Lecture 18
More Data Structures for Collections: Part 1
Outline

• **Linked Lists**

• **Stacks and Queues**

• **Trees** (next lecture)

• **HashSets and HashMaps** (next lecture)
Linked Lists
What is a **LinkedList**? (1/2)

- A collection of nodes stored anywhere in memory that are linked in a “daisy chain” to form a sequence of elements
  - as with **Arrays** and **ArrayLists**, it can represent an unordered set or an ordered (sorted) sequence of your data elements
- A **LinkedList** holds a reference (pointer) to its first node (**head**) and its last node (**tail**) – the internal nodes maintain the list via their references to their next nodes
What is a **LinkedList?** (2/2)

- Each node holds an **element** and a **reference** to the next node in the list

- Most methods will involve:
  - “pointer-chasing” through the **LinkedList** (for **search** and finding the 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

LinkedList<HTA> //note generic

Node<HTA> head  Node<HTA> tail

null

Node<HTA>
Node<HTA>
Node<HTA>
Node<HTA>
Node<HTA>

Node<HTA> next
Node<HTA> next
Node<HTA> next
Node<HTA> next
Node<HTA> next

HTA data element
HTA data element
HTA data element
HTA data element
HTA data element

Daniel
Harriet
Lila
UV
Will

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)

- **ArrayLists** get their name because they implement Java’s **List** interface (defined soon) and are implemented using **Arrays**

- We define a building block called **LinkedList**, an alternative to **ArrayLists** that avoids data movement for insertion and deletion
  - by using 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?
  o choose based on the way data is accessed and stored in your algorithm
  o access and store operations of different data structures can have very different impacts on an algorithm’s overall efficiency—recall Big-O analysis
  o even without N very large, there can be significant performance differences
  o roughly, Arrays if mostly static collection, ArrayLists if need more update dynamics, and LinkedList if more updates than searches
# Data Structure Comparison

<table>
<thead>
<tr>
<th><strong>Array</strong></th>
<th><strong>ArrayList</strong></th>
<th><strong>LinkedList</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed (explicit access to $i^{th}$ item)</td>
<td>Indexed (explicit access to $i^{th}$ item)</td>
<td><strong>Not</strong> indexed – to access the $n^{th}$ element, must start at the beginning and go to the next node $n$ times → no random access!</td>
</tr>
<tr>
<td>If user moves elements during insertion or deletion, their indices will change correspondingly</td>
<td>Indices of successor items automatically updated following an inserted or deleted item</td>
<td>Can grow/shrink dynamically</td>
</tr>
<tr>
<td>Cannot change size dynamically</td>
<td>Can grow/shrink dynamically</td>
<td>Uses nodes and pointers instead of Arrays</td>
</tr>
<tr>
<td>Java uses an Array as the underlying data structure (and does the data shuffling)</td>
<td></td>
<td>Can insert or remove nodes anywhere in the list without data movement through the rest of the list</td>
</tr>
</tbody>
</table>
Linked List Implementations

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

- To learn list processing, we are going to make our own implementation of this data structure, MyLinkedList:
  - difference between MyLinkedList and Java’s implementation is that Java uses something like our MyLinkedList 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, their names, and their return types
  - in 200, you will use Java’s LinkedList in your own programs

- MyLinkedList is a general building block for more specialized versions we’ll build: Stacks, Queues, Sorted Linked Lists…

- 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 this data structure don’t see any of these internals!
Singly Linked List (1/3)

- Linked list is maintained by **head** and **tail** pointers; the internal structure changes dynamically
- Constructor initializes instance variables
  - **head** and **tail** are initially set to null
  - **size** set to 0
- **addFirst** creates first Node and updates **head** to reference it
- **addLast** appends a Node to the end of the list and updates **tail** to reference it

```java
class MyLinkedList<Type> {
    private Node<Type> head;
    private Node<Type> tail;
    private int size;

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

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

    public Node<Type> addLast(Type 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<Integer>, Java substitutes Integer where Type is)*
Singly Linked List (2/3)

- **removeFirst** removes first Node and returns element
- **removeLast** removes last Node and returns element
- **remove** removes the first occurrence of a Node containing the element `el` and returns it

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

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

public Node<Type> remove(Type 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 the last element with index = length-1)
Singly Linked List (3/3)

- **search** finds and returns the **Node** containing **el** or **null** (note the difference with **remove**)
- **size** returns **size** of the list
- **isEmpty** checks if the list is empty
- **getHead/getTail** return a reference to the head/tail **Node** of the list

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

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

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

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

public Node<Type> getTail() {
    //...
}
```
Singly Linked List Summary

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

    public MyLinkedList() {
        //...
    }

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

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

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

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

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

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

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

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

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

    public Node<Type> getTail() {
        //...
    }
}
```
The **Node** Class

- Also uses generics; user of MLL specifies type and Java substitutes the specified type in the 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()`

