CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Linear Data Structures

Department of Computer Science
University of Maryland, College Park

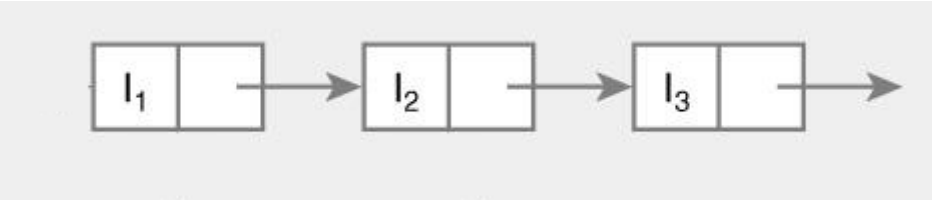
Information

- Linked lists summary
  - http://www.cs.umd.edu/~nelson/classes/resources/LinkedLists/

List Implementations

- Two basic implementation techniques for lists
  - Store elements in an array
  - Store as a linked list
    - Place each element in a separate object (node)
    - Node contains reference to other node(s)
    - Link nodes together

```
Class Node {
    Object data;
    Node next;
}
```



    - Node head → points to first node
- **Example:** Class definitions in **MyLinkedList.java** code distribution

Array vs. LinkedList Implementations

- **Array**
  - **Advantages**
    - Can efficiently access element at any position (O(1))
    - Efficient use of space (space just to hold reference to each element)
  - **Disadvantages**
    - Expensive to grow / shrink array
      - Can amortize cost (grow / shrink in spurts)
    - Expensive to insert / remove elements in middle (O(n))
    - Tricky to insert / remove elements at both ends

Array vs. LinkedList Implementations

- **LinkedList**
  - **Advantages**
    - Can efficiently insert / remove elements anywhere
  - **Disadvantages**
    - Cannot efficiently access element at any position
      - Need to traverse list to find element (O(n))
    - Less efficient use of space
      - 1-2 additional references per element

Implementing Linked List Code

- Drawing diagrams can help you to understand the steps needed to complete a task and the order of the steps (order makes a difference for some linked list tasks)
- Make sure that you check your code works for different scenarios
- **Typical Scenarios**
  - List is empty
  - List has only one element
  - Processing takes place at the beginning or end of the list (e.g., insertion)

Implementing Linked List Code

- Most tasks (insert, delete, find, etc.) you need to implement using a linked list can be implemented using one of the following approaches:
  - **Print traversal, Prev/Curr Traversal, Recursion**
- **Print traversal** - Loop that visits every element of the list. We like to call it the "print traversal" as it is the traversal you use when printing a list. If in the worst case (e.g., find) you need to visit every node of the list, without modifying it, then chances are this loop will take care of the task

```
Node curr = head;
while (curr != null) {
    /* Process data (e.g., curr.data) */

    curr = curr.next;  /* Move to next element */
}
```

- Updating the loop expression to **curr.next != null** will allow you to have a reference to the last node after the loop is over **if the list is not empty**

Implementing Linked List Code

- **Prev/Curr "Tom and Jerry" Traversal** - Loop that visits every element of the list using two references: **curr** (has the address of the current element been processed) and **prev** (points to the element that precedes the current one). If you need to modify the list (e.g., remove or insert an element) usually you need to update references of the node preceding the current one. In this loop **prev** will follow **curr** (Tom will follow Jerry ☺). **prev** and **curr** move in parallel

```
Node prev = null, curr = head;  /* Element preceding head is null */

while (curr != null) {
    /* Processing here */

    prev = curr;          /* Adjusting prev to current element */
    curr = curr.next;     /* Moving to next element */
 }
```

Implementing Linked List Code

- **Recursion** - Some problems can be easily solved using recursion (e.g., printing a list in reverse order). For problems involving recursion, you will use an auxiliary method to satisfy the recursive requirement. The following is an example of a recursive method that returns a string with the data elements. We are assuming you cannot add instance/static variable to support the recursion

