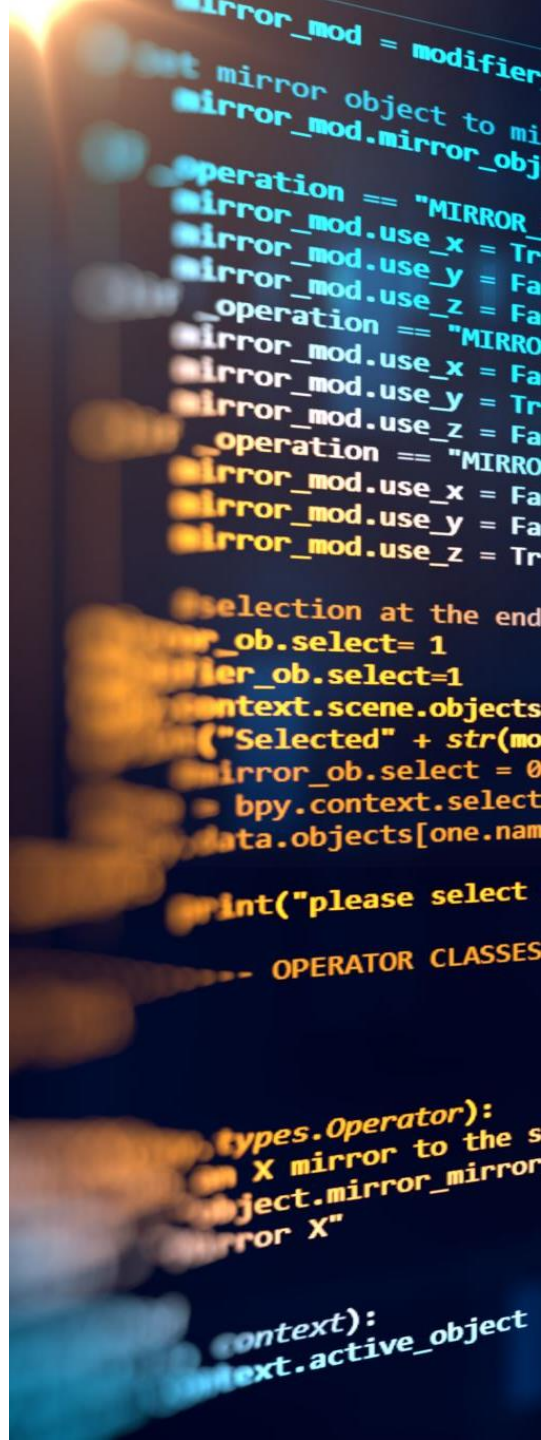
Please use the following QR code to check in and record your attendance.