```java
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 internals, let’s see how to use one to model a simple pile of Books
  o “user” here is another programmer
• The elements in our pile will be of type Book
  o each has title, author(s), date and ISBN (International Standard Book Number) number
  o we want a list that can store any Book

<table>
<thead>
<tr>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>String author</td>
</tr>
<tr>
<td>String title</td>
</tr>
<tr>
<td>int isbn</td>
</tr>
<tr>
<td>getAuthor()</td>
</tr>
<tr>
<td>getTitle()</td>
</tr>
<tr>
<td>getISBN()</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
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>`

- `MyLinkedList<Book>` `books`
  - Node<Book> `head`
  - Node<Book> `tail`
  - int `size = 4`

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

Note: all this machinery hidden from user!

```
null
```

- Node<Book> `next`
  - Book element

- Node<Book> `next`
  - Book element

- Node<Book> `next`
  - Book element

- Node<Book> `next`
  - Book element

- Book
  - this.author = "Roald Dahl"
  - this.title = "The BFG"
  - this.isbn = 0142410381

- Book
  - this.author = "Jon Krakauer"
  - this.title = "Into The Wild"
  - this.isbn = 0385486804

- Book
  - this.author = "Neal Stephenson"
  - this.title = "Snow Crash"
  - this.isbn = 055308853

- Book
  - this.author = "J. R. R. Tolkien"
  - this.title = "The Hobbit"
  - this.isbn = 0345339681

Andries van Dam © 2021 11/09/21
Implementation: `addFirst` – empty list

- If list is empty, **head** and **tail** are **null**
  - let’s only show the list pointers

- Create a new `Node<ElementType>`

- Update new node’s **next** variable to **null**, which is where current **head** and **tail** point in this case

- Update the **head** and **tail** variables to the new node

For simplicity we elide initialization of **element** and showing what it points to
**addFirst** – non empty

- **Construct new Node**

- **Initialize 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** ((S14: its 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
  - return newNode

```java
public MyLinkedList() {
    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 ─ as shown before
  - initialize instance variables
- addFirst method
  - increment size by 1
  - create new **Node** ((S14: its 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
  - return newNode
Constructor and **addFirst** Runtime (2/2)

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

→ **constructor is O(1)**

```java
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) {
        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 the head and tail variables to the 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

```java
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

```java
public Node<Type> addLast(Type el) {
    Node<Type> newNode = new Node<Type>(el); // 1 op
    if (this.size == 0) {
        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

```java
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 the termination of the 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 be 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

```java
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

```java
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

```java
this.head.getNext();
```

- This does not get the `next` field of `head`, which doesn’t have such a field, being just a pointer
- Instead, read this as “get the `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, its index, its contents, or here, via a pointer
TopHat Question

Given a **Linked List** of Nodes,

\[ A \rightarrow B \rightarrow C \rightarrow D \]

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

A. Nothing, throws a `NullPointerException`
B. `C`
C. `B`
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 to get predecessor to tail and reset predecessor’s next instance variable to null
  o very inefficient—stay tuned
• Update tail
• Last Node is thereby garbage-collected!
```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 = 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;
}
```

- **Edge case(s)**
  - can’t delete from an empty list
  - if there is 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--;
    } 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;
}
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{  // 1 op
        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
}
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 input, return **curr** (note: returning always exits a method)
• If no elements match, return **null**

```java
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

```java
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 the first and the last elements of MyLinkedList

• What if we want to remove any element from MyLinkedList?

• Let’s write a general remove method
  o think of it in 2 phases:
    – a search loop to find the right 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)
  o again: can’t delete from an empty list
  o if removing first item or last item, delegate to removeFirst/removeLast

• General Case
  o iterate over list until itemToRemove is found in ptr-chasing loop
  o 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
public Type remove(Type itemToRemoval){
    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
}
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());
temp = prev.getNext();
System.out.println(temp.getElement());

A. List is unbroken, prints out removed animal
B. List is broken, prints out removed animal
C. List loses an animal, is intact, and prints out removed animal
D. List loses an animal, is intact, and prints out the animal after the one that was removed
Doubly Linked List (1/3)

• Is there an easier/faster way to get to the previous node while removing a node?

  o with Doubly Linked Lists, nodes have references both to next and previous nodes
  
  o can traverse list both backwards and forwards – Linked List still stores reference to front of the list with head and back of the list with tail
  
  o modify Node class to have two pointers: next and prev
Doubly Linked List (2/3)

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

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