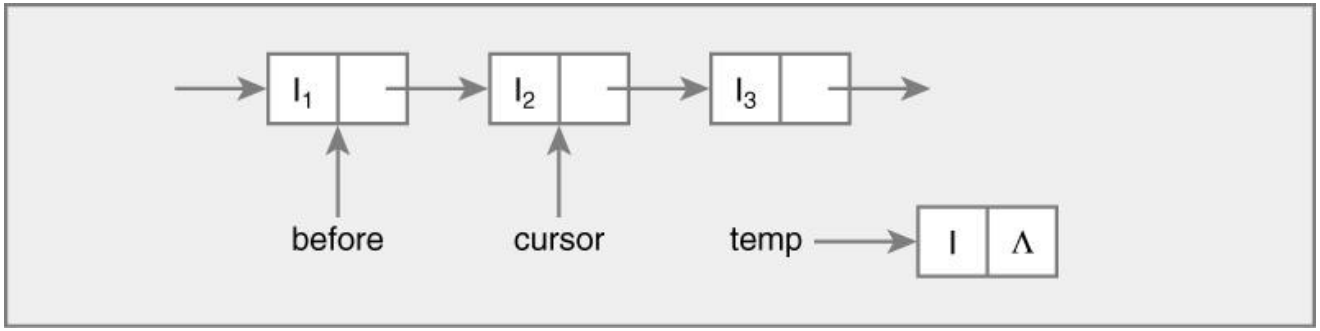
```
public String getListString() {
    return getListString(head);
}

 // Auxiliary method that satisfies the recursion requirement
private String getListString(Node headAux) {
    if (headAux == null) {
        return "";
    }
    return headAux.data + " " + getListString(headAux.next);
}
```

Linked List Code

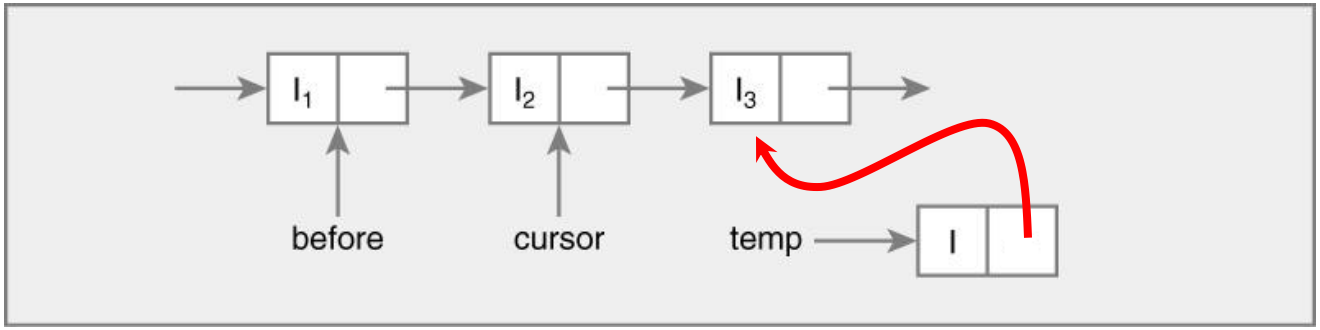
- Let's take a look at some operations implemented using linked list code. After this slide you will find diagrams similar to the ones we will draw for some of the methods we will discuss
- **Example:** MyLinkedList.java

Linked List – Insert (After Cursor)

