Lecture 11
More Linked Lists

CS 62, Spring 2016
Pomona College
February 12, 2016

Yi-Chieh (Jessica) Wu

Learning Goals

- Be able to describe the differences between linked list variants
  - Singly linked list
  - Singly linked list with tail reference
  - Circular linked list
  - Doubly linked list
- Be able to write list methods for any variant!

Linked Lists: an implementation of List

- Composed of Nodes

```
public class Node<E> {
        protected E data; // value stored in this element
        protected Node<E> next; // ref to next
        ...
}
```

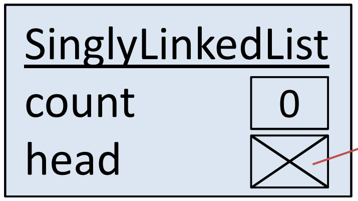
nodes is a self-referential data structure:
the `Node` object has a reference to a `Node`

- Keep track of size and head

```
public class SinglyLinkedList<E> {
        protected int count; // list size
        protected Node<E> head; // ref to first element
        ...
}
```

Visualizing Linked Lists

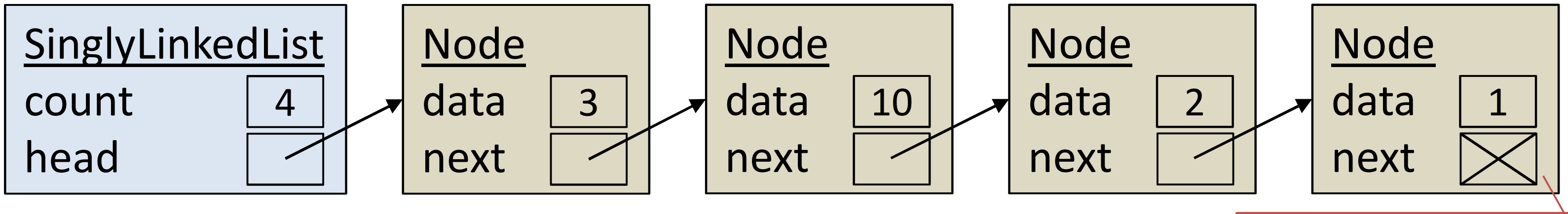
- an empty singly linked list



SinglyLinkedList
count 0
head

for an empty list, head contains a `null` reference

- a non-empty singly linked list



SinglyLinkedList
count 4
head

Node
data 3
next

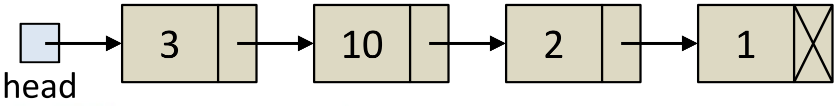
Node
data 10
next

Node
data 2
next

Node
data 1
next

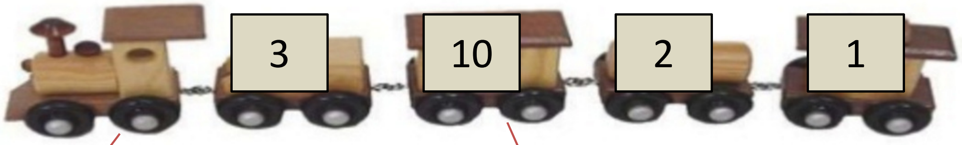
for a non-empty list, the final node has a `null`-valued `next` reference

(a "simplified" visualization)



3
10
2
1
head

real world analogy:
a train!



the locomotive is a `List` object and has a reference (`head`) to the first car

each car is a `Node` object, with cargo (`data`) and a link (`next`) to the next node

Exercise

Q

You need to fill in this slide!

