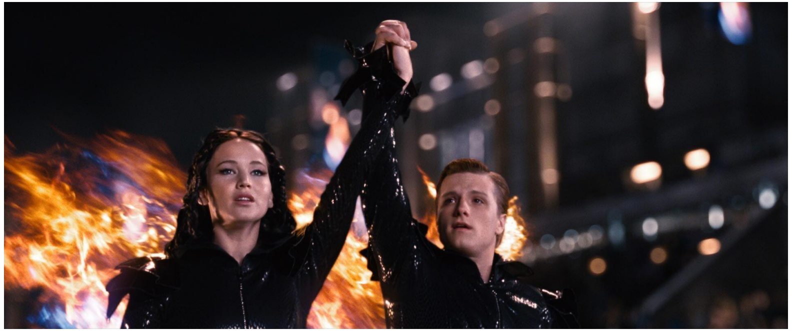
Lecture 18

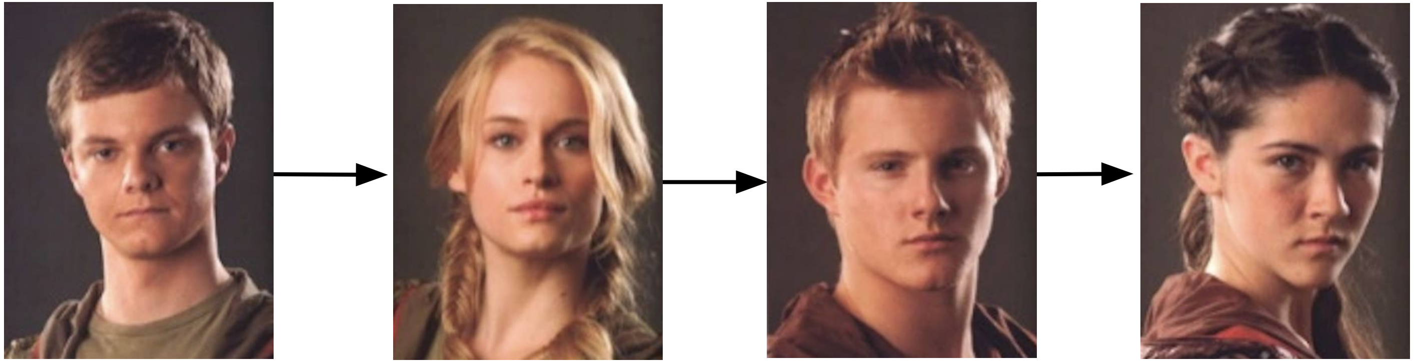
Data Structures I: LinkedLists

Outline



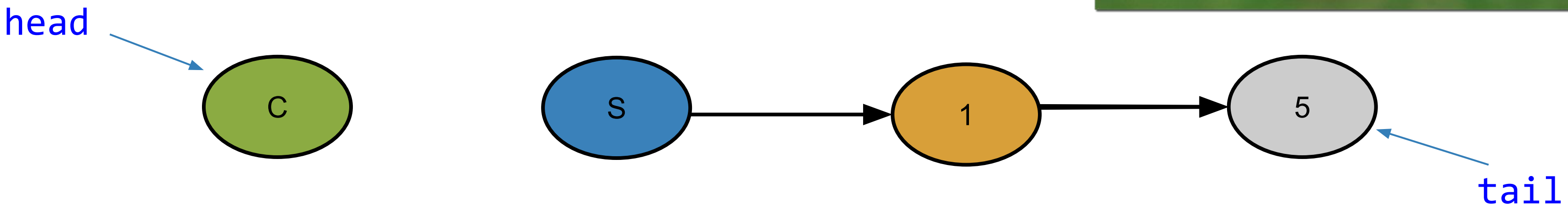
- Linked Lists
- Stacks and Queues (next lecture)
- Trees (next lecture)
- HashSets and HashMaps (next lecture)

Linked Lists



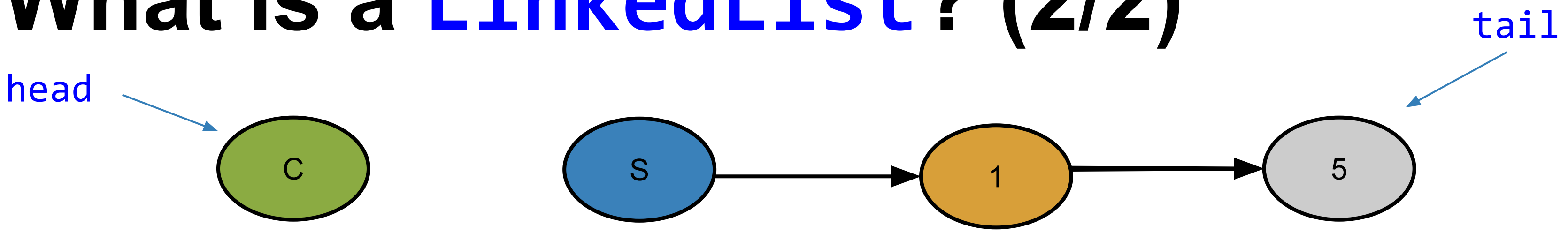
Tribute 1 Tribute 2 Tribute 3 Tribute 4

What is a `LinkedList`? (1/2)



- Collection of nodes stored anywhere in memory linked in a “daisy chain” to form sequence of elements
  - as with `Array`s and `ArrayList`s, it can represent an unordered set or an ordered (sorted) sequence of data elements
- A `LinkedList` holds a reference (pointer) to its first node (*head)* and its last node (*tail*) – internal nodes maintain list via their references to their next nodes

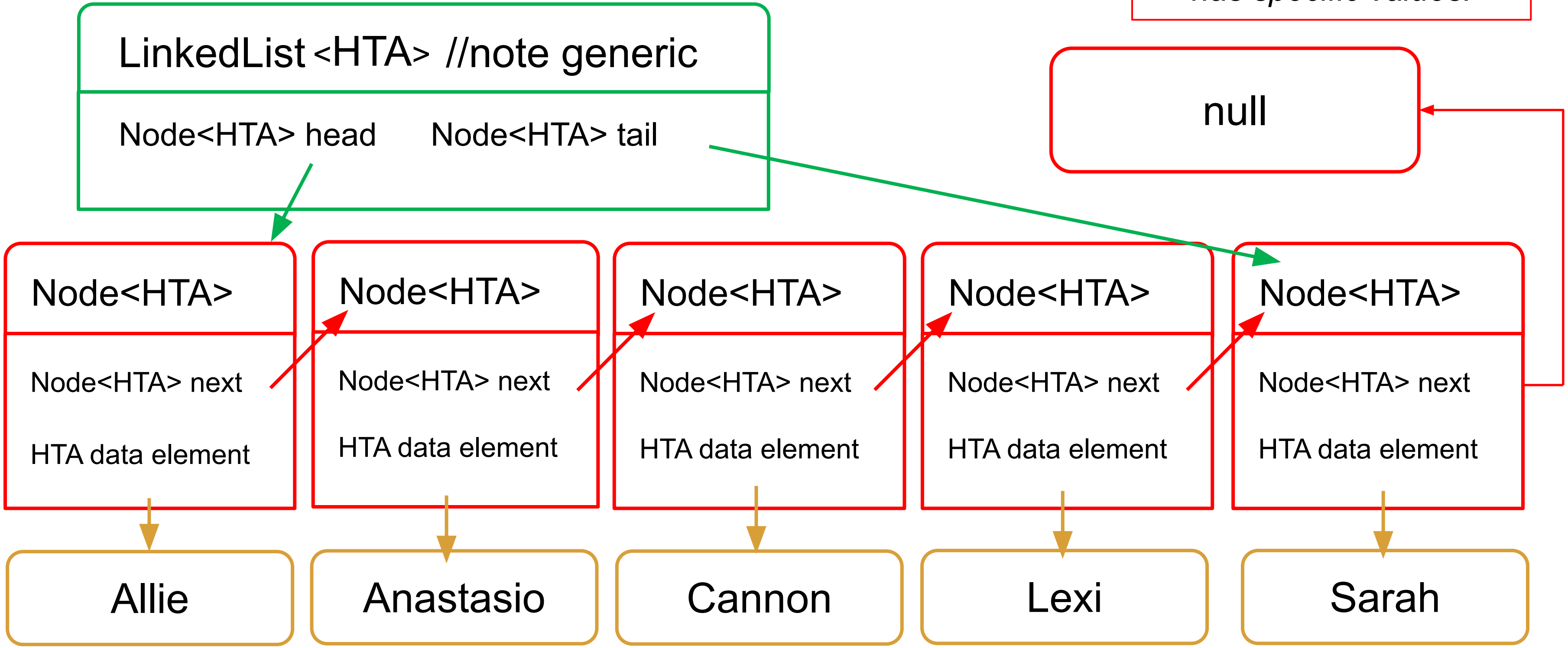
What is a LinkedList? (2/2)



