**LinkedList in Java**

A **LinkedList** is a linear data structure where each element (called a node) contains two parts:

1. **Data**: Stores the value of the element.
2. **Pointer/Reference**: Points to the next node in the sequence (or the previous node, in the case of a doubly linked list).

In Java, the LinkedList class is part of the java.util package and implements the List, Deque, and Queue interfaces.

**Types of LinkedList**

1. **Singly Linked List**
   * Each node has data and a reference to the next node.
   * The last node points to null.
2. **Doubly Linked List**
   * Each node has three parts: data, a reference to the next node, and a reference to the previous node.
   * Provides bidirectional traversal.
3. **Circular Linked List**
   * Similar to a singly or doubly linked list, but the last node points back to the first node, forming a circle.

**Example Diagram**

**Singly Linked List**:

[10|next] -> [20|next] -> [30|next] -> null

**Doubly Linked List**:

null <- [prev|10|next] <-> [prev|20|next] <-> [prev|30|next] -> null

**Circular Linked List**:

[10|next] -> [20|next] -> [30|next] --|

^ |

|------------------------------|

**Code Example in Java**

**Singly Linked List**

|  |
| --- |
| class SinglyLinkedList {  class Node {  int data;  Node next;  Node(int data) {  this.data = data;  this.next = null;  }  }  private Node head;  // Add a node at the end  public void add(int data) {  Node newNode = new Node(data);  if (head == null) {  head = newNode;  } else {  Node temp = head;  while (temp.next != null) {  temp = temp.next;  }  temp.next = newNode;  }  }  // Print the linked list  public void printList() {  Node temp = head;  while (temp != null) {  System.out.print(temp.data + " -> ");  temp = temp.next;  }  System.out.println("null");  }  public static void main(String[] args) {  SinglyLinkedList list = new SinglyLinkedList();  list.add(10);  list.add(20);  list.add(30);  list.printList(); // Output: 10 -> 20 -> 30 -> null  }  } |

**Doubly Linked List**

|  |
| --- |
| class DoublyLinkedList {  class Node {  int data;  Node prev, next;  Node(int data) {  this.data = data;  this.prev = this.next = null;  }  }  private Node head;  // Add a node at the end  public void add(int data) {  Node newNode = new Node(data);  if (head == null) {  head = newNode;  } else {  Node temp = head;  while (temp.next != null) {  temp = temp.next;  }  temp.next = newNode;  newNode.prev = temp;  }  }  // Print the linked list  public void printList() {  Node temp = head;  while (temp != null) {  System.out.print(temp.data + " <-> ");  temp = temp.next;  }  System.out.println("null");  }  public static void main(String[] args) {  DoublyLinkedList list = new DoublyLinkedList();  list.add(10);  list.add(20);  list.add(30);  list.printList(); // Output: 10 <-> 20 <-> 30 <-> null  }  } |

**Advantages of LinkedList**

1. **Dynamic Size**: Can grow or shrink dynamically without memory reallocation.
2. **Efficient Insert/Delete**: Inserting or deleting nodes is faster compared to arrays since it doesn’t require shifting elements.
3. **No Wasted Memory**: No need to pre-allocate memory.

**Disadvantages of LinkedList**

1. **Slow Access**: Accessing elements requires traversal, making it slower than arrays.
2. **Extra Memory**: Requires extra memory for pointers.
3. **Cache Unfriendliness**: Nodes are not stored contiguously, leading to poor cache performance.

**Time Complexity**

| **Operation** | **Singly Linked List** | **Doubly Linked List** | **Circular Linked List** |
| --- | --- | --- | --- |
| **Access (by index)** | O(n) | O(n) | O(n) |
| **Insertion** | O(1) (beginning) | O(1) (beginning) | O(1) |
|  | O(n) (end) | O(n) (end) | O(n) |
| **Deletion** | O(1) (beginning) | O(1) (beginning) | O(1) |
|  | O(n) (end) | O(n) (end) | O(n) |
| **Search** | O(n) | O(n) | O(n) |

**Space Complexity**

* **Singly Linked List**: O(n) for n nodes.
* **Doubly Linked List**: O(n) for n nodes (extra memory for the previous pointer).
* **Circular Linked List**: O(n) for n nodes.