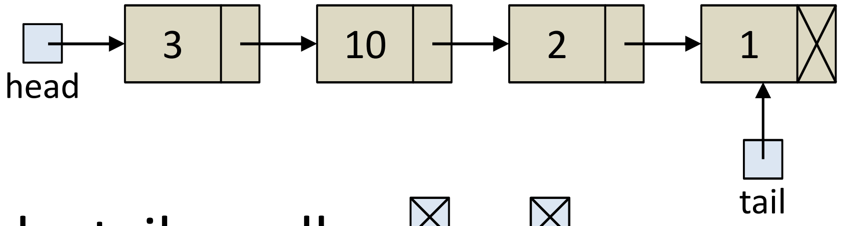
What is the worst-case time complexity of each?

	linked list
`addFirst(E value)`	
`getFirst()`	
`removeFirst()`	
`addLast(E value)`	
`getLast()`	
`removeLast()`	
`remove(E value)`	
`contains(E value)`	

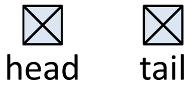
“Tail” Reference: Basics

```
public class SinglyLinkedList<E> {
        protected int count; // list size
        protected Node<E> head; // ref to first element
        protected Node<E> tail; // ref to last element
        ...
}
```

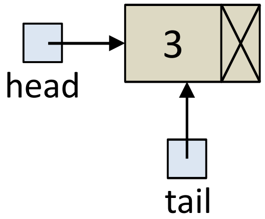
- general case: head != tail



- empty list: head = tail = null

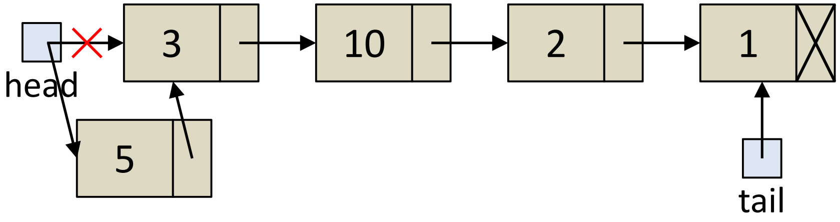


- one-element list: head = tail = reference to node



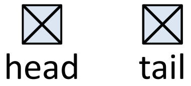
“Tail” Reference: Trade-offs

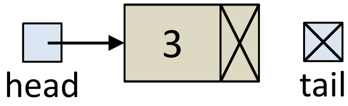
- What does this provide?
  - `getLast(), addLast(E value)` were `O(n)`, now `O(1)`
- What is the cost?
  - Adds complexity to add and remove methods
    - We have to worry about updating tail
    - Example: `addFirst(E value)`



```
public void addFirst(E value)
// post: value is added to beginning of list
{
    // note order that things happen:
    // head is parameter, then assigned
    head = new Node<E>(value, head);
    count++;
}
```

What happens if we addFirst(E value) to an empty list?



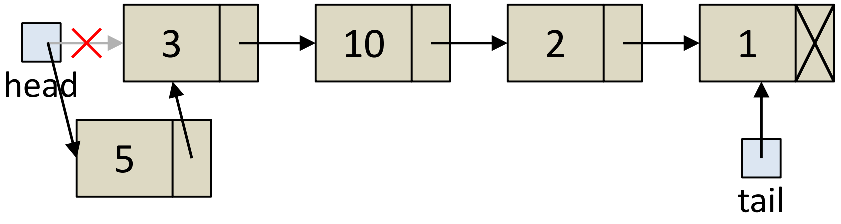


tail is still null! ☹

This is the `addFirst(E value)` method from `SinglyLinkedList`. It does NOT work for lists with tail references.

“Tail” Reference: Trade-offs

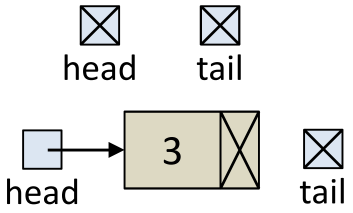
- What does this provide?
  - `getLast(), addLast(E value)` were `O(n)`, now `O(1)`
- What is the cost?
  - Adds complexity to add and remove methods
    - We have to worry about updating tail
    - Example: `addFirst(E value)`



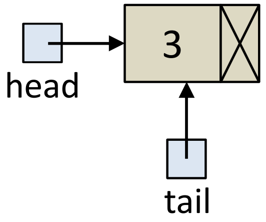
```
public void addFirst(E value)
// post: value is added to beginning of list
{
    // note order that things happen:
    // head is parameter, then assigned
    head = new Node<E>(value, head);
    if (tail == null) // first value added
        tail = head;
    count++;
}
```

New code needed to update the tail reference!