1. Original list & new element temp

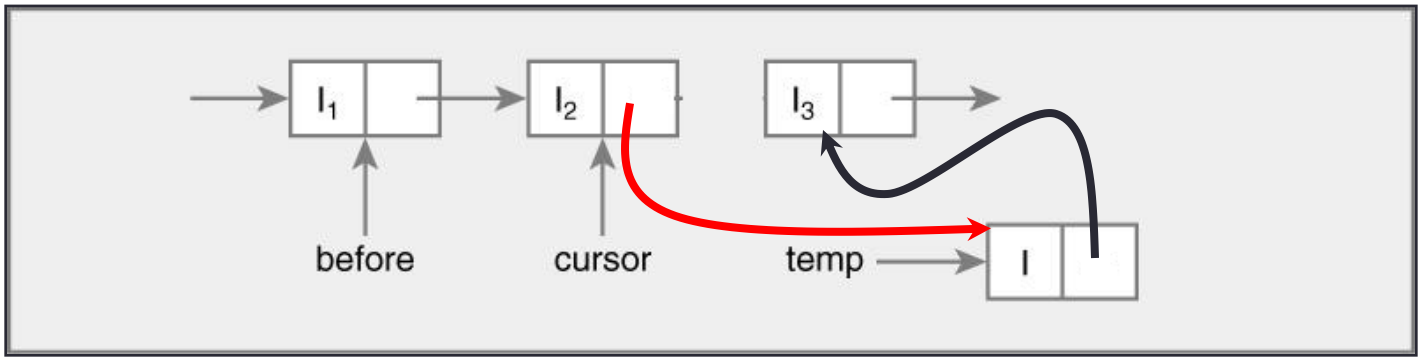


2. Modify temp.next → cursor.next

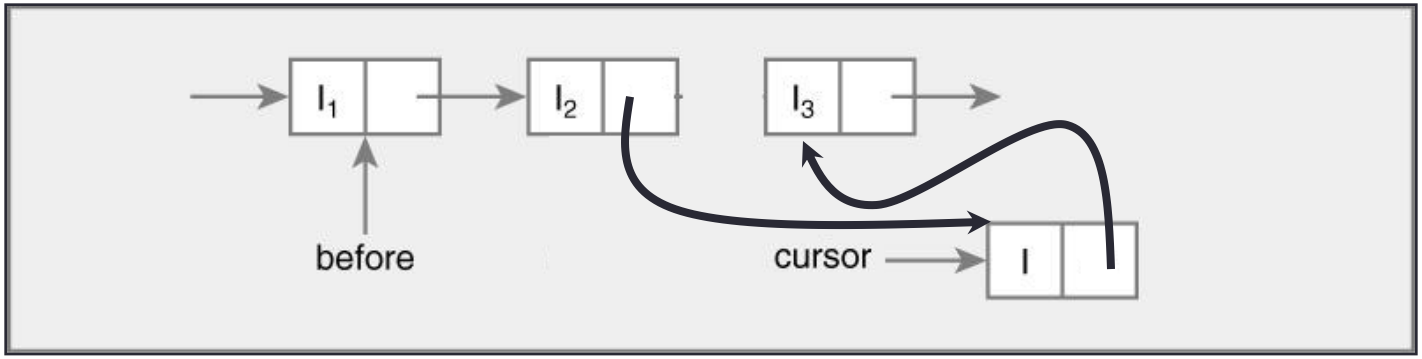


Linked List – Insert (After Cursor)