CS 1027
Fundamentals of Computer Science II

Linked Data Structures

Ahmed Ibrahim

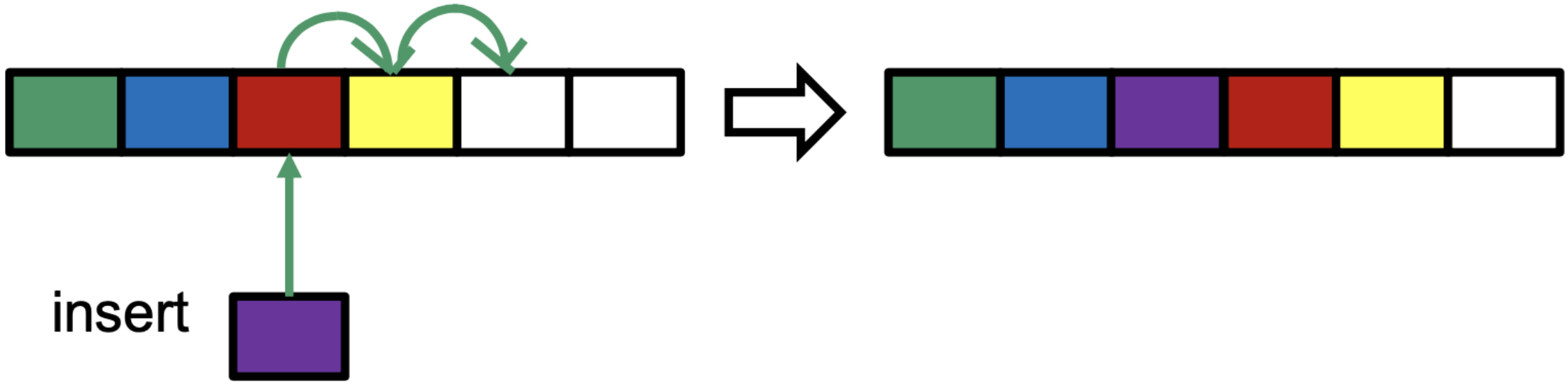


Objectives

- Describe linked structures
- Compare linked structures to array-based structures
- Explore the techniques for managing a linked list

Array Limitations

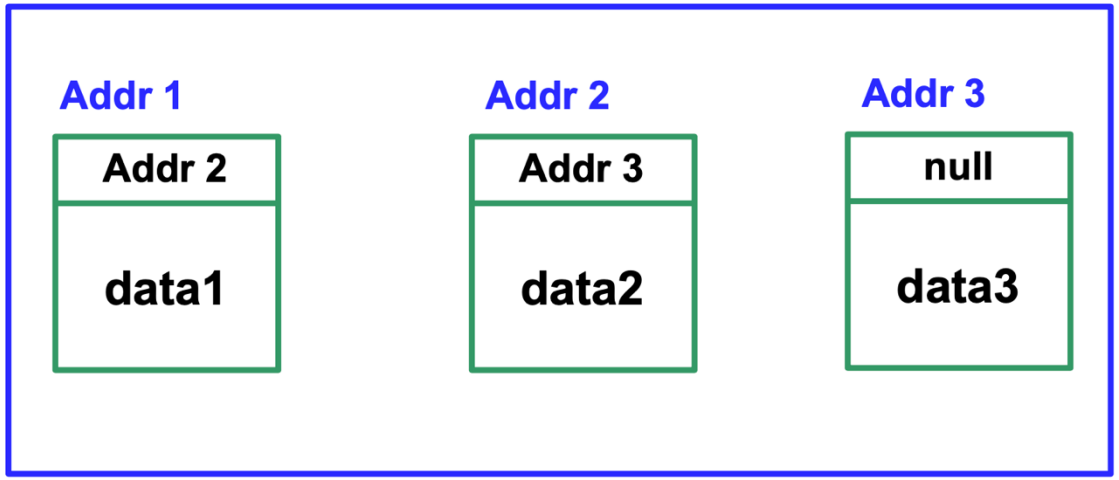
- What are the limitations of an array, as a data structure?
  - Fixed-size
  - Physically stored in consecutive memory locations
  - To insert or delete items, you may need to shift data



Linked Data Structures

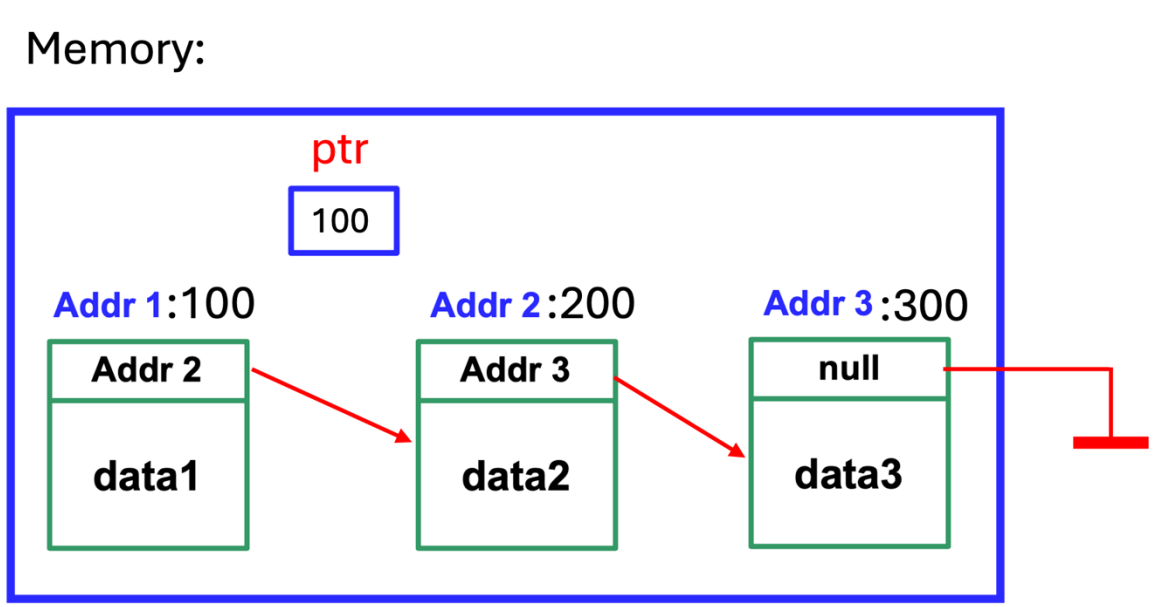
- A *linked* data structure consists of items that are linked to other items
- How? each item **points** to another item