What happens if we addFirst(E value) to an empty list?



tail is still null! ☹

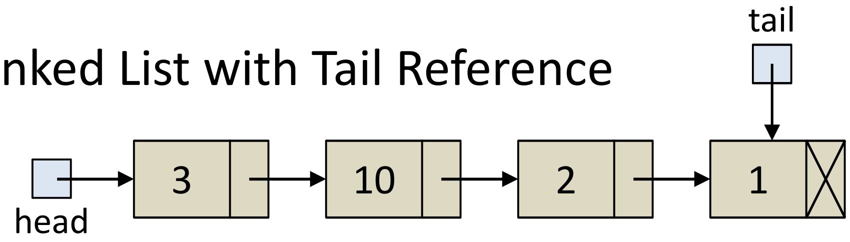


fixed! ☺

From Tail Reference to …

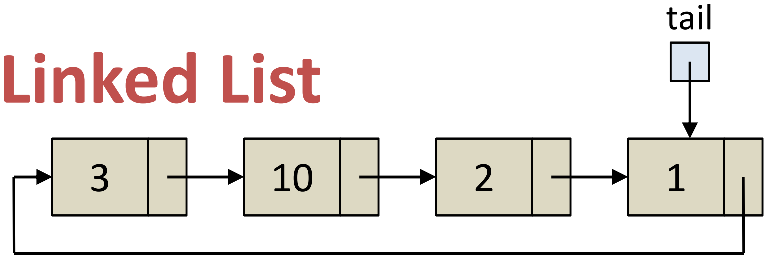
- Tail node has “wasted” `next` field (always `null`)

Singly Linked List with Tail Reference



- Why not use this field to point to beginning of list?
  - Then we do not need a head field!
  - `head` is always found as `tail.next()`

Circular Linked List

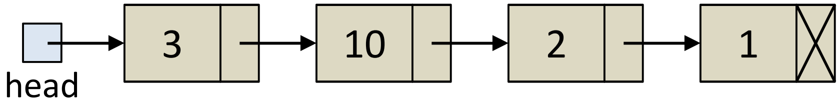


real world analogy:
a necklace!

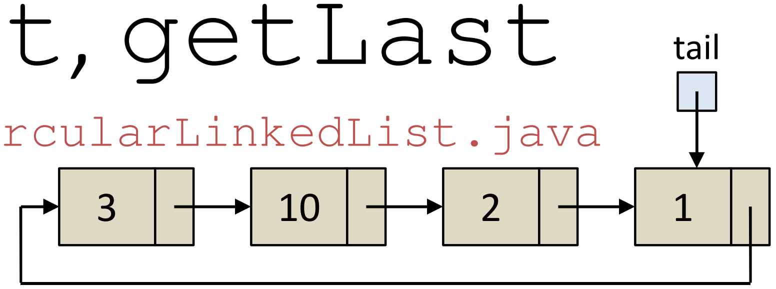
necklace = List
bead = Node
links = references to next bead

Circular LLs: getFirst, getLast

SinglyLinkedList.java



CircularLinkedList.java



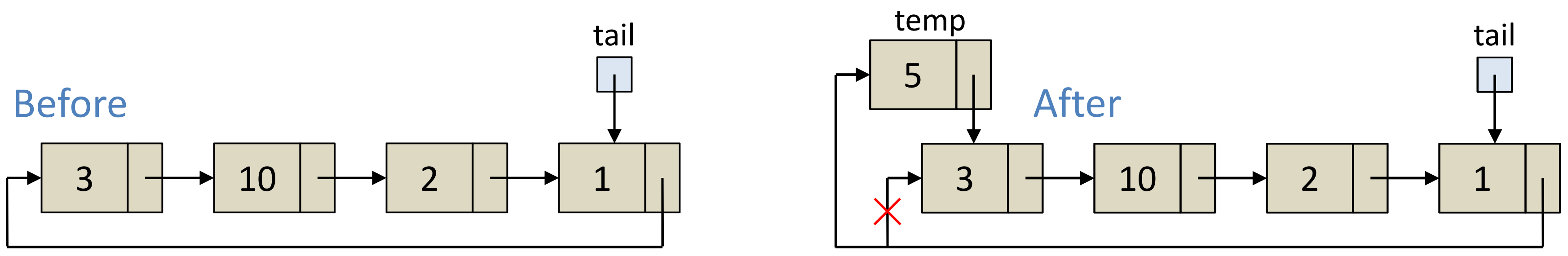
```
public E getFirst()
{
    return ~~head~~.value();
}         tail.next()
```

