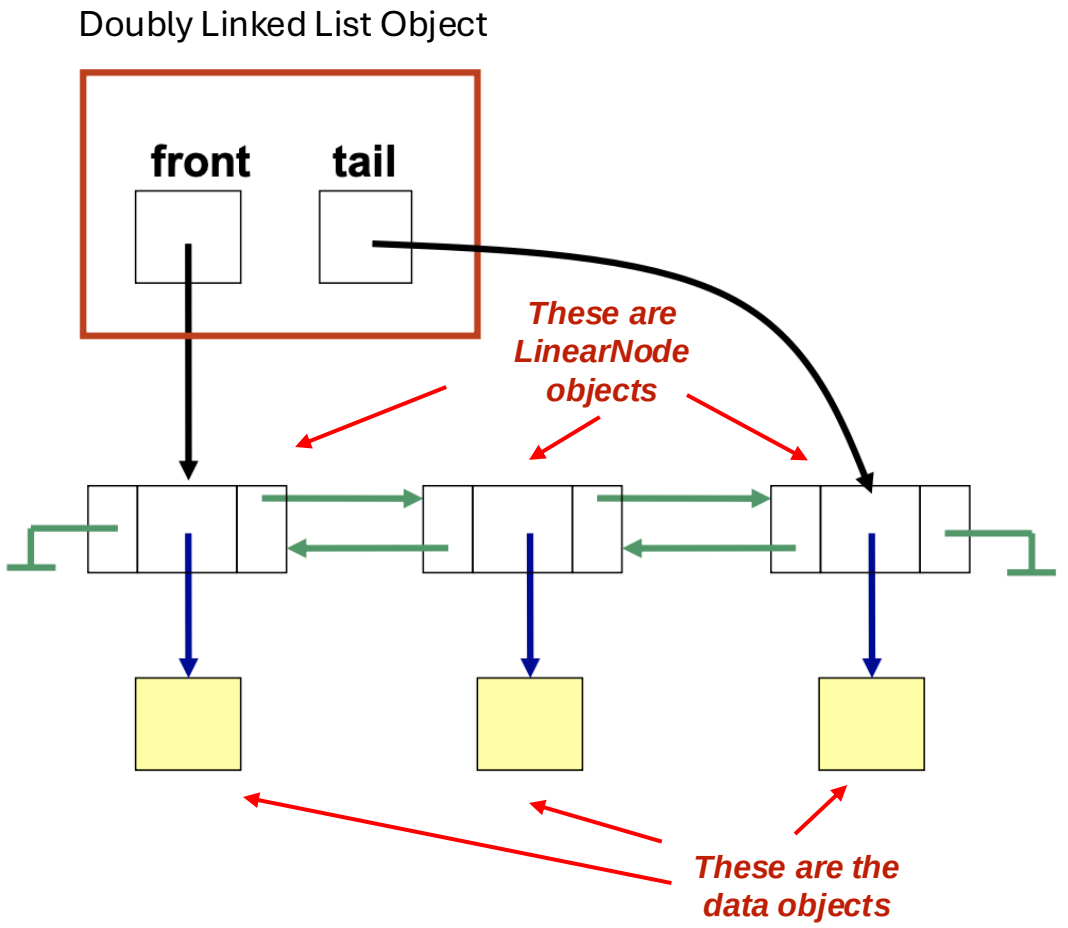
Recall: Doubly Linked Data Structures

- A doubly linked list is a type of linked list where each node contains three fields:
  - **Data**: The value stored in the node.
  - **Next Pointer**: A reference to the next node in the sequence.
  - **Previous Pointer**: A reference to the previous node in the sequence.
- The first node's previous pointer is NULL, and the last node's next pointer is NULL, signifying the list's boundaries.