Memory:




```java
public class Node {

// Reference to the next node
private Node next;

// Data in the node
private String data;

// Constructor to initialize the node
public Node(String data) {
   this.next = null;
   this.data = data;
}
```

Linked Data Structures (cont.)

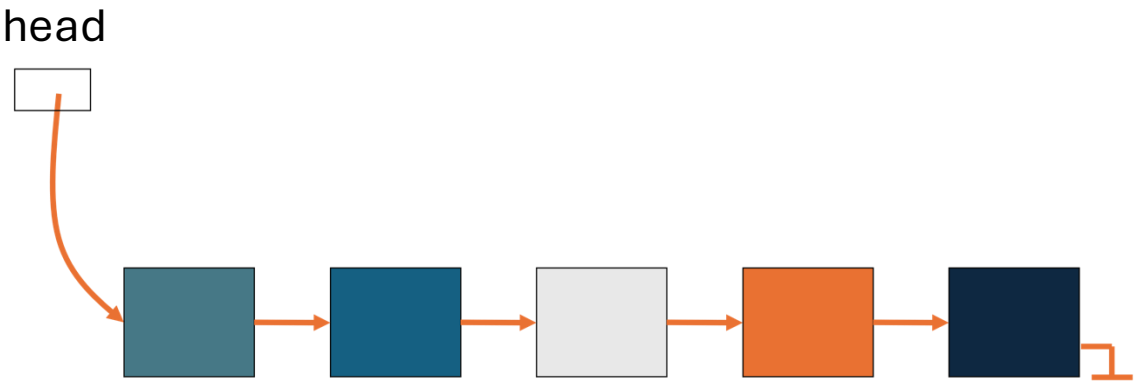


Singly-linked list in memory

```java
public class Node {

// Reference to the next node
private Node next;

// Data in the node
private String data;

// Constructor to initialize the node
public Node(String data) {
   this.next = null;
   this.data = data;
}
```

Linear Linked Data Structures

- A Linear or Singly linked list is a data structure in which each element (node) contains:
  - **Data**: The actual value stored in the node.
  - **Next Pointer**: A reference (or pointer) to the next node in the sequence.
- The last node's next pointer is set to null, indicating the end of the list.
- Characteristics:
  - Dynamic size: Nodes can be added or removed at runtime.
  - Each node points to the next one, forming a chain of nodes.



Conceptual Diagram of a Singly-Linked List

Advantages of Linked Lists

- The items **do not** have to be stored in consecutive memory locations; the successor can be anywhere physically.
  - So, you can insert and delete items without shifting data
  - Can increase the size of the data structure easily
- Linked lists can grow dynamically (i.e. at run time) – the amount of memory space allocated can grow and shrink as needed.

LinearList Class Design

- We need to design a class called LinearList that stores the head of the list (a reference to the first node).
- The class should also manage nodes in the linked list by providing methods for **adding**, **deleting**, and **traversing** nodes.
- The head node is initially set to NULL when the list is empty.

