

Linear Data Structures



Fawzi Emad
Chau-Wen Tseng

Department of Computer Science
University of Maryland, College Park

Linear Data Structures

- **Lists**
 - **Linked list**
 - **Doubly linked list**
 - **Circular list**
- **Stack**
- **Queue**
 - **Circular queue**

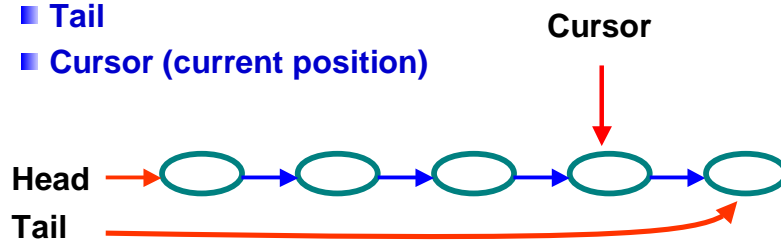
Linked List

■ Properties

- Elements in linked list are **ordered**
- Element has **successor**

■ State of List

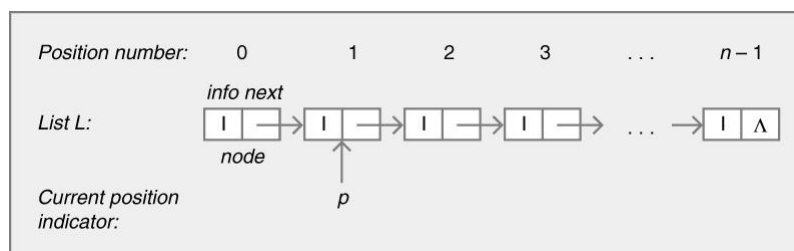
- **Head**
- **Tail**
- **Cursor (current position)**



Reference-based Implementation

■ Nodes contain references to other nodes

■ Example



■ Issues

- **Easy to find succeeding elements**
- **Start from head of list for preceding elements**

Array vs. Reference-based Linked List

■ Reference-based linked list

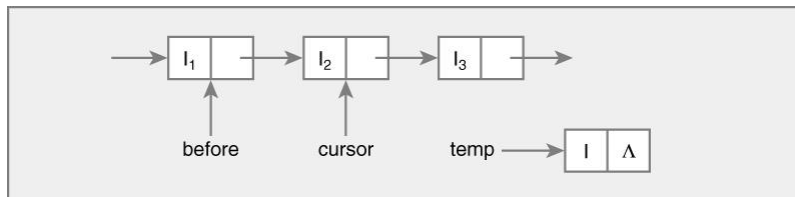
- Insertion / deletion = $O(1)$
- Indexing = $O(n)$
- Easy to dynamically increase size of list

■ Array

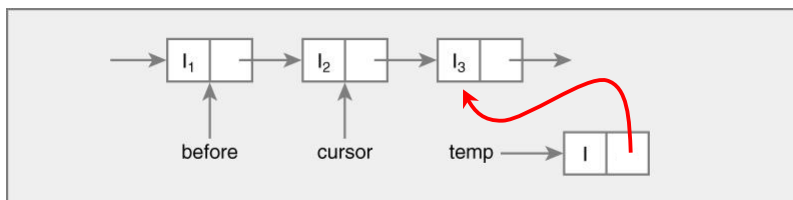
- Insertion / deletion = $O(n)$
- Indexing = $O(1)$
- Compact, uses less space
- Easy to copy, merge
- Better cache locality

Linked List – Insert (After Cursor)