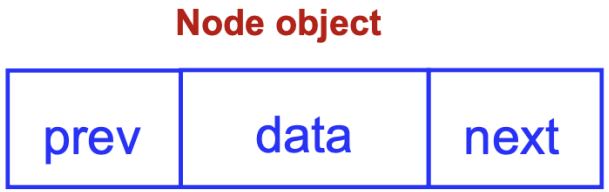


Doubly Linked List Object

- The front pointer maintains a reference to the **first node** in the list.
- The tail pointer maintains a reference to the **last node** in the list.
- The front and tail pointers efficiently access both ends of the list. This allows insertion, deletion, and traversal operations to be performed in **constant time** (**O(1)**) at either end.
- Bidirectional flexibility & optimized List operations



Node Class




```java
public class Node {
    // Data stored in the node
    int data;
    // Pointers to the previous and next nodes
    Node prev;
    Node next;

    // Constructor to create a new node
    public Node(int data) {
        this.data = data;
        this.prev = null; // Initialize previous pointer to null
        this.next = null; // Initialize next pointer to null
    }

}
```

Doubly Linked List Class

- The implementation allows efficient insertions and deletions at the list's head and tail.
- The Node class is designed to store both the next and prev pointers for each node, making the doubly linked list bidirectional.

```java
public class DoublyLinkedList {

    // Head and Tail pointers to maintain the start and end of the list
    private Node head;
    private Node tail;

    // Constructor to initialize an empty list
    public DoublyLinkedList() {
        this.head = null;
        this.tail = null;
    }

    // Method to check if the list is empty
    public boolean isEmpty() {
        return head == null;
    }
```

Doubly vs. **Singly** Linked List

Doubly Linked List:

- **Pros**: It allows bidirectional traversal and efficient insertion/deletion from both ends or the middle of the list.
- **Cons**: Higher memory usage is due to storing two-pointers, which is more complex to implement.

Singly Linked List:

- **Pros**: It is simpler and uses less memory, which makes it good for scenarios where only forward traversal is needed.
- **Cons**: Inefficient for operations that require traversal or insertion/deletion from the middle or end, as there is no easy way to move backwards.

Optimizing Node Design

- If a node is designed to store data of a **specific type**, then creating multiple linked lists that store different data types would require several nearly **identical** implementations of the node class.
- A more efficient solution is to create a node class that can handle any data type.
- This flexibility can be achieved through the use of **Generic Types**.

Generic Types

Generic Types

- Generic types (or just generics) allow us to make classes that work for any data type.
- To do this, the class definition needs a parameter.
- Class parameters are enclosed between angle brackets: <T>, <E>, <MyType>, …
- Using <T> as the name of a class parameter is conventional, but any name for the parameter is allowed in the < >.
- The generic type is **not** the same as the superclass **Object**.
  - Although both **generics** and **Object** can represent any type of data, they behave differently in terms of type safety and flexibility.

Type Safety

- **Generics**: Provide compile-time type safety. When you use a generic type, you can ensure that the data type is consistent throughout the code, reducing the risk of runtime errors due to type mismatches.
  - **Example**: If you define Node<T>, where T is the generic type, the **compiler** will **enforce** that all instances of the node are of the **same type** (e.g., Node<Integer>, Node<String>).

• **Object Class**: The Object class can hold any type of data, but it doesn't provide **type safety.**

• When retrieving data, you would need to cast objects to the appropriate type, which can lead to runtime exceptions if the wrong type is used.

- **Example**: If you use Object to store data, you must manually cast it when retrieving (e.g., String myString = (String) objectInstance;

Generic Types (cont.)

- When working with **generics** in Java, we use wrapper classes such as Integer, Double, Boolean, etc., as class parameters.
- The type of data items in the LinearNode class is determined at runtime when an application program creates an object of that class. This is known as **type erasure** in Java **generics**.
- Note that angle brackets <> specify the type parameter, which tells Java what data the LinearNode will store.
- The type parameter can be any class, including wrapper classes and user-defined classes.

```
// Using String class
LinearNode<String> s = new LinearNode<>("Hello");

// Using Integer wrapper class
LinearNode<Integer> n = new LinearNode<>(42);

// Using Double wrapper class
LinearNode<Double> d = new LinearNode<>(3.14);

// Using custom classes
LinearNode<Person> p = new LinearNode<>(new Person());
LinearNode<Rectangle> r = new LinearNode<>(new Rectangle());
```

LinearNode Class

```
public class LinearNode<T> {
// Pointer to the next node
private LinearNode<T> next;
// Data stored in the node
private T dataItem;
// Default constructor
public LinearNode() {
next = null;
dataItem = null;
  }
// Constructor with a data value
public LinearNode(T value) {
next = null;
dataItem = value;
  }
}
```

Write algorithms to add a new node to a doubly linked list and to remove a node from a doubly linked list.