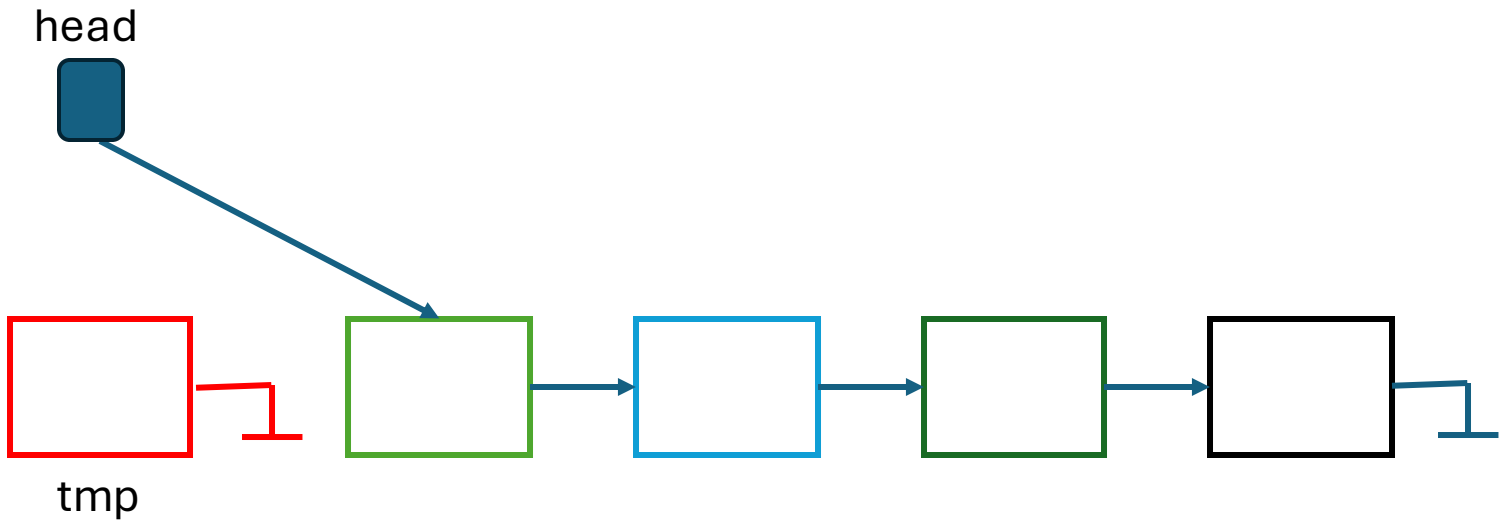
```
public class LinearList {
 // The head of the list (first node)
    private Node head;

// Constructor to create an empty list
    public LinearList()
{
 // The list starts off as empty
 // (head is null)
        head = null;
    }
```

Linked List Operations

Insert Node to a Linked List

1. Create a new node:

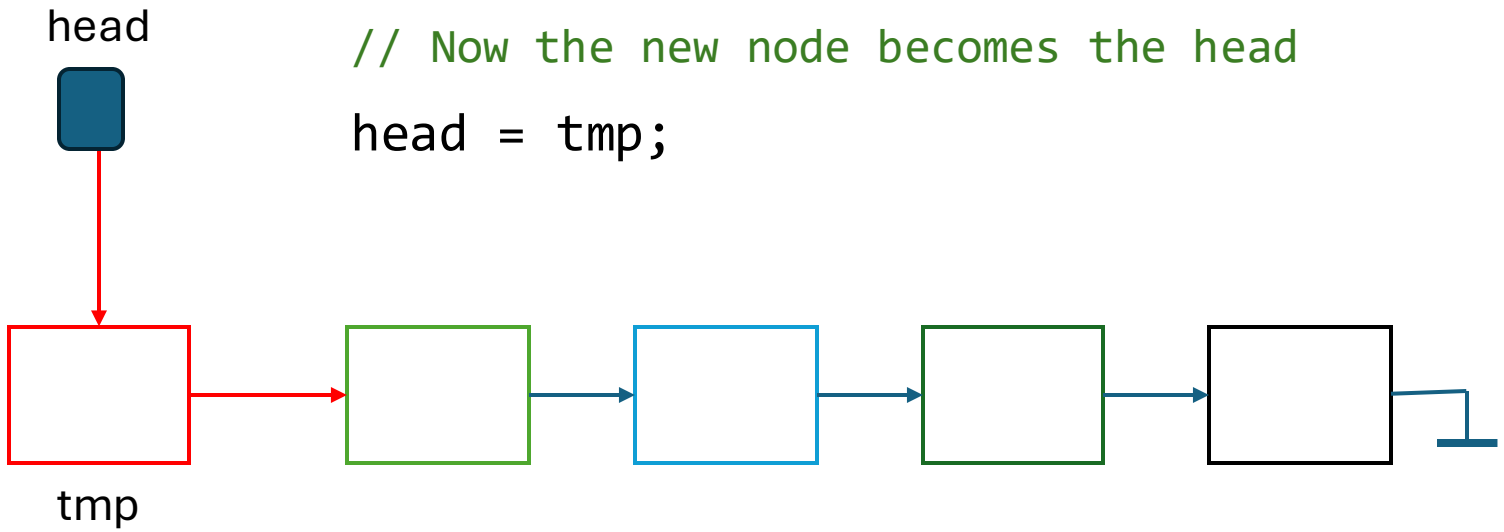
```
// Create a new node with data
        Node tmp = new Node(data);
```