**Inserting a Node in a Singly Linked List**

In a **Singly Linked List**, you can insert a new node in three common ways:

1. **At the Start (Head)**
2. **At the End (Tail)**
3. **At a Specific Position**

Let’s explore each method in detail with step-by-step explanations and code examples.

**1. Insert at the Start (Head)**

**Steps**

1. Create a new node with the given data.
2. Set the next pointer of the new node to the current head.
3. Update the head pointer to point to the new node.

**Code**

|  |
| --- |
| public void insertAtStart(int data) {  Node newNode = new Node(data); // Step 1: Create a new node  newNode.next = head; // Step 2: Point the new node to the current head  head = newNode; // Step 3: Update the head to the new node  } |

**Example**

**Before**: 20 -> 30 -> null  
**Insert 10**:  
**After**: 10 -> 20 -> 30 -> null

**2. Insert at the End (Tail)**

**Steps**

1. Create a new node with the given data.
2. If the list is empty (head is null), set the head to the new node.
3. Otherwise:
   * Traverse the list to the last node.
   * Set the next pointer of the last node to the new node.

**Code**

|  |
| --- |
| public void insertAtEnd(int data) {  Node newNode = new Node(data); // Step 1: Create a new node  if (head == null) { // Step 2: Check if the list is empty  head = newNode;  return;  }  Node temp = head;  while (temp.next != null) { // Step 3: Traverse to the last node  temp = temp.next;  }  temp.next = newNode; // Step 4: Point the last node to the new node  } |

**Example**

**Before**: 10 -> 20 -> null  
**Insert 30**:  
**After**: 10 -> 20 -> 30 -> null

**3. Insert at a Specific Position**

**Steps**

1. Create a new node with the given data.
2. If the position is 1 (insert at the head), call insertAtStart().
3. Otherwise:
   * Traverse the list to the (position - 1)-th node.
   * Set the next pointer of the new node to the next pointer of the current node.
   * Update the next pointer of the current node to point to the new node.

**Code**

|  |
| --- |
| public void insertAtPosition(int data, int position) {  Node newNode = new Node(data); // Step 1: Create a new node  if (position == 1) { // Step 2: If position is 1, insert at the start  newNode.next = head;  head = newNode;  return;  }  Node temp = head;  int count = 1;  while (count < position - 1 && temp != null) { // Step 3: Traverse to (position - 1)  temp = temp.next;  count++;  }  if (temp == null) { // Step 4: Check if position is valid  System.out.println("Position out of range!");  return;  }  newNode.next = temp.next; // Step 5: Update pointers  temp.next = newNode;  } |

**Example**

**Before**: 10 -> 30 -> 40 -> null  
**Insert 20 at position 2**:  
**After**: 10 -> 20 -> 30 -> 40 -> null

**Complete Java Code**

|  |
| --- |
| class SinglyLinkedList {  class Node {  int data;  Node next;  Node(int data) {  this.data = data;  this.next = null;  }  }  private Node head;  // Insert at the start  public void insertAtStart(int data) {  Node newNode = new Node(data);  newNode.next = head;  head = newNode;  }  // Insert at the end  public void insertAtEnd(int data) {  Node newNode = new Node(data);  if (head == null) {  head = newNode;  return;  }  Node temp = head;  while (temp.next != null) {  temp = temp.next;  }  temp.next = newNode;  }  // Insert at a specific position  public void insertAtPosition(int data, int position) {  Node newNode = new Node(data);  if (position == 1) {  newNode.next = head;  head = newNode;  return;  }  Node temp = head;  int count = 1;  while (count < position - 1 && temp != null) {  temp = temp.next;  count++;  }  if (temp == null) {  System.out.println("Position out of range!");  return;  }  newNode.next = temp.next;  temp.next = newNode;  }  // Print the linked list  public void printList() {  Node temp = head;  while (temp != null) {  System.out.print(temp.data + " -> ");  temp = temp.next;  }  System.out.println("null");  }  public static void main(String[] args) {  SinglyLinkedList list = new SinglyLinkedList();  list.insertAtStart(30); // Insert at start  list.insertAtStart(20);  list.insertAtStart(10);  list.printList(); // Output: 10 -> 20 -> 30 -> null  list.insertAtEnd(40); // Insert at end  list.insertAtEnd(50);  list.printList(); // Output: 10 -> 20 -> 30 -> 40 -> 50 -> null  list.insertAtPosition(25, 3); // Insert at position 3  list.printList(); // Output: 10 -> 20 -> 25 -> 30 -> 40 -> 50 -> null  }  } |