- Each node holds an **element** and a **reference** to next node in list
- Most methods will involve:
  - “pointer-chasing” through the `LinkedList` (for `search` and finding correct place to insert or delete)
  - breaking and resetting the `LinkedList` to perform insertion or deletion of nodes
- But there won’t be data movement! Hence efficient for dynamic collections

Ex: HTA **`LinkedList`**

Note that this is an instance diagram, ***not*** *a class diagram, because it has specific values!*



When to Use Different Data Structures for Collections (1/2)

- `ArrayList`s get their name because they implement Java's `List` interface (defined soon) and are implemented using `Array`s
- `LinkedList`s also implement the `List` interface and are an alternative to `ArrayList`s that avoid data movement for insertion and deletion
    - uses pointer manipulation rather than moving elements in an array

When to Use Different Data Structures for Collections (2/2)

- How to decide between data structures?
  - choose based on the way data is *accessed* and *stored* in your algorithm
  - *access* and *store* operations of different data structures can have very different impacts on an algorithm's overall efficiency–recall Big-O analysis
  - even without N very large, there can be significant performance differences
  - roughly, `Arrays` if mostly static collection, `ArrayLists` if need more update dynamics while retaining easy accessibility, and `LinkedList` if more updates than accesses

Data Structure Comparison

`Array`

- Indexed (explicit access to i^{th} item)
- If user moves elements during insertion or deletion, their indices will change correspondingly
- Can't change size dynamically

`ArrayList`

- Indexed (explicit access to i^{th} item)
- Indices of successor items automatically updated following an inserted or deleted item
- Can grow/shrink dynamically
- Java uses an `Array` as underlying data structure (and does data shuffling itself)

`LinkedList`

- **Not** indexed – to access the n^{th} element, must start at the beginning and go to the next node n times → no random access!
- Can grow/shrink dynamically
- Uses nodes and pointers instead of `Array`s
- Can insert or remove nodes anywhere in the list without data movement through the rest of the list

Linked List Implementations (1/2)

- Find java.util implementation at:
  http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

- To learn list processing, we'll make our own implementation of this data structure, `MyLinkedList` (MLL):
  - difference between MLL and Java's implementation is that Java uses something like our MLL to build a more advanced data structure that implements Java's `List` interface
  - while there is overlap, there are also differences in the methods provided, and their names/return types
  - in CS200, you will use `LinkedLists` in your own programs

Linked List Implementations (2/2)

- `MyLinkedList` (MLL) is a general building block for more specialized data structures we'll build: `Stack`s, `Queue`s, Sorted `Linked List`s…

- We'll start by defining a **Singly Linked List** for both unsorted and sorted items, then we'll define a **Doubly Linked List** – users of these data structures don't see any of these internals!
    - will implement MLL as a **Singly Linked List** in next few slides

Singly Linked List (1/3)

- MLL doesn't implement full `List` interface
- Linked list is maintained by `head` and `tail` pointers; internal structure changes dynamically
- Constructor initializes instance variables
  - `head` and `tail` are initially set to null
  - `size` set to 0
- `addFirst()` appends `Node` to front of list and updates `head` to reference it
- `addLast()` appends `Node` to end of list and updates `tail` to reference it

```
public class MyLinkedList<CS15TA> {
        private Node<CS15TA> head;
        private Node<CS15TA> tail;
        private int size;

        public MyLinkedList() {
           this.head = null;
           this.tail = null;
           this.size = 0;
        }

        public Node<CS15TA> addFirst(CS15TA el) {
                //...
        }
        public Node<CS15TA> addLast(CS15TA el) {
                //...
        }

      // more on next slide
}
```

***Generic* – we literally code** *"<Type>" as a placeholder for the type chosen by the user of this data structure (ex.: `MyLinkedList<CS15TA>`, Java substitutes `CS15TA` with whatever `Type`)*

Singly Linked List (2/3)

- `removeFirst()` removes first `Node` and returns element
- `removeLast()` removes last `Node` and returns element
- `Remove()` removes first occurrence of `Node` containing element `el` and returns it (implicit `search`)

```
public Node<CS15TA> removeFirst() {
    //...
}

public Node<CS15TA> removeLast() {
    //...
}

public Node<CS15TA> remove(CS15TA el) {
    //...
}


// still more on next slide
```

Note: we have aligned methods of `LinkedList` and `ArrayList` where possible, with methods differing as the data structures differ (i.e., `ArrayList` has no `removeLast()` since you can get last element with index = length-1)

Singly Linked List (3/3)

- `search()` finds and returns `Node` containing `el`
- `size()` returns `size` of list
- `isEmpty()` checks if list is empty (returns `boolean`)
- `getHead/getTail()` return reference to head/tail `Node` of list

```
public Node<CS15TA> search(CS15TA el) {
    //...
}

public int size() {
    //...
}

public boolean isEmpty() {
    //...
}

public Node<CS15TA> getHead() {
    //...
}

public Node<CS15TA> getTail() {
    //...
}
```

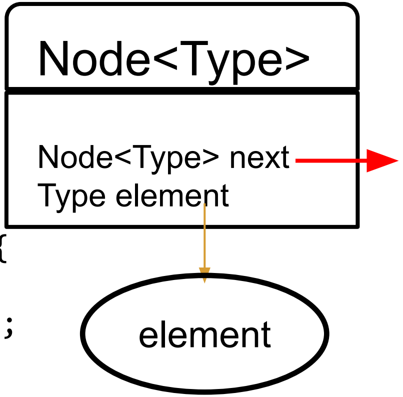
Singly Linked List Summary

```
public class MyLinkedList<CS15TA> {
    private Node<CS15TA> head;
    private Node<CS15TA> tail;
    private int size;

    public MyLinkedList() {
        //...
    }

    public Node<CS15TA> addFirst(CS15TA el) {
        //...
    }

    public Node<CS15TA> addLast(CS15TA el) {
        //...
    }

    public Node<CS15TA> removeFirst() {
        //...
    }

    public Node<CS15TA> removeLast() {
        //...
    }

    public Node<CS15TA> remove(CS15TA e1) {
        //...
    }

    public Node<CS15TA> search(CS15TA e1) {
        //...
    }

    public int size() {
        //...
    }

    public boolean isEmpty() {
        //...
    }

    public Node<CS15TA> getHead() {
        //...
    }

    public Node<CS15TA> getTail() {
        //...
    }
}
```

The Node Class

- Also uses **generics**; user of MLL specifies type and Java substitutes specified type in Node class' methods
- Constructor initializes instance variables `element` and `next`
- Its methods are made up of **accessors** and **mutators** for these variables:
    - `getNext()` and `setNext()`
    - `getElement()` and `setElement()`
- *Type* is a placeholder for whatever object Node will hold

```
public class Node<Type> {
    private Node<Type> next;
    private Type element;

    public Node(Type element) {
        this.next = null;
        this.element = element;
    }

    public Node<Type> getNext() {
        return this.next;
    }

    public void setNext(Node<Type> next) {
        this.next = next;
    }

    public Type getElement() {
        return this.element;
    }

    public void setElement(Type element) {
        this.element = element;
    }
}
```



Ex: A pile of Books

- Before implementing `LinkedList`'s internals, let's see how to **use** one to model a simple pile of `Book`s
  - "user" here is another programmer using the `MyLinkedList` we're making

- Elements in our pile will be of type `Book`
  - each has title, author(s), date and ISBN (International Standard Book Number)
  - want list that can store any `Book`

Book

`String author`
`String title`
`int isbn`

`getAuthor()`
`getTitle()`
`getISBN()`
`...`

Book Class

- Book's constructor stores author, date and ISBN number of Book as instance variables