1. Original list & new element **temp**

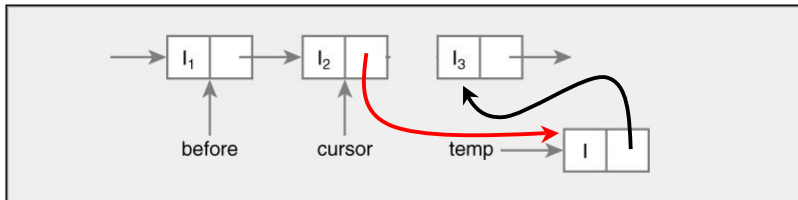


2. Modify **temp.next** → **cursor.next**

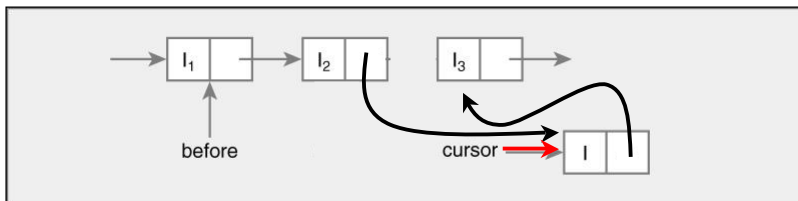


Linked List – Insert (After Cursor)

3. Modify **cursor.next** → **temp**

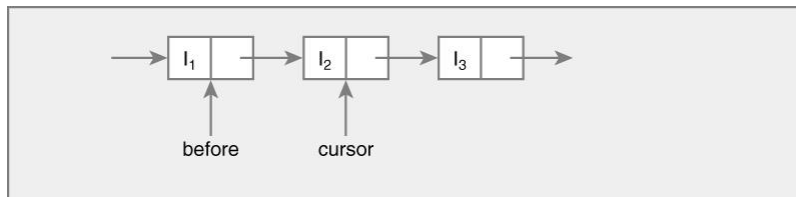


4. Modify **cursor** → **temp**

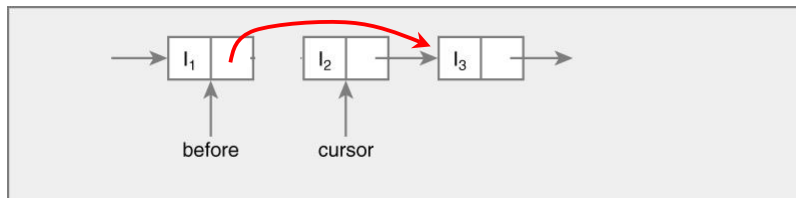


Linked List – Delete (Cursor)

1. Find **before** such that **before.next = cursor**

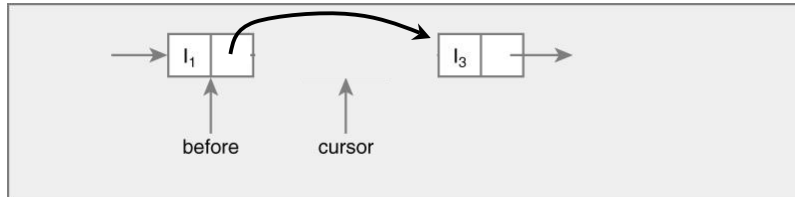


2. Modify **before.next** → **cursor.next**

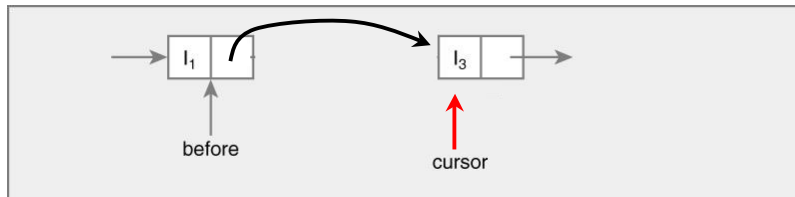


Linked List – Delete (Cursor)

3. Delete **cursor**



4. Modify **cursor** \rightarrow before.next



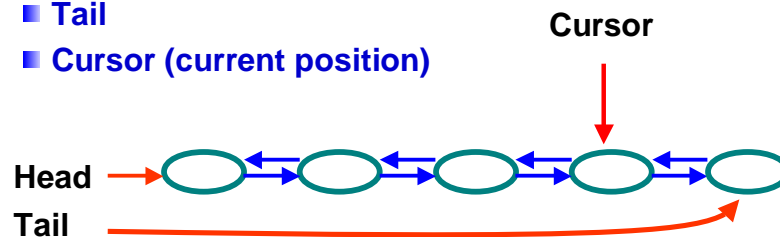
Doubly Linked List

■ Properties

- Elements in linked list are **ordered**
- Element has **predecessor & successor**