Insert Node to a Linked List (cont.)

2. Insert the new node:

```
// New node points to the current front
tmp.next = head;
// Now the new node becomes the head
head = tmp;
```



Linked lists can grow and shrink dynamically (i.e., at run time).

LinearList Class Design (cont.)

- The add() method adds a new Node to the front of the list.
- By adding to the front, we can maintain the flexibility of the list growing in size as needed.
- Use the isEmpty method to check if the list is empty.
- Without checking if the list is empty, methods that assume the existence of nodes (like accessing or deleting nodes) could result in null pointer exceptions.

```
// Method to add a node to the front of the list
public void add(String data)
{
// Create a new node with the given data
Node tmp = new Node(data);

// The new node points to the current
tmp.next = head;

// Now the new node becomes the head
head = tmp;
}

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

LinearList Class Design (cont.)

- This Main class is designed to demonstrate the basic functionality of the LinearList class.
  1. It creates an empty linked list.
  2. It adds nodes to the list with the add() method.
  3. It checks if the list is empty with isEmpty().

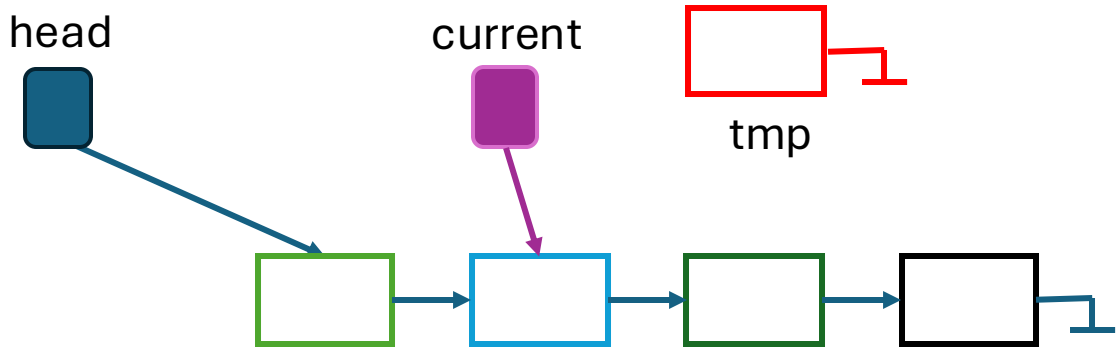
Note: We will hereafter refer to a singly linked list just as a "linked list"

```
public class Main {
    public static void main(String[] args) {
        // Create a new linked list
        LinearList list = new LinearList();


        // Add nodes with string data
        list.add("Node 1");
        list.add("Node 2");
        list.add("Node 3");