3. Modify cursor.next → temp

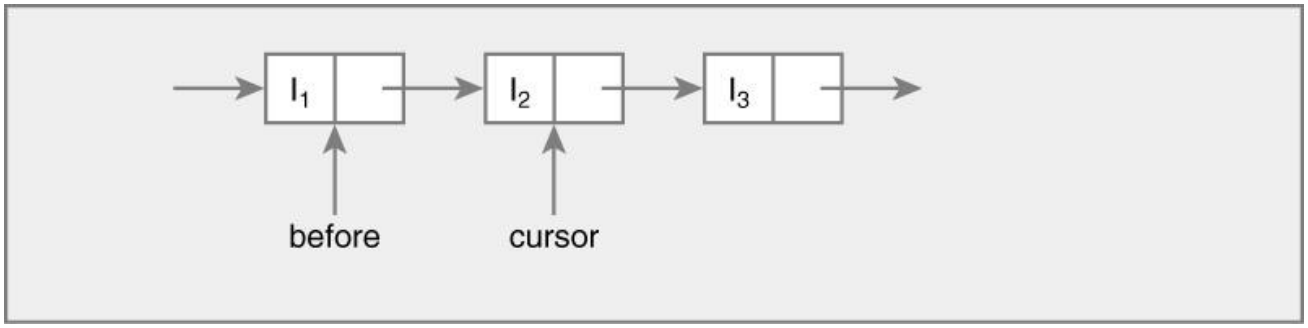


4. Modify cursor → temp

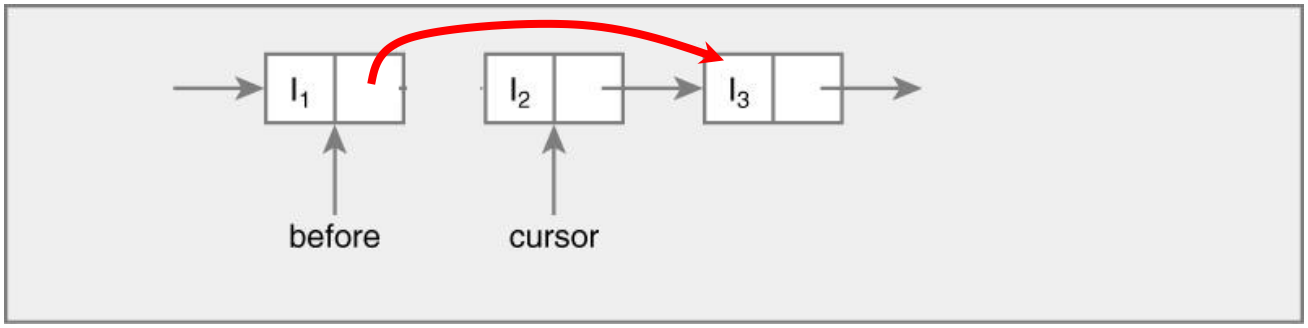


Linked List – Delete (Cursor)

1. Find before such that before.next = cursor

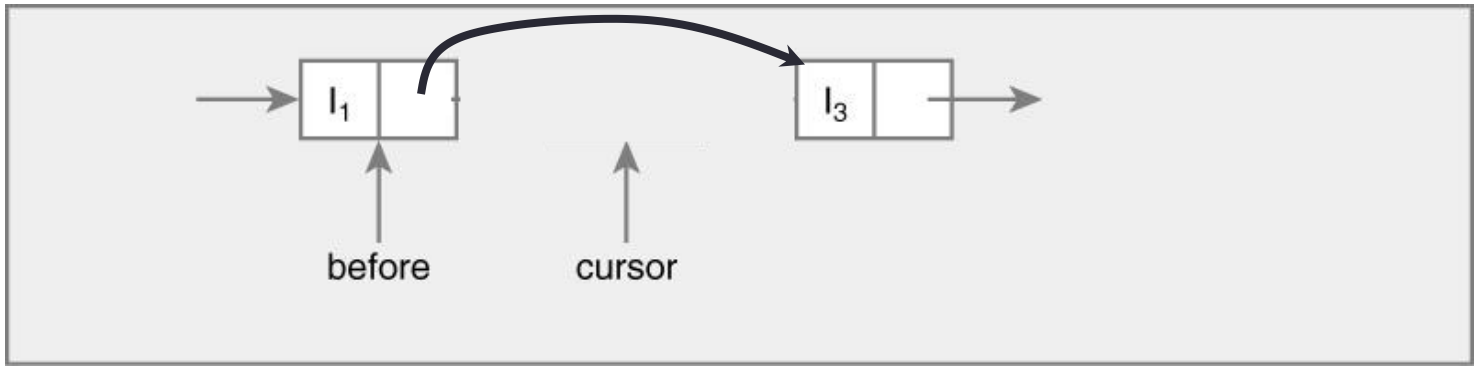


2. Modify before.next → cursor.next

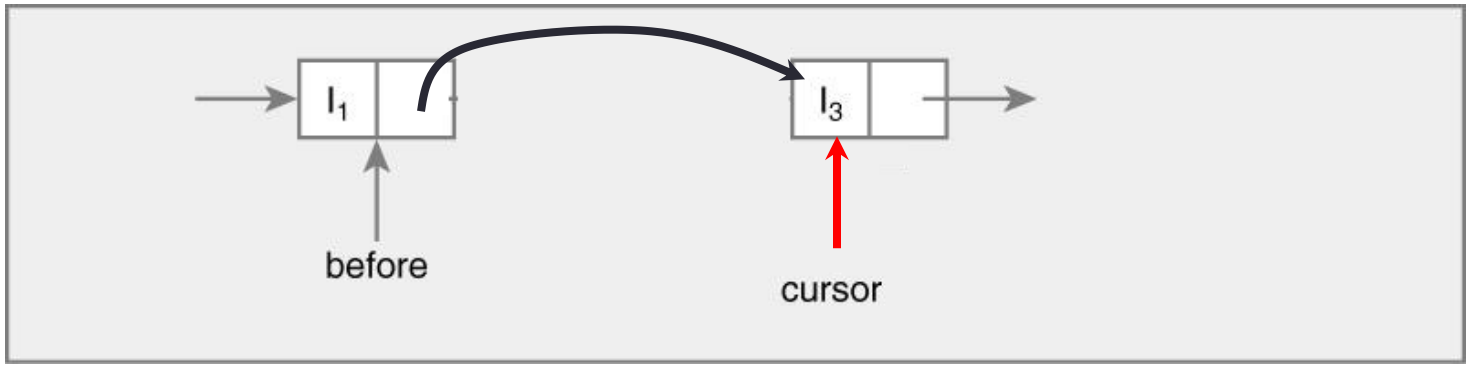


Linked List – Delete (Cursor)