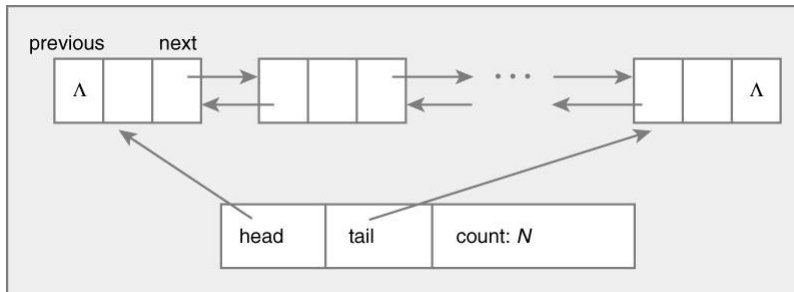
■ State of List

- Head
- Tail
- Cursor (current position)



Doubly Linked List

■ Example



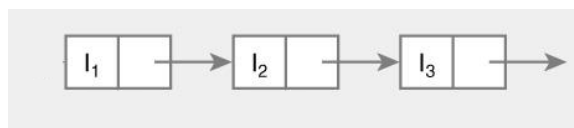
■ Issues

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

Node Structures for Linked Lists

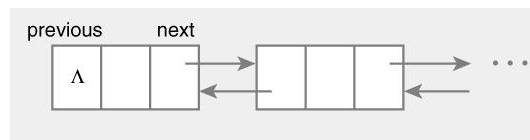
■ Linked list

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



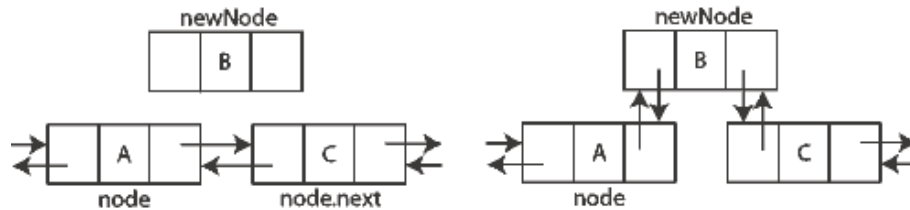
■ Doubly linked list

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



Doubly Linked List – Insertion

■ Example



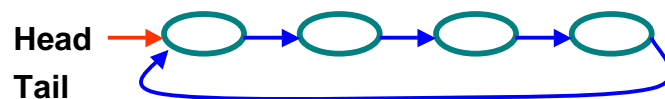
- Must update references in **both** predecessor and successor nodes

Circular Linked Lists

- Last element links to first element

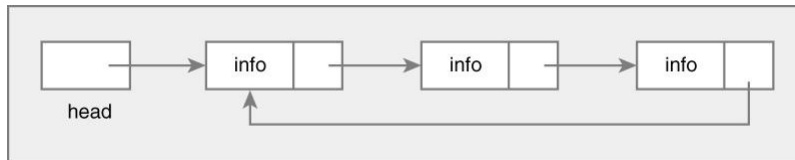
■ Properties

- Can reach entire list from any node
- Need special test for end of list
- Represent
 - Buffers
 - Naturally circular data