        // Check if the list is empty
     System.out.println("Is the list empty? " +
list.isEmpty());
    }
}
```

Insert Node in the Middle

- Find the Position to Insert the Node:

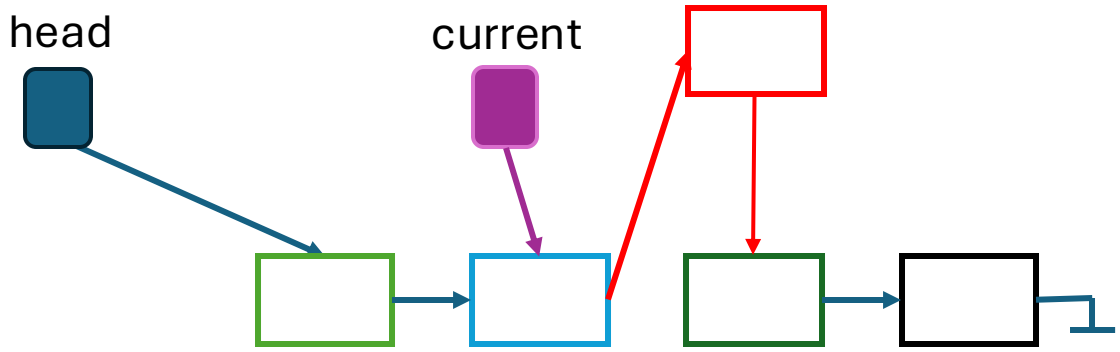
```
// Create a new node with data
Node tmp = new Node(data);
// Find the node before the desired position
Node current = head;
for (int i = 0; i < position - 1 && current != null; i++) {
    // Traverse to the node before the insertion point
    current = current.next;
}
```



Insert Node in the Middle

- Insert the New Node:

```
// The new node's next points to the next node in the list
tmp.next = current.next;
// The previous node (current) points to the new node
current.next = tmp;
```

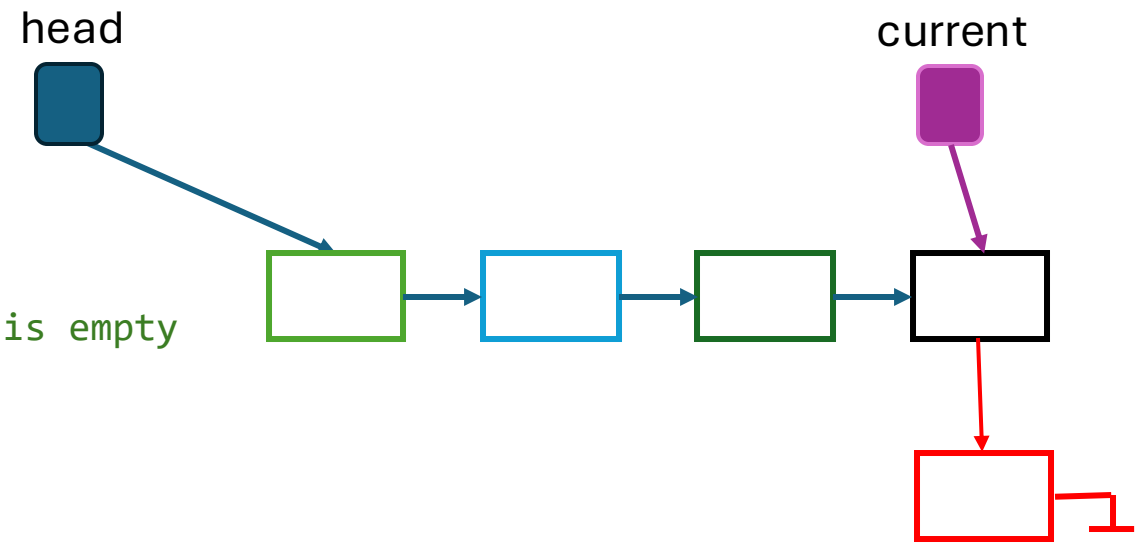


```java
// Method to insert a node at a specific position
public void addInMiddle(String data, int position) {
    Node tmp = new Node(data);

    // If position is 1, insert at the front
    if (position == 1) {
        tmp.next = head;
        head = tmp;
        return;
    }

    // Traverse to the node before the desired position
    Node current = head;
    for (int i = 0; i < position - 1 && current != null; i++) {
        current = current.next;
    }