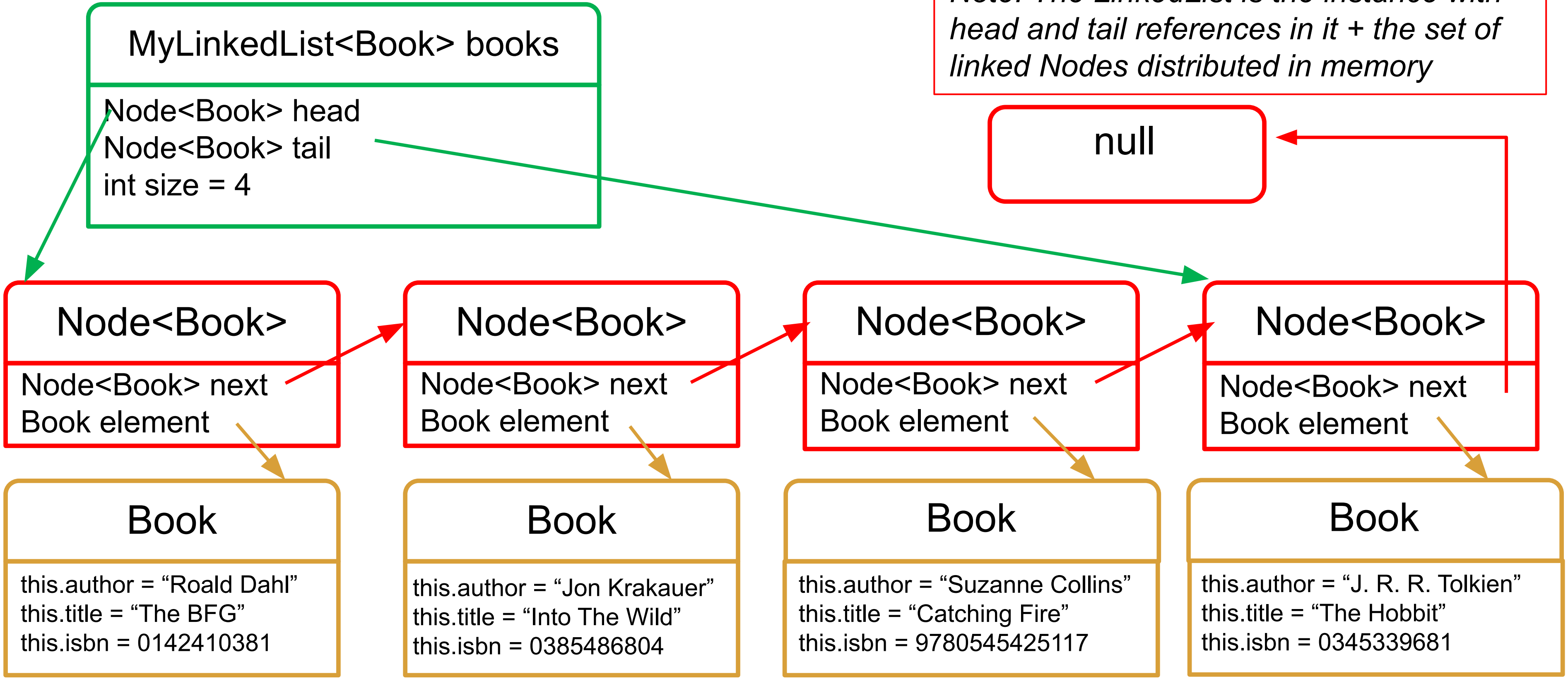
- For each property, get method returns that property's value
    - ex. getISBN() returns isbn

```
public class Book {
    private String author;
    private String title;
    private int isbn;

    public Book(String author,
                String title, int isbn) {
        this.author = author;
        this.title = title;
        this.isbn = isbn;
    }

    public int getISBN(){
        return this.isbn;
    }

    //other mutator and accessor
    //methods elided
}
```

Ex: MyLinkedList<Book>

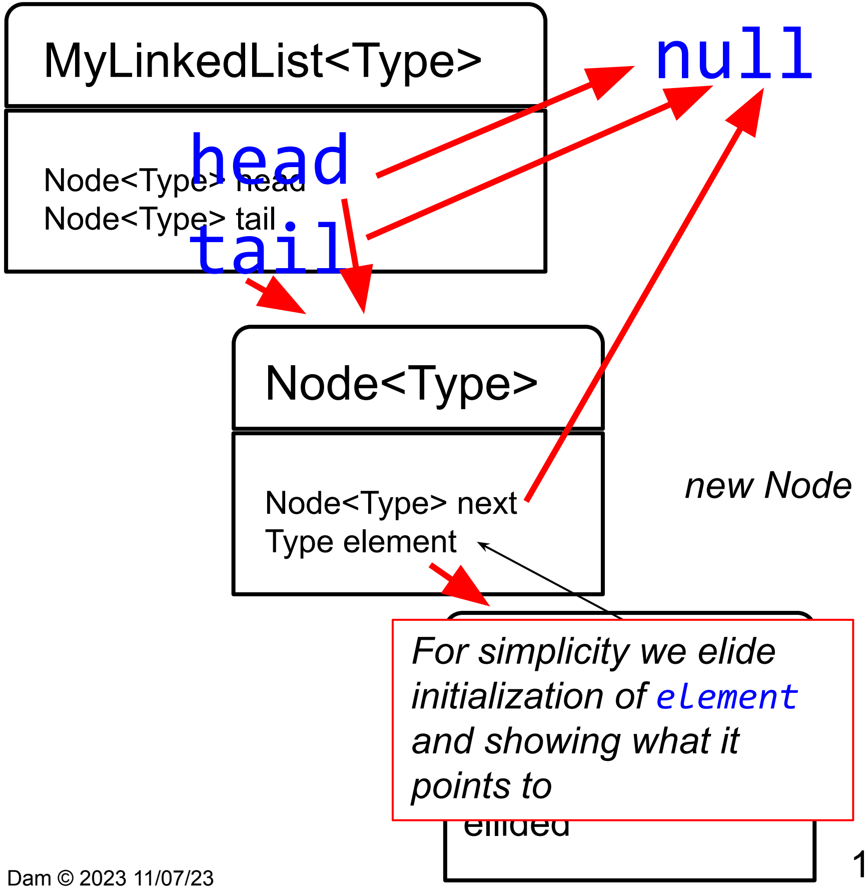
Note: all this machinery hidden from user!

Note: The LinkedList is the instance with head and tail references in it + the set of linked Nodes distributed in memory



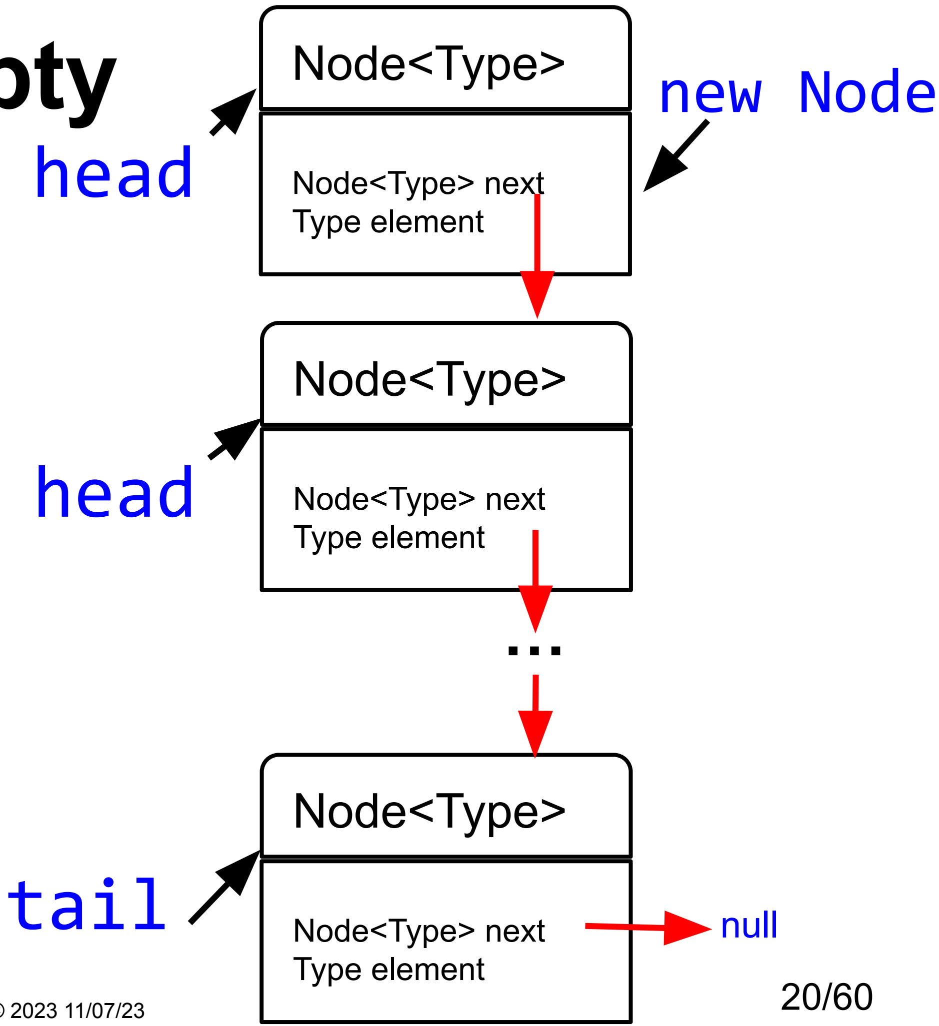
Implementation: addFirst – empty list

- If list is empty, head and tail are null
    - let's only show list pointers
- Create new Node<ElementType>
- Update new node's next variable to where head points to, which is null in this case
    - constructor already had null – we're accounting for general case
- Update head and tail variables to new node



addFirst – non empty

- Construct new Node
- Update its next variable to current head (in this case, some previously added Node that headed list)
- Update MLL’s head variable to the new Node