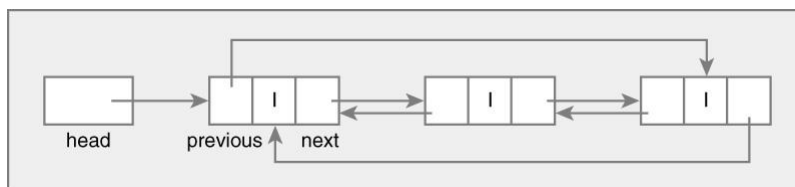


Circular Linked Lists – Examples

■ Circular linked list



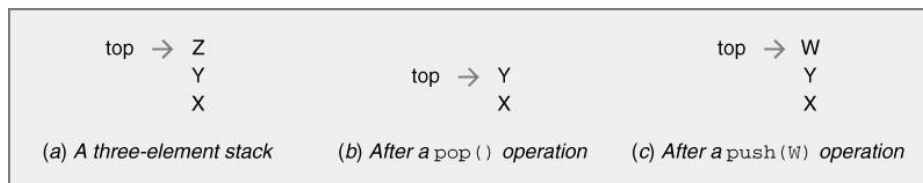
■ Circular doubly linked list



Stack

■ Properties

- Elements removed in **opposite** order of insertion
- Last-in, First-out (LIFO)
- Must track position of **Top** (last element added)



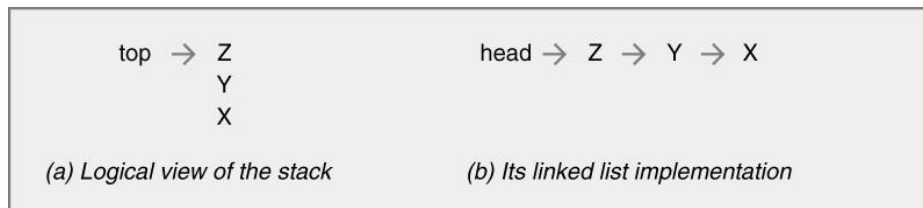
■ Stack operations

- Push = add element (to top)
- Pop = remove element (from top)

Stack Implementations

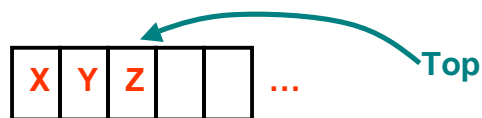
■ Linked list

■ Add / remove from head of list



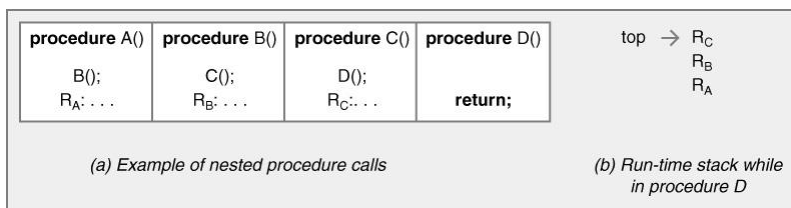
■ Array

■ Increment / decrement Top pointer after push / pop



Stack Applications

■ Run-time procedure information



■ Arithmetic computations

■ Postfix notation

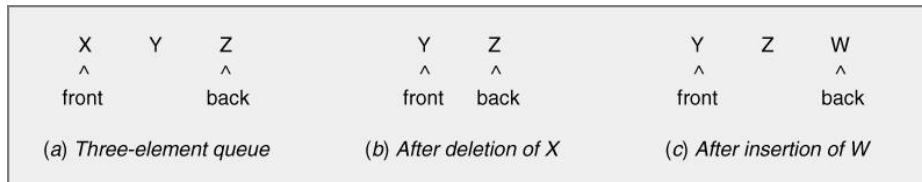
■ Simplified instruction set

■ Java bytecode

Queue

■ Properties

- Elements removed **in order** of insertion
- **First-in, First-out (FIFO)**
- Must track **Front** (first in) and **Back** (last in)



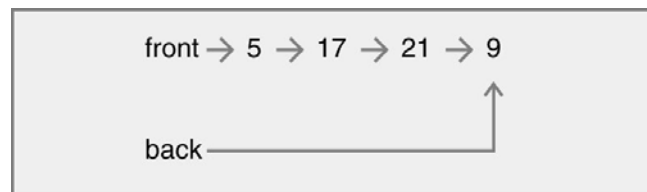
■ Queue operations

- **Enqueue** = add element (to back)
- **Dequeue** = remove element (from front)