Question!

In a doubly linked list, what happens when you delete a node in the middle of the list?

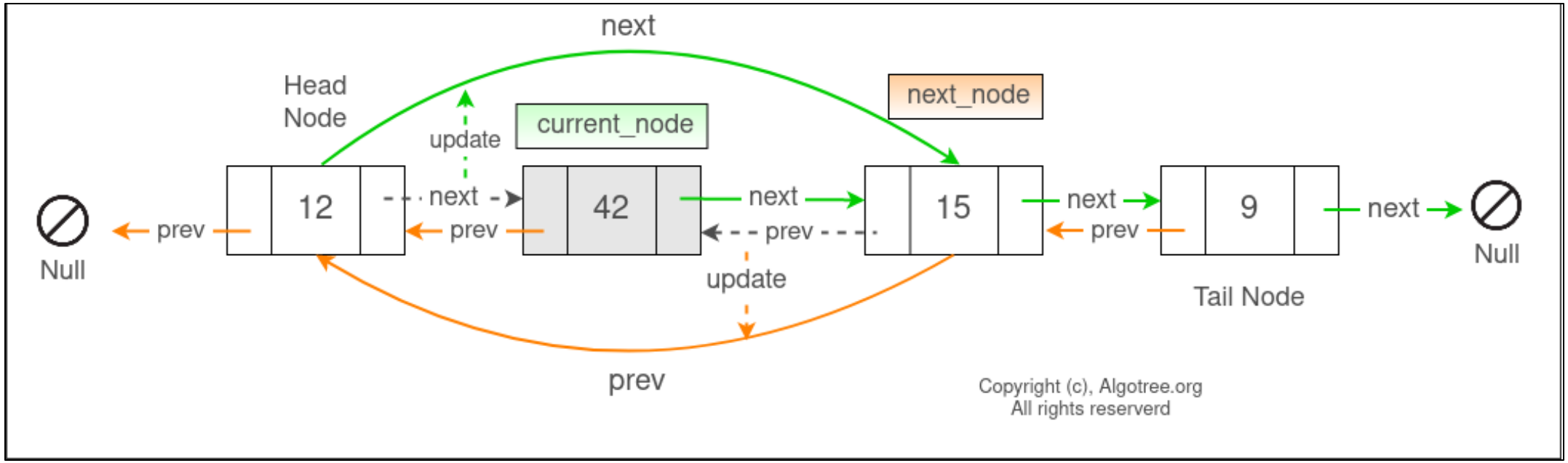
A) The previous pointer of the node before the deleted node is updated to point to the next node, and the next pointer of the next node is updated to point to the previous node of the deleted node.

B) Only the next pointer of the node before the deleted node needs to be updated to point to the next node.

C) Only the previous pointer of the next node needs to be updated to point to the previous node.