Constructor and `addFirst` Method (1/2)

- Constructor — as shown before
  - initialize instance variables
- `addFirst` method
  - increment `size` by 1
  - create new `Node` ((S15: constructor stores `el` in `element`, `null` in `next`)
  - update `newNode`'s `next` to first `Node` (pointed to by `head`)
  - update MLL's `head` to point to `newNode`
  - if `size` is 1, `tail` must also point to `newNode` (edge case)
  - return `newNode`

```
public MyLinkedList<Type>() {
    this.head = null;
    this.tail = null;
    this.size = 0;
}

public Node<Type> addFirst(Type el) {
    this.size++;
    Node<Type> newNode
              = new Node<Type>(el);
    newNode.setNext(this.head); //previous head
    this.head = newNode;

    if (size == 1) {
        this.tail = newNode;
    }

    return newNode;
}
```

Constructor and addFirst Runtime (2/2)

```
public MyLinkedList() {
    this.head = null;                   // 1 op
    this.tail = null;                   // 1 op
    this.size = 0;                      // 1 op
}
```

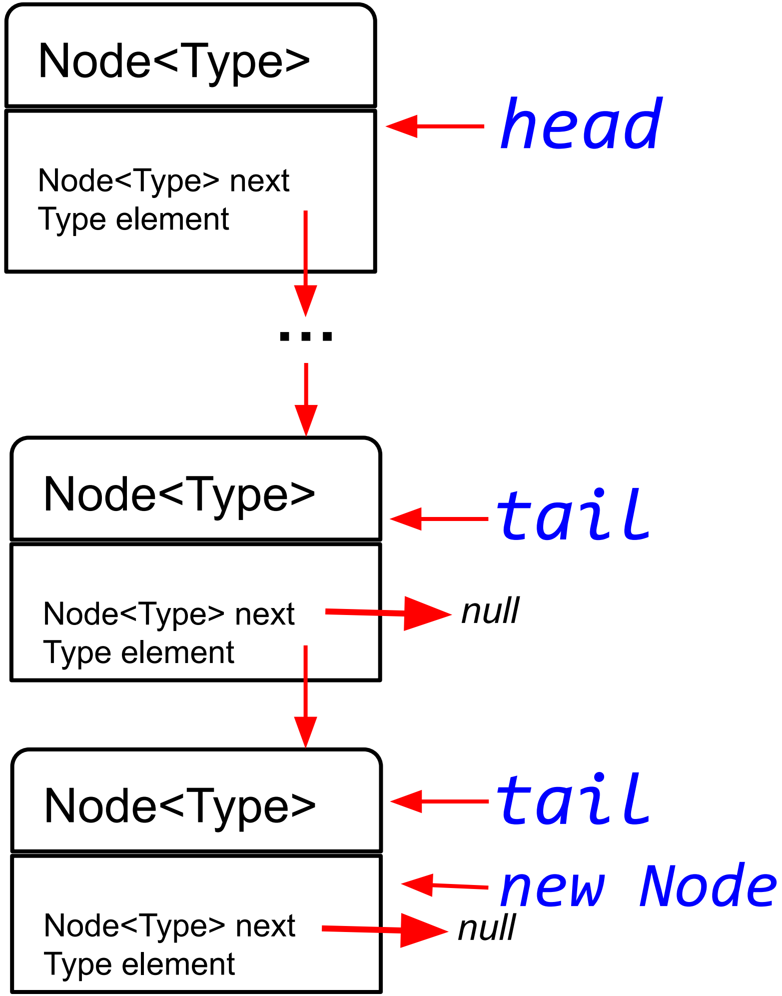
→ *constructor is O(1)*

```
public Node<Type> addFirst(Type el) {
    this.size++;                        // 1 op
    Node<Type> newNode = new Node<Type>(el);   // 1 op
    newNode.setNext(this.head);              // 1 op
    this.head = newNode;                     // 1 op

    if (size == 1) {                    // 1 op
        this.tail = newNode;            // 1 op
    }
    return newNode;                     // 1 op
}
```

→ *`addFirst(Type el)` is O(1)*

`addLast` Method (1/2)

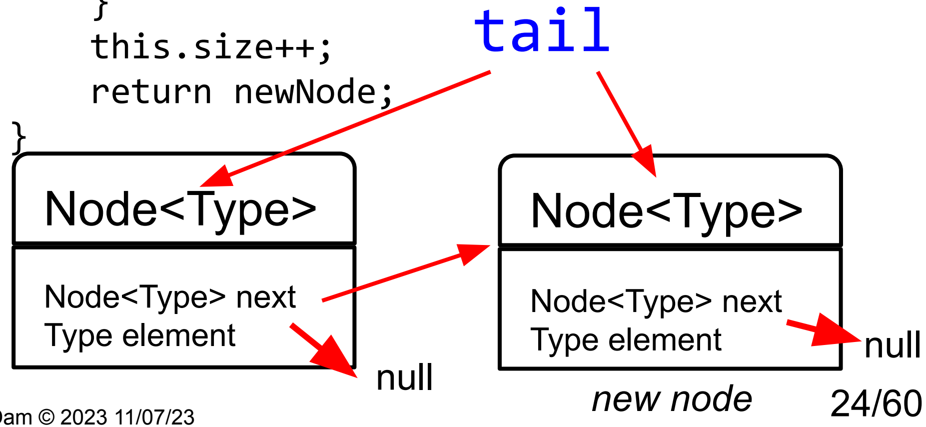
- MLL's `tail` already points to the last `Node` in the list
- Create a new `Node<Type>`
- Update `tail`'s node's `next` pointer to the new node
- Then, update `tail` to the new `Node`



addLast Method (2/2)

- Edge Case
  - if list is empty, update head and tail variables to newNode
- General Case
  - update next of current last Node (to which tail is pointing – “update tail’s next”) to new last Node
  - update tail to that new last Node
  - new Node’s next variable already points to null

```
public Node<Type> addLast(Type el) {
    Node<Type> newNode
            = new Node<Type>(el);
    if (this.size == 0) {
        this.head = newNode;
        this.tail = newNode;
    }
    else {
        this.tail.setNext(newNode);
        this.tail = newNode;
    }
    this.size++;
    return newNode;
}
```



addLast Runtime

```
public Node<Type> addLast(Type el) {
    Node<Type> newNode = new Node<Type>(el)        // 1 op
    if (this.size == 0) {                          // 1 op
        this.head = newNode;                       // 1 op
        this.tail = newNode;                       // 1 op
    }
    else {
        this.tail.setNext(newNode);                // 1 op
        this.tail = newNode;                       // 1 op
    }
    this.size++;                                   // 1 op
    return newNode;                                // 1 op
}
```

→ *addLast(Type el)* is *O(1)*

size and isEmpty Methods and Runtime

```
public int size() {
    return this.size;          // 1 op
}

public boolean isEmpty() {
     return this.size == 0;      // 2 ops
}
```

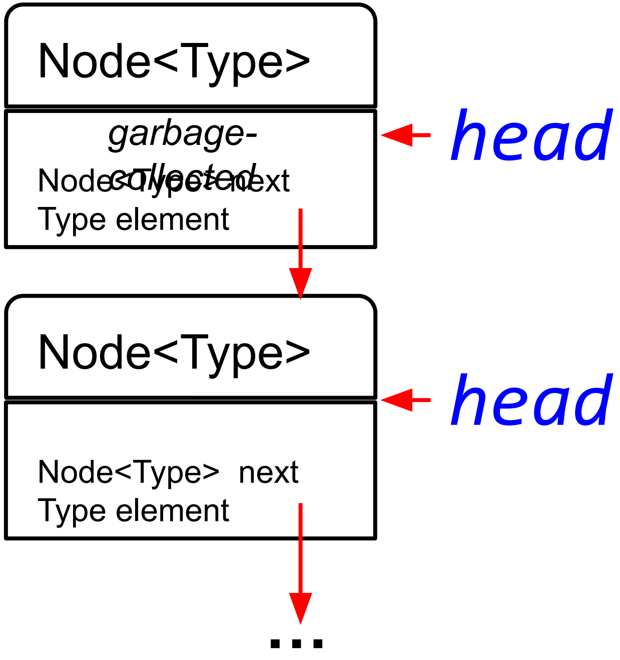
→ *size() is O(1)*

→ *isEmpty() is O(1)*

`removeFirst` Method (1/2)

