# CSC207.01 2013F, Class 29: Doubly-Linked Lists

*Overview*

- Admin.
- Review of singly-linked lists.
- Insertion and deletion in singly-linked lists.
- A trick for simplifying insertion and deletion.
- Doubly-linked lists.
- Circularly-linked lists.

*Admin*

- Assignment 7 remains "Implement doubly linked lists and then add Ushahidi incidents."
- We will continue the work a little / talk a little approach.
- Upcoming extra credit opportunities
    - CS Table: Cool readings on building software
    - Others?
- Have a good break!

## Review of singly-linked lists

- Basic idea: Combine *nodes* that have
- We're going to wrap these nodes into something that implements our list interface
    - Fields in LinkedList
    - How to prepend and append
    - How to build an iterator

## Insertion and deletion in singly-linked lists

- Prepend and APpend are our first examples
- Question:
    - What fields? You decide.
    - What do nodes look like class Node { T val; Node next; public Node(val, next) { this.val = val; this.next = next; } } // class Node
    - What are the method signatures? void prepend(T val) throws Exception; void append(T val) throws Exception;
- Useful fields
    - front, the front of the list

Prepend

```
public void prepend(T val) {
    // Create a new node, with T as a val and front as next
   Node<T> newfront = new Node(T, this.front);
   this.front = newfront;
} // prepend(T)

public void prepend(T val) {
    // Create a new node, with T as a val and front as next
   this.front = new Node(T, this.front);
} // prepend(T)
```

Two strategies:

- Keep a field for the end of the list
- Iterate through to the end of the list
- A field is the winner

Code, revisited

```
Node<T> front;
Node<T> back;

public void prepend(T val) {
    // Create a new node, with T as a val and front as next
   this.front = new Node(T, this.front);
} // prepend(T)

public void append(T val) {
    // Create a new node with val and no successor
    // Update the former back of the list to make this new node the next node
    this.back.next =  new Node(T, null);
    // Update our notion of the back of the list
    this.back = this.back.next;
} // append(T)
```

What do we initialize front and back to for an empty list?

- null!

  public void prepend(T val) { // Create a new node, with T as a val and front as next this.front = new Node(T, this.front); // Special case: Empty list if (this.back == null) this.back = this.front; } // prepend(T)

  public void append(T val) { // Deal with empty list if (this.back == null) { this.back = new Node(T, null); this.front = this.back; } else { // Create a new node with val and no successor // Update the former back of the list to make this new node the next node this.back.next = new Node(T, null); // Update our notion of the back of the list this.back = this.back.next; } } // append(T)

Worrying about special cases in linked lists is really important! (Was it empty? for insert; Will it be empty? for delete)

## A trick for simplifying insertion and deletion

On to insert and delete with a cursor

```
public class LinkedListCursor<T> implements Cursor {

} // LinkedListCursor<T>
```

Add after current element is easy, delete current element is hard

- So have the cursor point to the previous element

Deletion

```
cursor.prev.next = cursor.prev.next.next;
```

A problem: How do we delete the first element, and where is cursor.prev when we start?

## Doubly-linked lists

Twice the links! Twice the fun! Twice the potential for errors.

## Circularly-linked lists

Dummy node with link to front and back