    // Insert the new node in the correct position
    tmp.next = current.next;
    current.next = tmp;
}
```

Insert Node at the End

- Find the Last Node in the List:

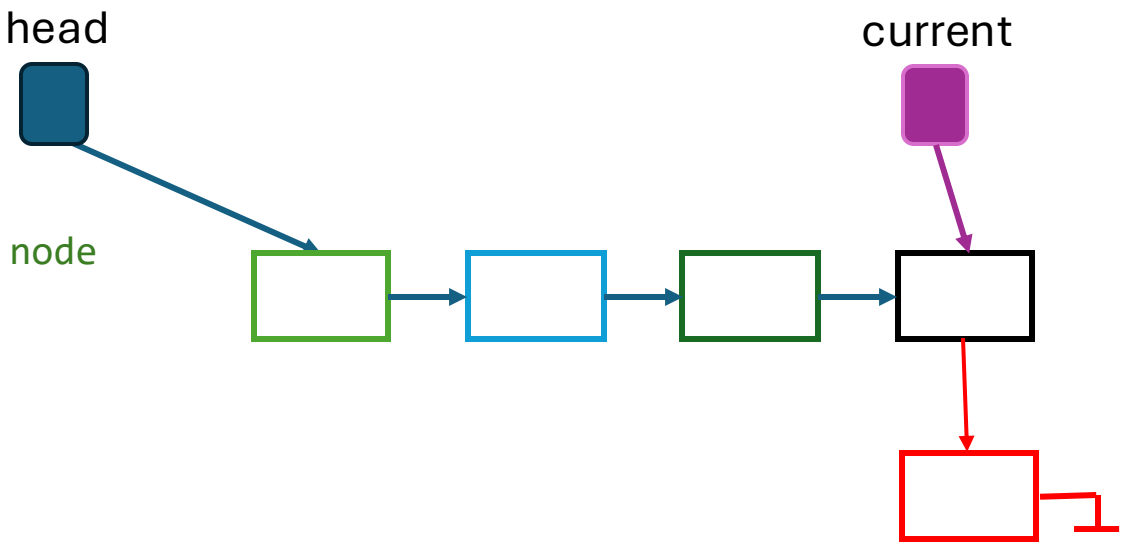
```
// Create a new node with data
Node tmp = new Node(data);
// Check if the list is empty
if (head == null) {
// The new node becomes the head if the list is empty
    head = tmp;
    return;}
// Traverse to the last node in the list
Node current = head;
while (current.next != null) {
// Move to the next node until we reach the end
    current = current.next;}
```



Insert Node at the End

- Insert the New Node:

```
// The last node's next now points to the new node
current.next = tmp;
```



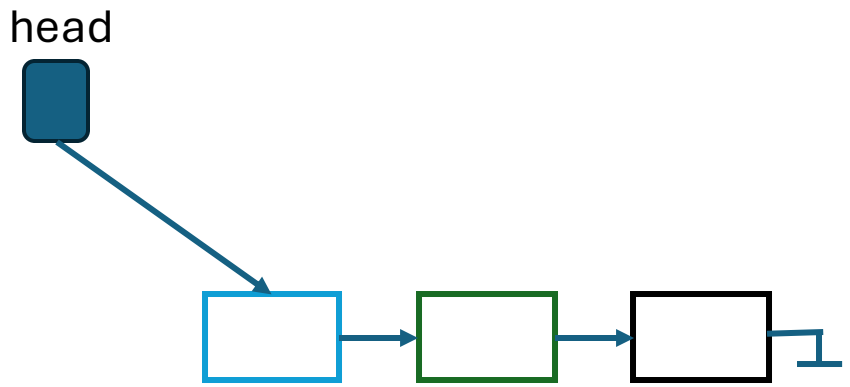
```java
// Method to insert a node at the end of the list
public void addAtEnd(String data) {
    Node tmp = new Node(data);

    // If the list is empty, make the new node the head
    if (head == null) {
        head = tmp;
        return;
    }

    // Traverse to the last node
    Node current = head;
    while (current.next != null) {
        current = current.next;
    }

    // Insert the new node at the end
    current.next = tmp;
}
```

Delete Node from the Front

- Delete the First Node (Head):