Queue Implementations

■ Linked list

- Add to **tail** (Back) of list
- Remove from **head** (Front) of list



■ Array

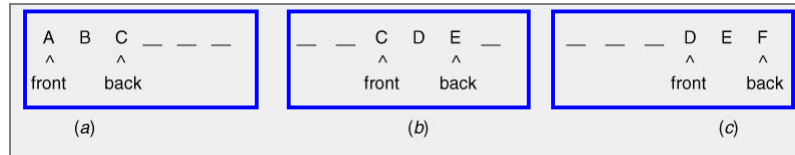
■ Circular array

Queue – Array

- Store queue as elements in array

- Problem

- Queue contents move (“inchworm effect”)



- As result, can not add to back of queue, even though queue is not full

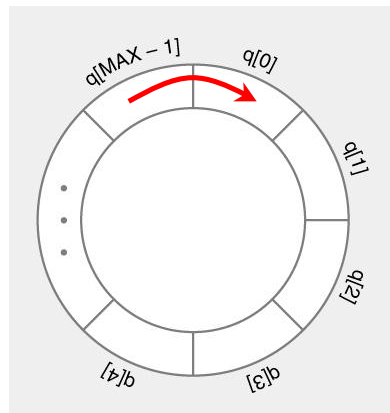
Queue – Circular Array

- Circular array (ring)

- $q[0]$ follows $q[\text{MAX} - 1]$
 - Index using $q[i \% \text{MAX}]$

- Problem

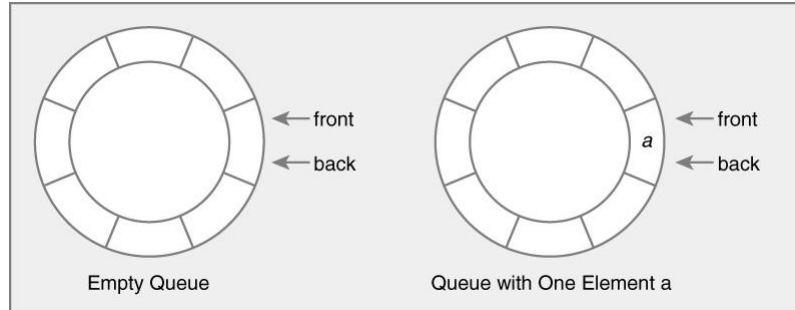
- Detecting difference between empty and nonempty queue



Queue – Circular Array

■ Approach 1

- Keep **Front** at first in
- Keep **Back** at last in



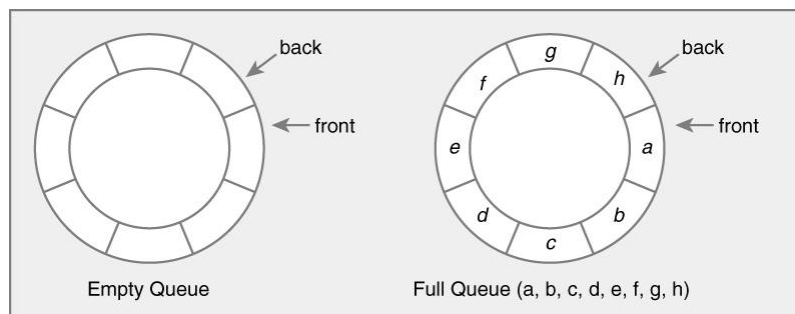
■ Problem

- Empty queue identical to queue with **1** element

Queue – Circular Array

■ Approach 2

- Keep **Front** at first in
- Keep **Back** at last in – 1



■ Problem

- Empty queue identical to **full** queue

Queue – Circular Array

- Inherent problem for queue of size **N**
 - Only **N** possible (Front – Back) pointer locations
 - **N+1** possible queue configurations
 - Queue with 0, 1, ... **N** elements
- Solutions
 - Maintain additional state information
 - Use state to recognize empty / full queue
 - Examples
 - Record **Size**
 - Record **QueueEmpty** flag
 - Leave empty element in queue
 - Store marker in queue