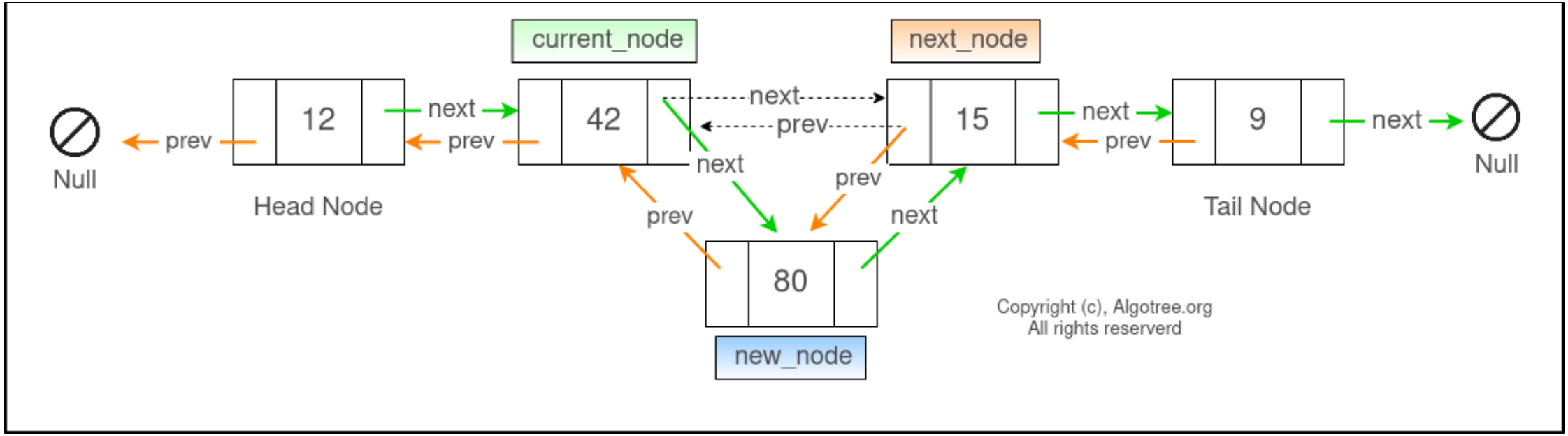
D) Both the next pointer of the previous node and the previous pointer of the next node must be updated to bypass the deleted node.

Deleting a node from a Doubly Linked List



Source: https://www.algotree.org/algorithms/linkedlists/doubly_linked_list/

Insert a node from a Doubly Linked List



Source: https://www.algotree.org/algorithms/linkedlists/doubly_linked_list/

Question!

Consider a **generic** doubly linked list implementation. What is the most efficient way to iterate over the list from the head to the tail and print each node's data?

A) Use two pointers, one starting from the head and one from the tail, traverse the list in both directions.

B) Use a single pointer starting from the head and moving forward until the tail is reached.

C) Use recursion to visit each node starting from the head.

D) Use the next pointer of each node, then reverse the list and use the previous pointer.

When might you need two pointers in a doubly linked list scenario?

Scenarios!

- When you need to find the **middle node** of the list in a single traversal.
- If you suspect, there's a cycle in the doubly linked list.
- You need to reverse a specific section of the list.
  - Mark the start and end of the section with two pointers, then reverse the nodes between them by adjusting their pointers.
- When you want to merge two sorted doubly linked lists into one.
- Comparing Two Lists
- Partitioning a List

Question!

You want to delete the last node in a **generic** doubly linked list. Which of the following pointer updates is correct?

A) Set the tail to null

B) Set the tail.previous.next to null and update the tail pointer to tail.previous

C) Set the head to null and update the tail.next pointer to null

(D) Set the tail.next pointer to the head and delete the last node

Question!

You want to implement a doubly linked list that can store nodes containing two different data types (e.g., a String and an Integer). Which of the following changes would allow you to implement this in Java?

A) Implement two separate generic linked lists, one for each data type

B) Modify the Node class to accept two **generic** parameters like Node<T1, T2>

C) Use a single **generic** parameter Node<Object> and cast each node's data to the appropriate type.

D) You cannot implement this with **generics**.

Node Class with two **generic** types, T1 and T2

```
class Node<T1, T2> {
        T1 data1; // First data of type T1 (e.g., String)
        T2 data2; // Second data of type T2 (e.g., Integer)
        Node<T1, T2> prev; // Previous node reference
        Node<T1, T2> next; // Next node reference

        // Constructor
        public Node(T1 data1, T2 data2) {
        this.data1 = data1;
        this.data2 = data2;
        this.prev = null;
        this.next = null;
        }
}
```


thank you