remember, head = tail.next()

```
public E getLast()
{
    ~~Node<E> finger = head;~~
    ~~Assert.condition(finger != null,~~
                     ~~"List is not empty.");~~
    ~~while (finger != null &&~~
           ~~finger.next() != null)~~
    ~~{~~
        ~~finger = finger.next();~~
    ~~}~~
    return ~~finger.value();~~  tail.value();
}
```

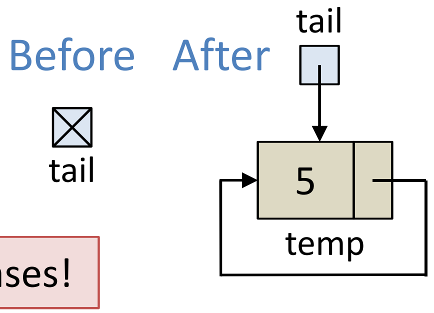
no need to use finger-based traversal to get to last element in list anymore

Circular LLs: addFirst



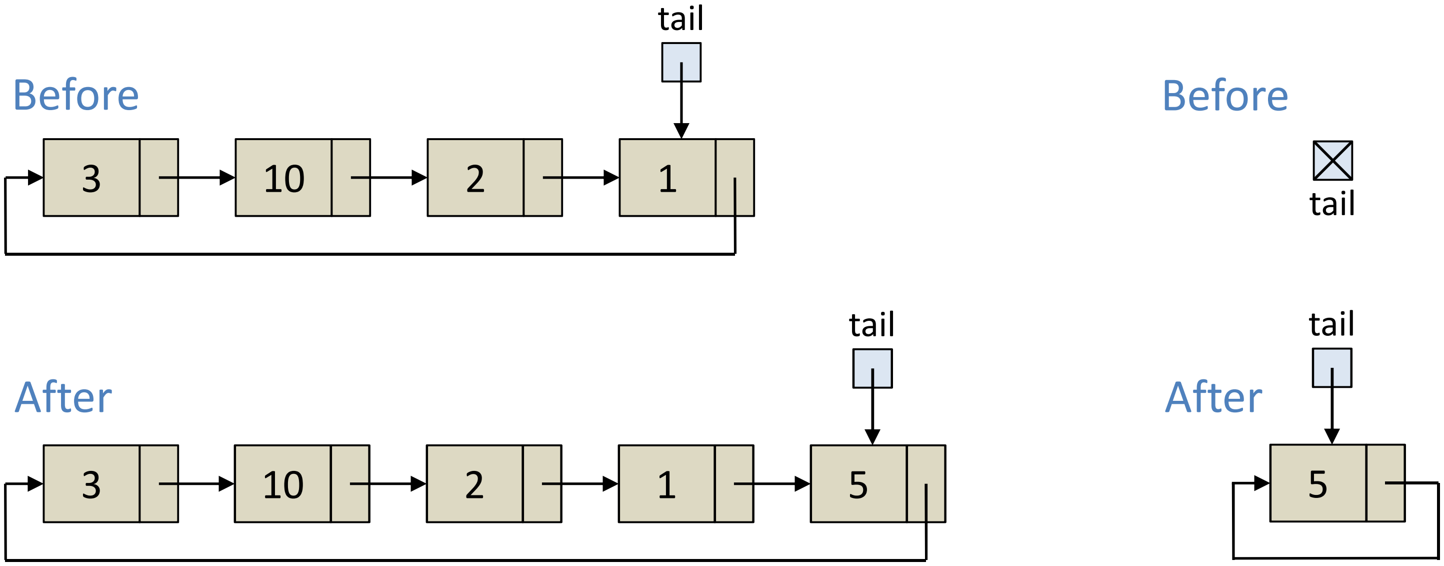
```
public void addFirst(E value)
{
    Node<E> temp = new Node<E>(value);
    if (tail == null) { // first value added
        tail = temp;
        tail.setNext(tail);
    } else { // element exists in list
        temp.setNext(tail.next());
        tail.setNext(temp);
    }
    count++;
}
```