**Key Points to Remember**

1. **Edge Cases**:
   * Inserting at the head (position 1).
   * Inserting at an invalid position (greater than the size of the list).
   * Inserting into an empty list.
2. **Traversing Nodes**:
   * Always check if the node is null to avoid NullPointerException.
3. **Time Complexity**:
   * Insert at Start: **O(1)**
   * Insert at End: **O(n)** (requires traversal)
   * Insert at Specific Position: **O(n)** (requires traversal to position)

With this detailed explanation and code examples, you can easily insert nodes in a singly linked list.

**Deleting a Node in a Singly Linked List**

In a **Singly Linked List**, you can delete a node based on three main scenarios:

1. **Delete the Head Node (First Node)**
2. **Delete the Last Node (Tail)**
3. **Delete a Node at a Specific Position or with a Specific Value**

Let's explore these cases in detail with step-by-step explanations and Java code examples.

**1. Delete the Head Node (First Node)**

**Steps**

1. Check if the list is empty (head is null). If yes, print an appropriate message and return.
2. Update the head pointer to point to the next node.
3. The original head node is removed, and garbage collection will clean it up.

**Code**

|  |
| --- |
| public void deleteAtStart() {  if (head == null) { // Step 1: Check if the list is empty  System.out.println("List is empty. Nothing to delete.");  return;  }  head = head.next; // Step 2: Move the head pointer to the next node  } |

**Example**

**Before**: 10 -> 20 -> 30 -> null  
**After Deleting Head**: 20 -> 30 -> null

**2. Delete the Last Node (Tail)**

**Steps**

1. Check if the list is empty. If yes, print an appropriate message and return.
2. If the list has only one node:
   * Set the head to null.
3. Otherwise:
   * Traverse the list until the second-to-last node.
   * Set the next pointer of the second-to-last node to null.

**Code**

|  |
| --- |
| public void deleteAtEnd() {  if (head == null) { // Step 1: Check if the list is empty  System.out.println("List is empty. Nothing to delete.");  return;  }  if (head.next == null) { // Step 2: If there's only one node  head = null;  return;  }  Node temp = head;  while (temp.next.next != null) { // Step 3: Traverse to the second-to-last node  temp = temp.next;  }  temp.next = null; // Step 4: Remove the last node  } |

**Example**

**Before**: 10 -> 20 -> 30 -> null  
**After Deleting Tail**: 10 -> 20 -> null

**3. Delete a Node at a Specific Position**

**Steps**

1. Check if the list is empty. If yes, print an appropriate message and return.
2. If the position is 1, call deleteAtStart().
3. Otherwise:
   * Traverse the list to the (position - 1)-th node.
   * Update the next pointer of the (position - 1)-th node to skip the target node and point to the node after it.
4. If the position is invalid (beyond the list size), print an appropriate message.