```
// Check if the list is empty
if (head == null) {
    System.out.println("List is empty.
Nothing to delete.");
    return;
}
// Move head to the next node
head = head.next;
```

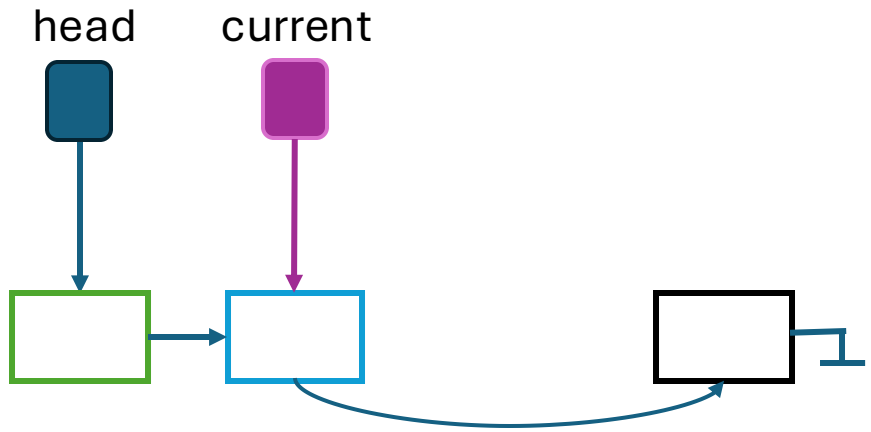
- The previous head node will no longer be referenced and automatically cleared by the garbage collector.



Delete Node from the Middle

- Find the Node to Delete:

```
// Traverse the list to find the node before the one to delete
Node current = head;
for (int i = 0; i < position - 1 && current.next != null; i++) {
    current = current.next;
}
// If the node to delete is beyond the list bounds
if (current.next == null) {
    System.out.println("Position out of bounds.");
    return;
}
// The node to delete is current.next
current.next = current.next.next;
```



Delete Node from the End

- Delete the Last Node:

```
// Check if the list is empty
if (head == null) {
    System.out.println("List is empty. Nothing to
delete.");
    return;
}
// Traverse to the second-to-last node
Node current = head;
while (current.next.next != null) {
    current = current.next;
}
// Set the second-to-last node's next to null
current.next = null;
```

Linked List Traversal

- This operation involves visiting each node in the list and is essential for printing, searching, or processing each node's data.

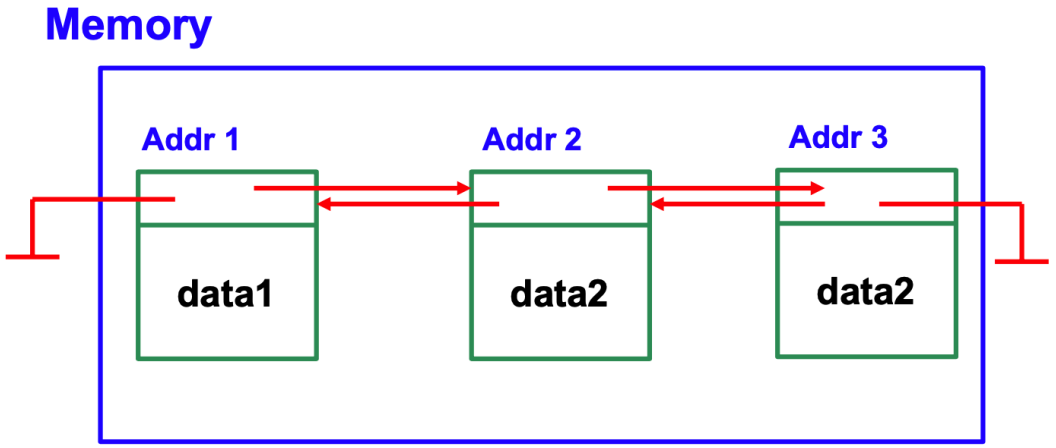
```
public void traverse() {
    Node current = head;
    while (current != null) {
        System.out.println(current.data);
        current = current.next;
    }
}
```

- Searching allows you to find if a particular data element exists in the list.

```
public boolean search(String data) {
    Node current = head;
    while (current != null) {
        if (current.data.equals(data)) {
            return true;  // Data found
        }
        current = current.next;
    }
    return false;  // Data not found
}
```

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.




thank
you