Always consider boundary cases!

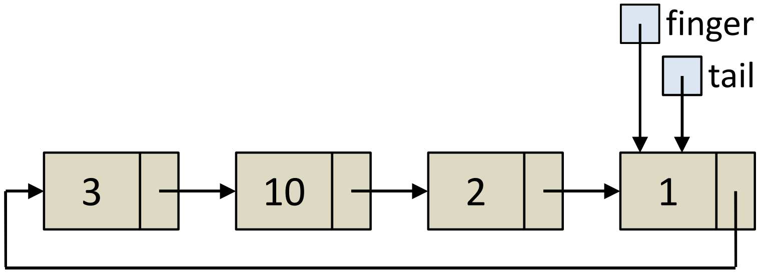


Circular LLs: `addLast`

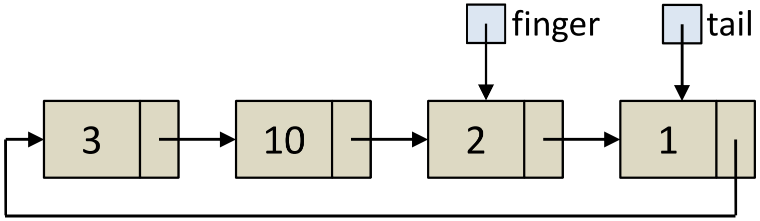
- Exercise: Implement `addLast(E value)`



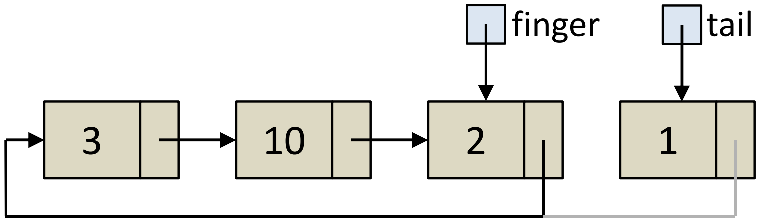
Circular LLs: `removeLast`



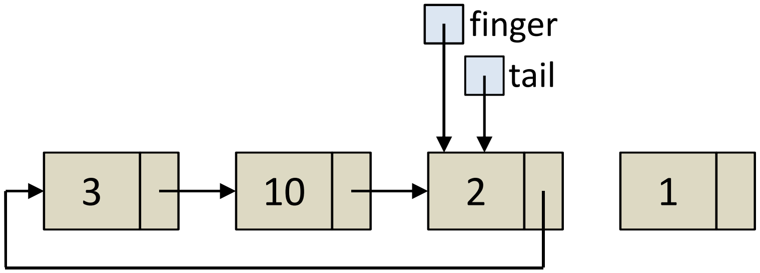
Set `finger` to `tail` (our only way to “enter” the list).



Loop `finger` until it points to predecessor of `tail` (i.e. while `finger.next() != tail`).



Update references using `finger.setNext(tail.next()).`



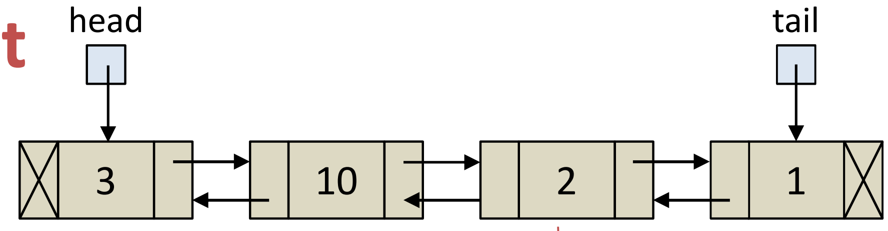
Update `tail = finger.`

Bonus: This does not handle the case in which the list starts with a single element. What would you have to add?