- Remove reference to original first `Node` by setting `head` variable to second `Node`, i.e., first `Node`’s successor `Node,` via first’s `next`

- `Node` to remove is garbage-collected after termination of method



`removeFirst` Method (2/2)

- Edge case for empty list
  - `println` is optional, just one way to handle error checking; caller should check for null in any case
- Store data element from first `Node` to `removed`
- Then unchain first `Node` by resetting `head` to point to first `Node`’s successor
- If list is *now* empty, update `tail` to `null` (what did `head` get set to?)
- `Node` to remove is garbage-collected at method’s end

```
public Type removeFirst() {
    if (this.size == 0) {
        System.out.println(“List is empty”);
        return null;
    }

    Type removed = this.head.getElement();
    this.head = this.head.getNext();
    this.size--;
    if (this.size == 0) {
        this.tail = null;
    }
    return removed;
}
```

`removeFirst` Runtime

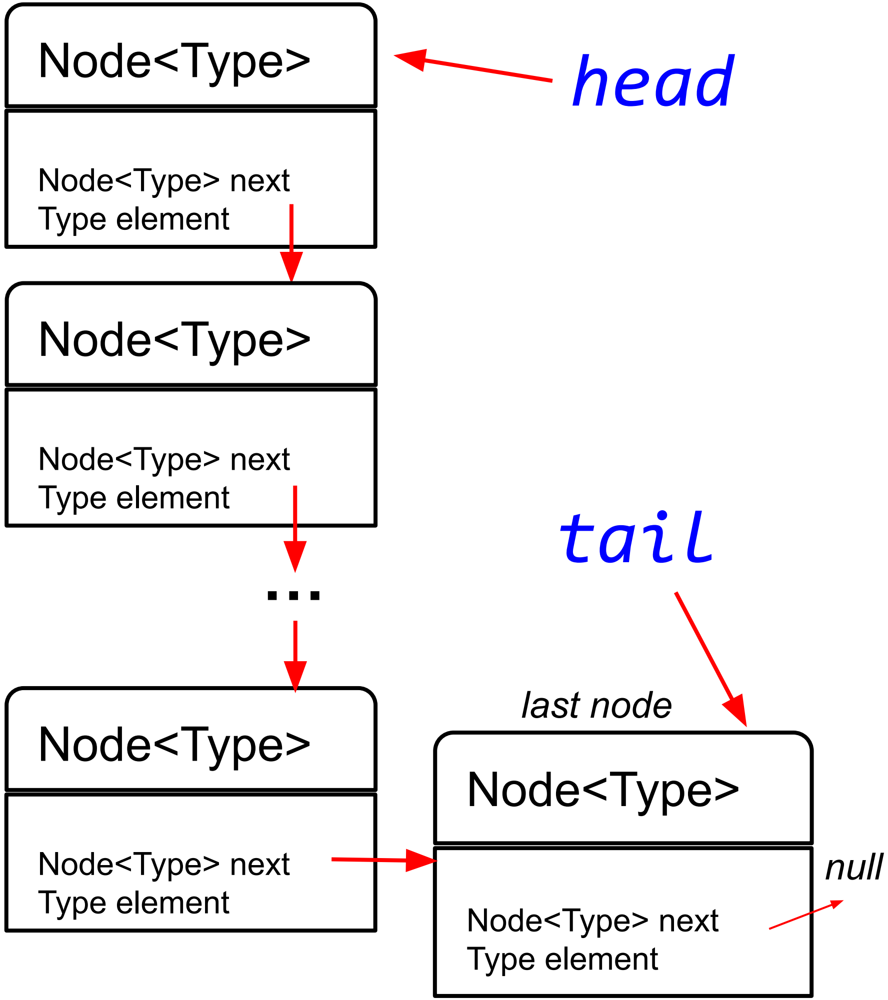
```
public Type removeFirst() {
    if (this.size == 0) {                        // 1 op
        System.out.println("List is empty");          // 1 op
        return null;                         // 1 op
    }

    Type removed = this.head.getElement();          // 1 op
    this.head = this.head.getNext();                // 1 op
    this.size--;                                 // 1 op
    if (this.size == 0) {                        // 1 op
        this.tail = null;                        // 1 op
    }
    return removed;                              // 1 op
}
```

→ *`removeFirst()`* *is O(1)*

Review: Accessing Nodes Via Pointers

`this.head.getNext();`

- This does not get `next` field of `head`, which doesn’t have such a field, being just a pointer
- Instead, read this as “get `next` field of the node `head` points to”
- What does `this.tail.getNext()` produce?
- What does `this.tail.getElement()` produce?
- note we can access a variable by its unique name, index, contents, or here, via a pointer



TopHat Question

Given a `Linked List` of `Nodes`,

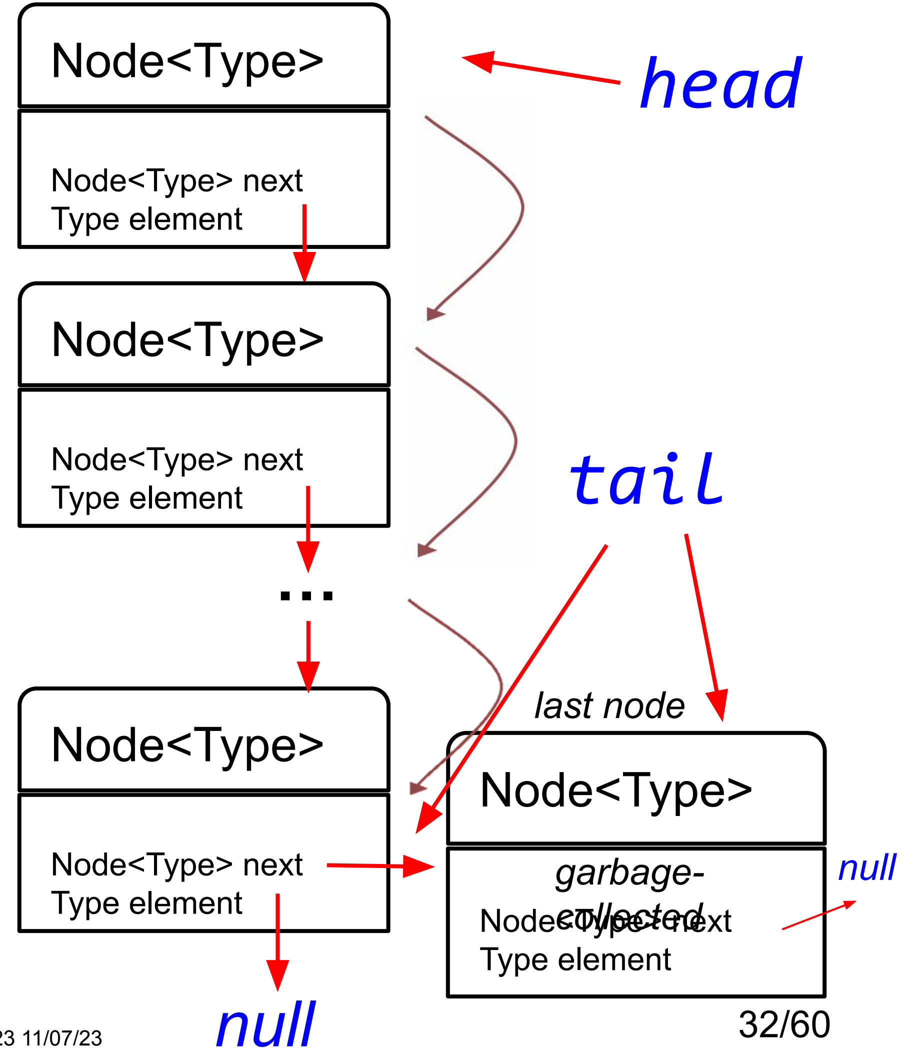
`A -> B -> C -> D`

where `head` points to node `A`, what is `this.head.getNext().getNext()`?

A. Nothing, throws a `NullPointerException`
B. `B`
C. `C`
D. `D`

removeLast Method

- As with removeFirst, remove Node by removing any references to it. Need to know predecessor, but no pointer to it!
- "Pointer-chase" in a loop until predecessor's next is tail and reset predecessor's next instance variable to null
  - very inefficient–stay tuned
- Update tail
- Last Node is thereby garbage-collected!