**Code**

|  |
| --- |
| public void deleteAtPosition(int position) {  if (head == null) { // Step 1: Check if the list is empty  System.out.println("List is empty. Nothing to delete.");  return;  }  if (position == 1) { // Step 2: If position is 1, delete the head  head = head.next;  return;  }  Node temp = head;  int count = 1;  while (count < position - 1 && temp != null) { // Step 3: Traverse to (position - 1)  temp = temp.next;  count++;  }  if (temp == null || temp.next == null) { // Step 4: Check if position is valid  System.out.println("Position out of range!");  return;  }  temp.next = temp.next.next; // Step 5: Update the next pointer to skip the target node  } |

**Example**

**Before**: 10 -> 20 -> 30 -> 40 -> null  
**Delete Node at Position 3**:  
**After**: 10 -> 20 -> 40 -> null

**4. Delete a Node with a Specific Value**

**Steps**

1. Check if the list is empty. If yes, print an appropriate message and return.
2. If the head node contains the target value, call deleteAtStart().
3. Otherwise:
   * Traverse the list to find the target node while keeping track of the previous node.
   * Update the next pointer of the previous node to skip the target node.

**Code**

|  |
| --- |
| public void deleteByValue(int value) {  if (head == null) { // Step 1: Check if the list is empty  System.out.println("List is empty. Nothing to delete.");  return;  }  if (head.data == value) { // Step 2: If the head node contains the target value  head = head.next;  return;  }  Node temp = head;  while (temp.next != null && temp.next.data != value) { // Step 3: Find the target node  temp = temp.next;  }  if (temp.next == null) { // Step 4: Check if the value was not found  System.out.println("Value not found in the list.");  return;  }  temp.next = temp.next.next; // Step 5: Remove the target node  } |

**Example**

**Before**: 10 -> 20 -> 30 -> 40 -> null  
**Delete Node with Value 30**:  
**After**: 10 -> 20 -> 40 -> null

**Complete Java Code**

|  |
| --- |
| class SinglyLinkedList {  class Node {  int data;  Node next;  Node(int data) {  this.data = data;  this.next = null;  }  }  private Node head;  public void deleteAtStart() {  if (head == null) {  System.out.println("List is empty. Nothing to delete.");  return;  }  head = head.next;  }  public void deleteAtEnd() {  if (head == null) {  System.out.println("List is empty. Nothing to delete.");  return;  }  if (head.next == null) {  head = null;  return;  }  Node temp = head;  while (temp.next.next != null) {  temp = temp.next;  }  temp.next = null;  }  public void deleteAtPosition(int position) {  if (head == null) {  System.out.println("List is empty. Nothing to delete.");  return;  }  if (position == 1) {  head = head.next;  return;  }  Node temp = head;  int count = 1;  while (count < position - 1 && temp != null) {  temp = temp.next;  count++;  }  if (temp == null || temp.next == null) {  System.out.println("Position out of range!");  return;  }  temp.next = temp.next.next;  }  public void deleteByValue(int value) {  if (head == null) {  System.out.println("List is empty. Nothing to delete.");  return;  }  if (head.data == value) {  head = head.next;  return;  }  Node temp = head;  while (temp.next != null && temp.next.data != value) {  temp = temp.next;  }  if (temp.next == null) {  System.out.println("Value not found in the list.");  return;  }  temp.next = temp.next.next;  }  public void printList() {  Node temp = head;  while (temp != null) {  System.out.print(temp.data + " -> ");  temp = temp.next;  }  System.out.println("null");  }  public static void main(String[] args) {  SinglyLinkedList list = new SinglyLinkedList();  list.head = list.new Node(10);  list.head.next = list.new Node(20);  list.head.next.next = list.new Node(30);  list.head.next.next.next = list.new Node(40);  list.printList(); // Output: 10 -> 20 -> 30 -> 40 -> null  list.deleteAtStart();  list.printList(); // Output: 20 -> 30 -> 40 -> null  list.deleteAtEnd();  list.printList(); // Output: 20 -> 30 -> null  list.deleteAtPosition(2);  list.printList(); // Output: 20 -> null  list.deleteByValue(20);  list.printList(); // Output: null  }  } |

**Key Points**

* **Edge Cases**:
  + Deleting from an empty list.
  + Deleting the only node in the list.
  + Invalid position or value not found.
* **Time Complexity**:
  + Delete Head: **O(1)**
  + Delete Tail or Specific Node: **O(n)** (requires traversal)
* **Space Complexity**: **O(1)** (no additional space required)

Top of Form

Bottom of Form