From Circular Linked Lists to …

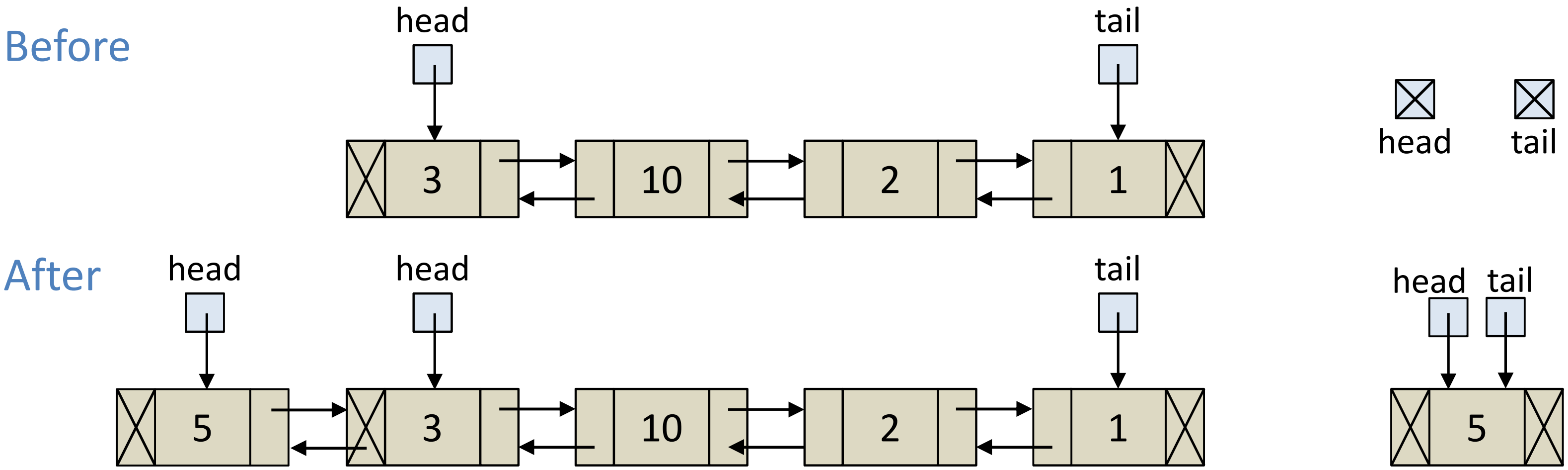
- `removeLast()` is still `O(n)` slow ☹
- Why?
  - We needed predecessor of tail
  - To get predecessor, we needed to traverse the list
- Solution?
  - Put references in both directions!
  - Keep track of head and tail for quick access from both ends

Doubly Linked List



Each `DoublyLinkedNode` has `value`, `next`, AND `previous`.