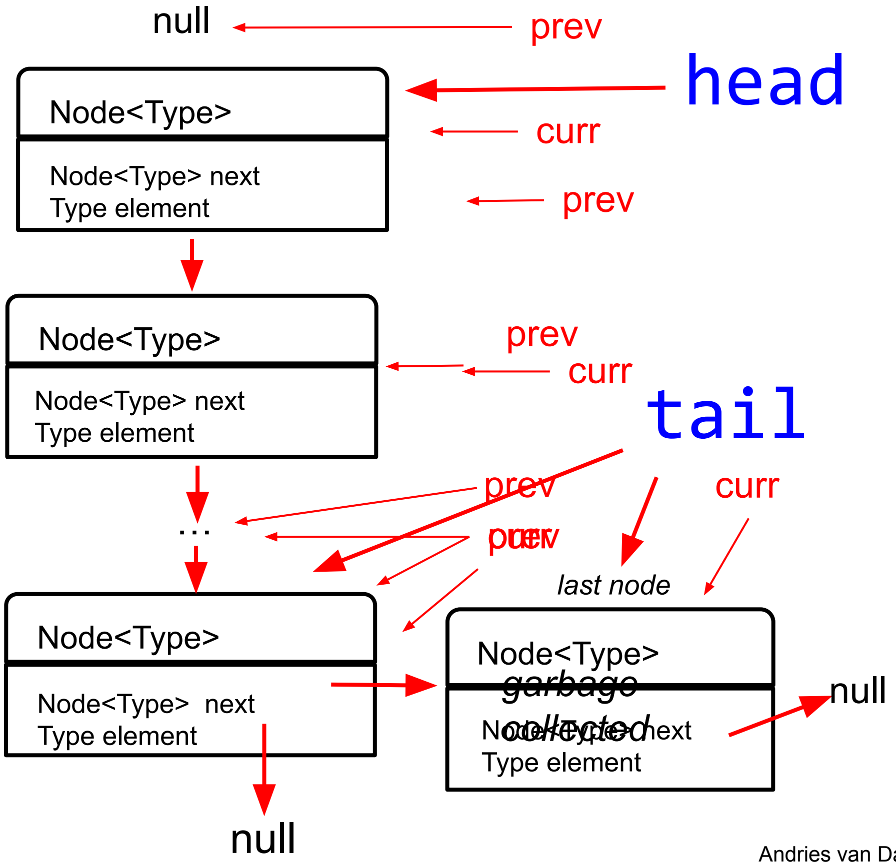


removeLast Method

- Edge case(s)
  - can't delete from empty list
  - if there's only one Node, update head and tail references to null
- General case
  - iterate ("pointer-chase") through list – common pattern using pointers to current and previous node in lockstep
  - after loop ends, prev will point to Node just before last Node and curr will point to last Node

```
public Type removeLast() {
    Type removed = null;
    if (this.size == 0) {
        System.out.println("List is empty");
    } else if (this.size == 1) {
        removed = this.head.getElement();
        this.head = null;
        this.tail = null;
        this.size = 0;
    } else { //classic pointer-chasing loop
        Node curr = this.head;
        Node prev = null;
        while (curr.getNext() != null) {
            //bop the pointers
            prev = curr;
            curr = curr.getNext();
        }
        removed = curr.getElement();
        prev.setNext(null); //unlink last
        this.tail = prev; //update tail
        this.size--;
    }
    return removed;
}
```

removeLast Method




```java
public Type removeLast() {
    Type removed = null;
    if (this.size == 0) {
        System.out.println("List is empty");
    } else if (this.size == 1) {
        removed = this.head.getElement();
        this.head = null;
        this.tail = null;
        this.size--;
    } else { //classic pointer-chasing loop
        Node curr = this.head;
        Node prev = null;
        while (curr.getNext() != null) {
            //bop the pointers
            prev = curr;
            curr = curr.getNext();
        }
        removed = curr.getElement();
        prev.setNext(null); //unlink last
        this.tail = prev; //update tail
        this.size--;
    }
    return removed;
}
```

removeLast Runtime

```
public Type removeLast() {
    Type removed = null;                        // 1 op
    if(this.size == 0) {                        // 1 op
        System.out.println("List is empty");           // 1 op
    }
    else if(this.size == 1) {                        // 1 op
        removed = this.head.getElement();              // 1 op
        this.head = null;                       // 1 op
        this.tail = null;                       // 1 op
        this.size--;                            // 1 op
    }
    else{
        Node curr = this.head;                         // 1 op
        Node prev = null;                       // 1 op
        while (curr.getNext() != null) {               // n ops
            prev = curr;                        // 1 op
            curr = curr.getNext();                   // 1 op
        }
        removed = curr.getElement();                 // 1 op
        prev.setNext(null);                          // 1 op
        this.tail = prev;                       // 1 op
        this.size--;                            // 1 op
    }
    return removed;                             // 1 op
}
```

→ *removeLast()* *is O(n)*

TopHat Question

Given that `animals` is a Singly Linked List of ***n*** animals, what is `node` pointing to?

```
curr = this.head;
prev = null;
while (curr.getNext().getNext() != null) {
    prev = curr;
    curr = curr.getNext();
}
node = curr.getNext();
```

A. Nothing useful, throws a `NullPointerException`
B. Points to the last node on the list
C. Points to the second node on the list
D. Points to the head of the list

`search` Method for `MyLinkedList`

- Loops through list until element is found or end is reached (`curr==null`)
- If a `Node`'s element is same as the argument, `return curr`
- If no elements match, `return null`

```
public Node<Type> search(Type el) {
    Node<Type> curr = this.head;

    while (curr != null) {
        if (curr.getElement().equals(el)) {
            return curr;
        }
        curr = curr.getNext(); //bop pointer
    }

    return null; //got to end of list w/o finding
}
```

search Runtime

```
public Node<Type> search(Type el) {
    Node<Type> curr = this.head;               // 1 op

    while (curr != null) {                     // n ops
        if (curr.getElement().equals(el)) {         // 1 op
            return curr;                       // 1 op
        }
        curr = curr.getNext();                 // 1 op
    }

    return null;                          // 1 op
}
```

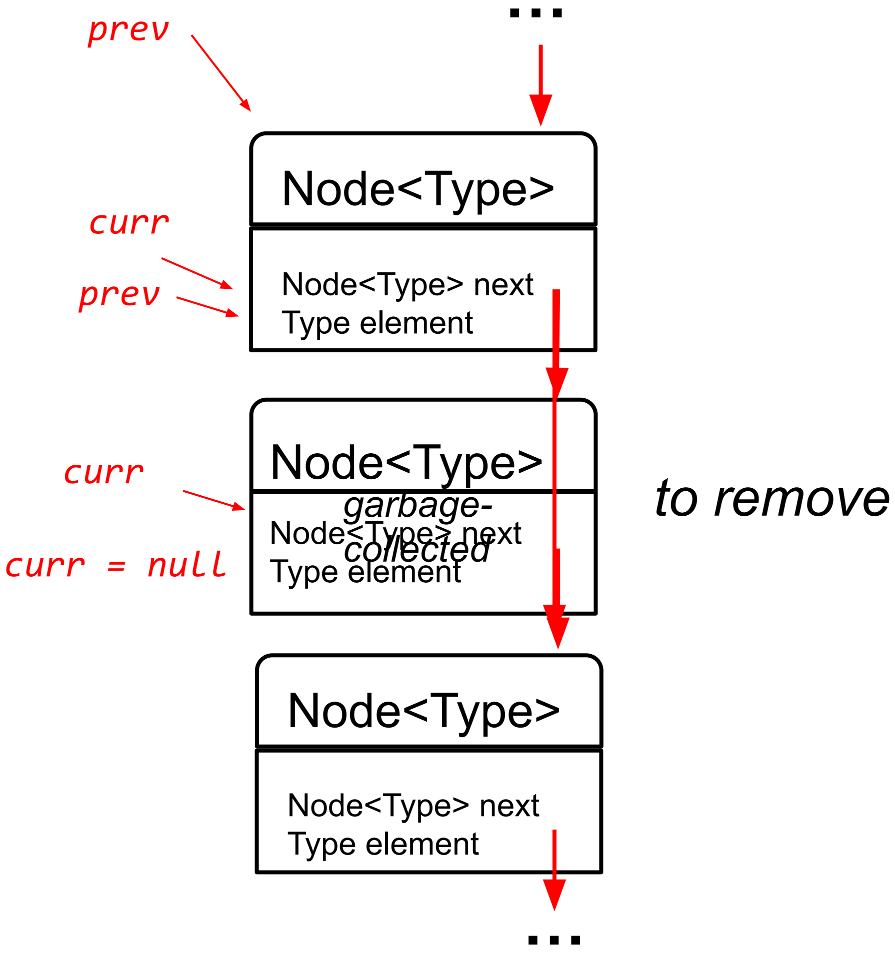
→ `search(Type el)` is *O(n)*

`remove` Method

- We have implemented methods to remove first and last `element`s of `MyLinkedList`
- What if we want to remove *any* `element` from `MyLinkedList`?
- Let's write a general `remove` method
  - think of it in 2 phases:
    - a search loop to find correct element (or end of list)
    - breaking the chain to jump over the element to be removed

remove Method

- Loop through Nodes until an element matches itemToRemove
- “Jump over” Node by re-linking predecessor of Node (again using loop’s prev pointer) to successor of Node (via its next reference)
- With no more reference to Node, it is garbage collected at termination of method