3. Delete cursor



4. Modify cursor → before.next

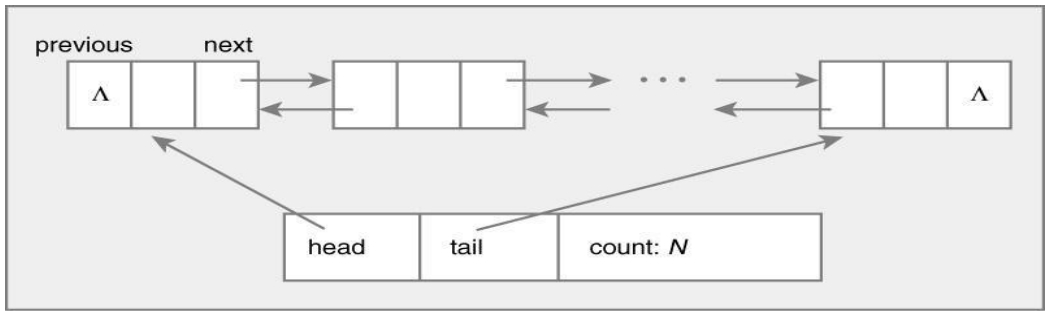


Maintaining List Sorted

- One approach to maintain a linked list sorted with every insertion is
  - **If the list is empty**
    - Just make the element the first of the list (insertion is trivial)
  - **Otherwise**
    - Traverse the list until you find an element (B) larger than the one you want to insert (A)
    - Once you find B, insert A before B
    - If you don’t find B, A will become the last element of the list
    - Let’s see an example

Doubly Linked List

- Linked list where element has predecessor & successor



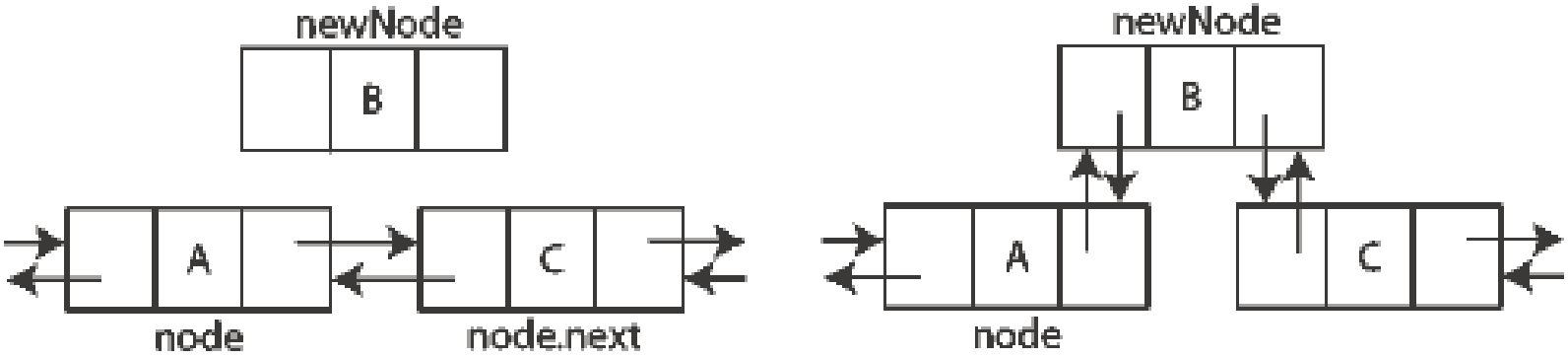
- **Structure**

```
Class Node {
    Object data;
    Node next;
    Node previous;
}
```

- **Issues**
  - Easy to find preceding / succeeding elements
  - Extra work to maintain links (for insert / delete)
  - More storage per node

Doubly Linked List – Insertion

- **Example**



- Must update references in both predecessor and successor nodes