Doubly LLs: `addFirst`

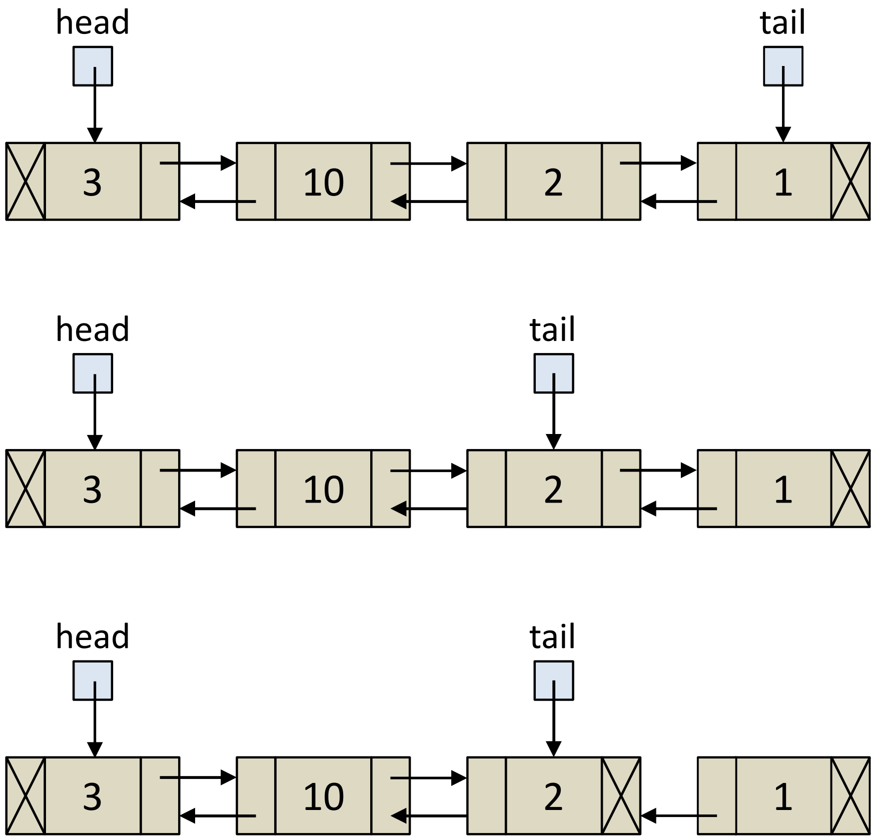


```
public void addFirst(E value)
{
    // construct a new element, making it head
    head = new DoublyLinkedNode<E>(value, head, null);
    // fix tail, if necessary
    if (tail == null) tail = head;
    count++;
}
```

`DoublyLinkedNode(value, next, previous)`
Constructor ensures consistency! When creating new node, if either `next` or `previous` is `non-null`, then constructor updates references in newly adjacent nodes.

Bonus: What changes for `addLast`?

Doubly LLs: `removeLast`



Set `tail` to `tail.previous()`.

Update references using `tail.setNext(null)`.

Bonus: This does not handle the case in which the list starts with a single element. What would you have to add?

Doubly LLs: Trade-offs

- `removeLast()` is finally `O(1)` efficient ☺
- At what cost?
  - All add and remove methods must set extra references
  - We must store additional `previous` field for each node

- Note
  - `java.util.LinkedList` are doubly linked lists
  - but we are using `DoublyLinkedList` from Bailey

Linked Lists Summary

What is the worst-case time complexity of each?

	singly linked list	circular list	doubly linked list
`addFirst(E value)`	`O(1)`	`O(1)`	`O(1)`
`getFirst()`	`O(1)`	`O(1)`	`O(1)`
`removeFirst()`	`O(1)`	`O(1)`	`O(1)`
`addLast(E value)`	`O(n)`	**`O(1)`**	`O(1)`
`getLast()`	`O(n)`	**`O(1)`**	`O(1)`
`removeLast()`	`O(n)`	`O(n)`	**`O(1)`**
`remove(E value)`	`O(n)`	`O(n)`	`O(n)`
`contains(E value)`	`O(n)`	`O(n)`	`O(n)`
Trade-offs	--	• Takes extra time to get to head	• More storage needed • Must change twice as many links when adding or deleting

Expectations

- You should be able to write any linked list method for any linked list variant
    - Any method, not just ones covered today
- Midterms always include such a question
    - Common technical interview questions too
- Use pictures!
- Don’t try to memorize them!

- Compact description of linked list variants: https://wiki.cs.auckland.ac.nz/compsci105ss/index.php/Linked_Lists

What can we do with linked lists?

- Implement several other common abstract data types
- Stacks
  - Last In, First Out (LIFO)
  - Only add to top (head), remove from top
- Queues
  - First In, First Out (FIFO)
  - Only add to back (tail), remove from front (head)

Next week!

- Deques (doubly ended queues, pronounced “deck”)
  - Only add to front or back, remove from front or back