`remove` Method

- Edge Case(s)
  - again: can't delete from empty list
  - if removing first item or last item, delegate to `removeFirst`/`removeLast`
- General Case
  - iterate over list until `itemToRemove` is found in ptr-chasing loop
  - again: need `prev`, so we can re-link predecessor of `curr`. `Node` is GC'd upon `return`.

```
public Type remove(Type itemToRemove){
    if (this.isEmpty()) {
        System.out.println("List is empty");
        return null;
    }
    if (itemToRemove.equals(this.head.getElement())) {
        return this.removeFirst();
    }
    if (itemToRemove.equals(this.tail.getElement())) {
        return this.removeLast();
    }
    //advance to 2nd item
    Node<Type> curr = this.head.getNext();
    Node<Type> prev = this.head;
    while (curr != null) {//pointer-chasing loop to find el.
        if (curr.getElement().equals(itemToRemove)) {
            prev.setNext(curr.getNext()); //jump over node
            this.size--; //decrement size
            return curr.getElement();
        }
        prev = curr;  //if not found, bop pointers
        curr = curr.getNext();
    }
    return null; //return null if itemToRemove is not found
}
```

Note: caller of remove can find out if item was successfully found (and removed) by testing for `!= null`

remove Runtime

```
public Type remove(Type itemToRemove){
    if (this.isEmpty()) {                  // 1 op
        System.out.println("List is empty");     // 1 op
        return null;
    }
    if (itemToRemove.equals(this.head.getElement())) { // 1 op
        return this.removeFirst();              // O(1)
    }
    if (itemToRemove.equals(this.tail.getElement())) { // 1 op
        return this.removeLast();              // O(n) pointer chase till list end
    }
    Node<Type> curr = this.head.getNext();  // 1 op
    Node<Type> prev = this.head;           // 1 op
    while (curr != null) {                 // n ops
        if (itemToRemove.equals(curr.getElement())) {  // 1 op
            prev.setNext(curr.getNext());          // 1 op
            this.size--;                   // 1 op
          return curr.getElement();        // 1 op
        }
        prev = curr;                       // 1 op
        curr = curr.getNext();                 // 1 op
    }
    return null;                           // 1 op
}
```

→ *remove(Type itemToRemove)* is *O(n)*

TopHat Question

Given that `animals` is a Singly Linked List of ***n*** animals, `curr` points to the node with an animal to be removed from the list, that `prev` points to `curr`'s predecessor, and that `curr` is not the tail of the list, what will this code fragment do?

```
prev.setNext(curr.getNext());
curr = prev.getNext();
System.out.println(curr.getElement());
```

A. List is unchanged, prints out removed animal
B. List is unchanged, prints out the animal after the one that got removed
C. List loses an animal, prints out removed animal
D. List loses an animal, prints out the animal after the one that was removed

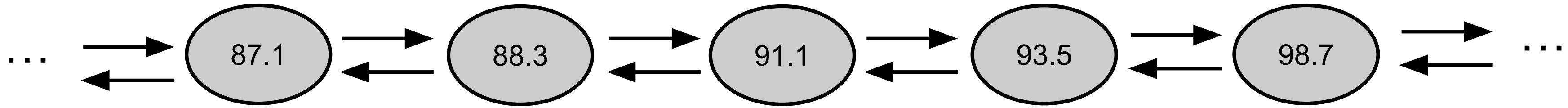
Doubly Linked List (1/3)

- Is there an easier/faster way to get to previous node while removing a node?
  - with Doubly Linked Lists, nodes have references both to next and previous nodes

  - can traverse list both backwards and forwards – Linked List still stores reference to front of list with `head` and back of list with `tail`

  - modify `Node` class to have *two* pointers: `next` and `prev`
  - eliminates pointer-chasing loop because `prev` points to predecessor of every `Node`, at cost of second pointer
    - classic space-time tradeoff!

Doubly Linked List (2/3)



- For Singly Linked List, processing typically goes from first to last node, e.g. `search`, finding place to insert or delete

- Sometimes, particularly for sorted list, need to go in the opposite direction
  - e.g., sort CS15 students on their final grades in ascending order. Find lowest numeric grade that will be recorded as an “A”. Then ask: who has a lower grade but is closer to the “A” cut-off, i.e., in the grey area, and therefore should be considered for “benefit of the doubt”?

Doubly Linked List (3/3)

- This kind of backing-up can't easily be done with the Singly Linked List implementation we have so far
  - could build our own *specialized* `search` method, which would scan from the `head` and be, at a minimum, O(n)

- It is simpler for Doubly Linked Lists:
  - find student with lowest "A" using search
  - use `prev` pointer, which points to the predecessor of a node (O(1)), and back up until hit end of B+/A- grey area

Announcements

- Tetris is out!
  - early handin: Saturday 11/11
  - on-time handin: Monday 11/13
  - late handin: Wednesday 11/15
  - Tetris Code-Along 11/08 7:00pm Friedman Hall
    - Recording on Website
- HTA hours in Friedman 101 Friday 3pm-4pm
  - come and chat about course registration, the upcoming final project or any other concerns you may have ☺

Cybersecurity and the Future of Warfare

CS15 Fall 2023

Cybersecurity: A Brief History

Andy with IBM Graphics Display Unit, 1968

1969 — The Pentagon develops the ARPANET, an early computer network.

1971 — Bob Thomas develops the world's first worm, the , "creeper."

1973 — Ray Tomlinson develops the first cybersecurity program, the "reaper".

1983 — ARPANET develops into the internet and becomes widely used.

Source: History of Computer Security

What is Cybersecurity?

"Cybersecurity is the art of protecting networks, devices, and data from unauthorized access or criminal use and the practice of ensuring confidentiality, integrity, and availability of information."
— *United States Cybersecurity & Infrastructure Security Agency*

[Ugrad] Phishing/scam message about summer break research

Fisler, Kathi Thu, Jun 8, 12:36 PM
to ugrad

Several Brown CS students (and faculty) have just reported receiving an email about a paid summer internship with me. Unfortunately, that is a phishing/scam message. Please don't send information to the text number in the message or reply to the sender.

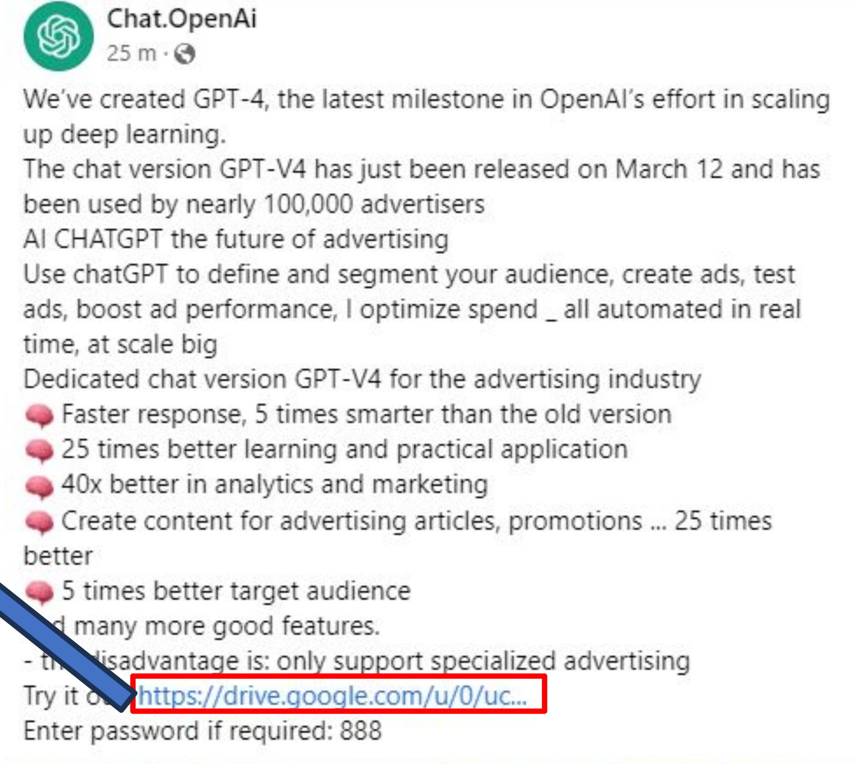
Brown IT is also being alerted about this.

Image Sources: Flat Icon

Chat GPT’s Popularity Leveraged to Spread Malware

https://google.drive.com/u/0/uc...

Link to malware

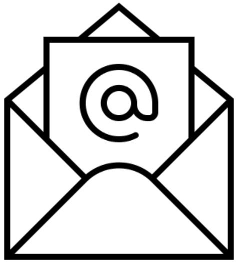


Threat actors using Adam Erhat, a well-known YouTuber market strategist to earn trust and facilitate this campaign

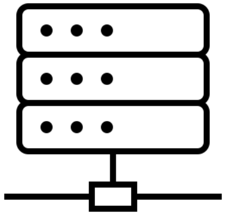
Source: CloudSEK

How Hackers Use Data: Ransomware

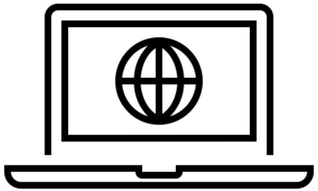
"Ransomware is a type of malware that locks a victim's data or device and threatens to keep it locked—or worse—unless the victim pays a ransom to the attacker."
— IBM



Scam emails



Server vulnerabilities

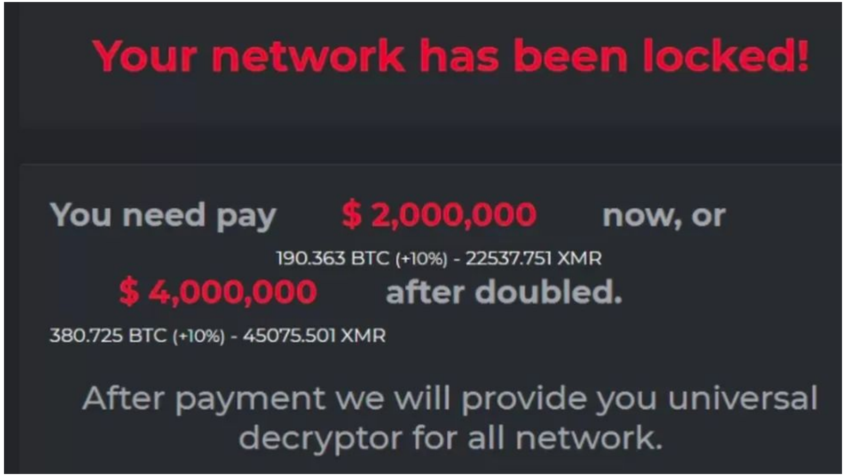


Infected websites

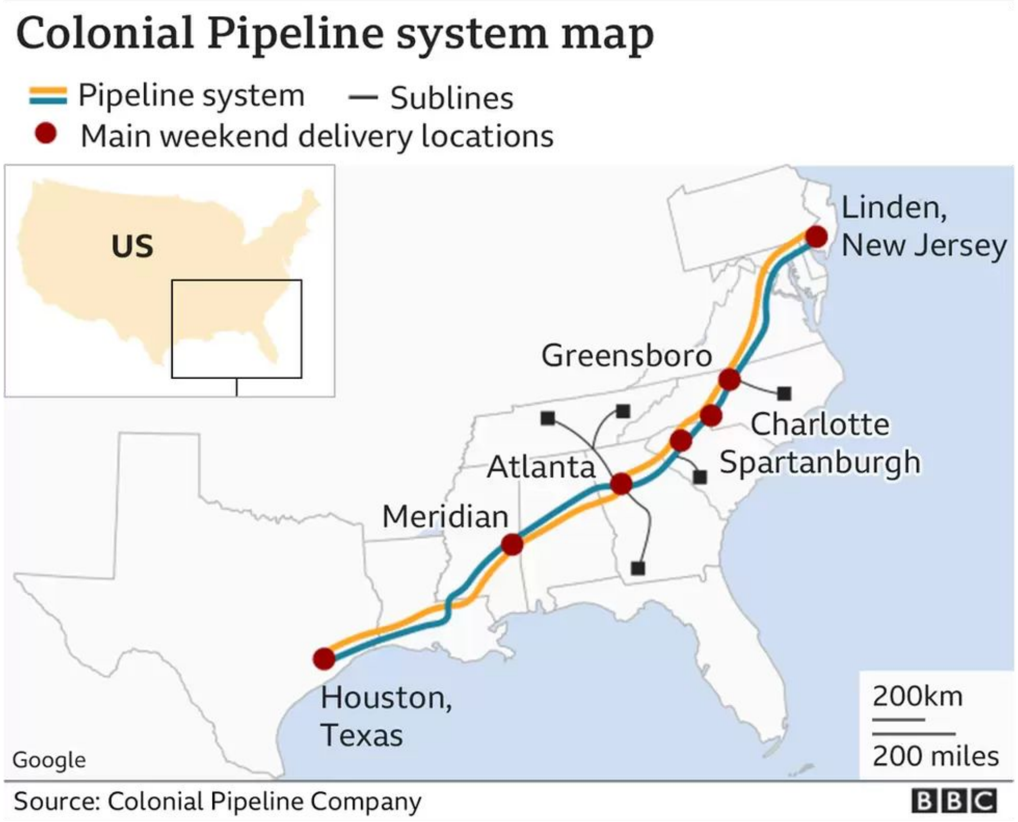
Online Ads

Source: Federal Trade Commission

Case Study: Colonial Pipelines Ransomware Attack



Example ransom message from DarkSide, the group that hacked Colonial Pipelines



Source: BBC, CISA, Reuters

Brown University hit by cyberattack, some systems still offline

By **Sergiu Gatlan** April 2, 2021 04:01 PM

Special_Events@brown.edu 08:43
A new day
You deets has been obtain by viper crewz https://
[redacted]

~500,000 email addresses were compromised in the 2021 cyberattack – this is the message leaked emails would receive

Anatomy of a phish: Questions to ask yourself when you receive a suspicious email

From: **BROWN Alert** <dr.jamwil@gmail.com>
Date: Mon, Aug 28, 2023 at 9:52 AM
Subject: Help Desk: Action Required
To:

Why would a Brown Alert be sent from dr.jamewil@gmail.com?

Why is the To field blank and not addressed to me?

BROWN

Is the Brown logo supposed to make it look official?

Is there a threat to try and get me to act promptly? This example includes one, saying "complete the process in order to avoid suspension."

Hello,

This is to notify active staff, students and alumni that all mailbox accounts are being validated. Use the link below to complete the process in order to avoid suspension.

VALIDATE MY MAILBOX

Would OIT ask me to click on a link? And why when I cursor over it does the URL look suspicious?

IT Support Desk

Who is this generic "IT Support Desk", why is there no mention of OIT, or a phone number provided to call with questions?

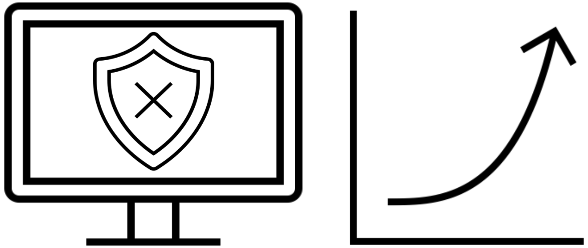
Source: BleepingComputer, GoLocal Prov, Brown University

Case Study: SolarWinds Cyber Attack

“As of today, 9 federal agencies and about 100 private sector companies were compromised.” *–Anne Neuberger, Deputy National Security Advisor*

Source: White House, Microsoft, CNET

Cybersecurity + International Affairs



As cyberattacks become more common…

…cybersecurity groups work together globally!

Communications Security Establishment
Centre de la sécurité des télécommunications

Canadian Centre for Cyber Security
Centre canadien pour la cybersécurité

National Cyber Security Centre
a part of GCHQ

Australian Government
Australian Signals Directorate

ACSC Australian Cyber Security Centre

National Cyber Security Centre
PART OF THE GCSB

Groups that helped neutralize the Russian malware "Snake," a cyber-espionage malware found in over 50 countries

Source: NSA

Future of cybersecurity

Executive Order on Improving the Nation's Cybersecurity

Source: NYTimes, The White House

Cybersecurity at Brown

BROWN | Office of Information Technology

Courses at Brown:

CSCI 1040: The Basics of Cryptographic Systems

CSCI 1360: Humans Factors in Cybersecurity

CSCI 1660: Introduction to Computer Security

CSCI 1800: Cybersecurity and International Relations

CSCI 1870: Cybersecurity Ethics

CSCI 2660: Computer Security