```
...  87.1  88.3  91.1  93.5  98.7  ...
```

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 the lowest “A” using search
  - use the *prev* pointer, which points to the predecessor of a node ($O(1)$), and back up until hit end of B+/A- grey area
Stacks and Queues

Abstractions that are Wrappers for MyLinkedList
Stacks

- **Stack** has special methods for insertion and deletion, and two others for size
  - push and pop
  - isEmpty, size
- Instead of being able to insert and delete nodes from anywhere in the list, can only add and delete nodes from top of Stack
  - LIFO (Last In, First Out)
- We’ll implement a stack with a linked list
Methods of a Stack

- Add element to top of stack
- Remove element from top of stack
- Returns whether stack has any elements
- Returns number of elements in stack

void push(Type el)

Type pop()

boolean isEmpty()

int size()

Note: public keyword not added here because users of the stack don't care about its internals!
push(1)  
push(2)  
push(3)  
pop()  
push(4)  
pop()  
pop()  
pop()
Stack Constructor

- When generic Stack is instantiated, it contains an empty MyLinkedList

- When using a stack, you will replace Type with the type of object your Stack will hold – enforces homogeneity

- Note the Stack contains a MyLinkedList (“composition” pattern), using the “wrapper” pattern to hide or modify functionality of the contained data structure and to add other methods

```java
public class Stack<Type> {
    private MyLinkedList<Type> list;
    public Stack() {
        this.list = new MyLinkedList<>();
    }
    /* other methods elided */
}
```
Implementing Push

// in the Stack<Type> class ...
public Node<Type> push(Type newData) {
    return this.list.addFirst(newData);
}

● Let’s see the behavior...
● When an element is **pushed**, it is always added to front of list
● Thus the **Stack** delegates to the MLL to implement **push**
Implementing Pop

• Let’s see what this does...

• When popping an element, it is always removed from top of Stack, so call removeFirst on MyLinkedList – again delegation

• removeFirst returns element removed, and Stack in turn returns it

• Remember that the removeFirst method of MyLinkedList first checks to see if list is empty

//in the Stack<Type> class ...
public Type pop() {
    return this.list.removeFirst();
}
isEmpty

- **Stack** will be empty if **list** is empty - delegation

- Returns a boolean that is **true** if **Stack** is empty and **false** otherwise

```java
//in the Stack<Type> class ...
public boolean isEmpty() {
    return this.list.isEmpty();
}
```
size

- Size of \textit{Stack} will be the number of elements that the Linked List contains – delegation

- Size is updated whenever a \textit{Node} is added to or deleted from \textit{list} during \textit{push} and \textit{pop} methods

```java
//in the Stack<Type> class ...
public int size() {
    return this.list.size();
}
```
TopHat Question

Look over the following code:

Stack<HeadTA> myStack = new Stack<>();
myStack.push(htaWill);
myStack.push(htaHarriet);
myStack.pop();
myStack.push(htaUV);
myStack.pop();

Who’s left in the stack?

A. htaWill
B. htaHarriet
C. htaUV
D. none of them!
Example: Execution Stacks

- Each method has an Activation Record (AR)
  - contains an execution pointer to instruction to be executed next in method – code is immutable but local variables are not
  - thus also contains all local variables and parameters of method
    - instance variables stored on the heap (CS33)

- When methods execute and call other methods, Java uses a Stack to keep track of the order of execution
  - when a method calls another method, Java adds activation record of called method to Stack
  - when new method is finished, its AR is removed from Stack, and previous method is continued
  - method could be different or a recursively called clone, when executable pointer points into same immutable code, but different values for variables/parameters
Execution Stacks

A calls B
B calls C
… etc.

When E finishes, its AR is popped. Then D’s AR is popped, etc. Note this handles the tracking of invocations (clones) in recursion automatically.
Stack Trace

- When an exception is thrown in a program, get a long list of methods and line numbers known as a stack trace
  - Exception in thread “main” <exception name>
    at <class>.<method>(<class>.java:<line>)
    ...

- A stack trace prints out all methods currently on execution stack

- If exception is thrown during execution of recursive method, prints all calls to recursive method
Bootstrapping Data Structures

● This implementation of the stack data structure uses a wrapper of a contained MyLinkedList, but user has no knowledge of that

● Could also implement it with an Array or ArrayList
  o Array implementation could be less efficient as we would have to expand our Array as we push more objects onto the Stack.

  o User’s code would not be affected if the implementation of Stack changed (as is true for methods as well, if their semantics isn’t changed) – loose coupling!

● We’ll use the same technique to implement a Queue
What are Queues?

● Similar to stacks, but elements are removed in different order
  o information retrieved in the same order it was stored
  o **FIFO**: First In, First Out (as opposed to stacks, which are **LIFO**: Last In, First Out)

● Examples:
  o standing in line at the checkout counter or movie theater
  o waitlist for TA hours after randomization
Methods of a Queue

• Add element to end of queue
• Remove element from beginning of queue
• Returns whether queue has any elements
• Returns number of elements in queue

void enqueue(Type el)
Type dequeue()
boolean isEmpty()
int size()
Enqueuing and Dequeuing

- Enqueuing: adds a node
- Dequeuing: removes a node

Before Enqueuing

1  2  3
head of queue  tail of queue  student to add

After Enqueuing

1  2  3  4
head of queue  tail of queue

● Enqueuing: adds a node
● Dequeuing: removes a node
Enqueuing and Dequeuing

- Enqueuing: adds a node
- Dequeuing: removes a node

Before Dequeuing

1
2
3
4

head of queue
tail of queue

After Dequeuing

1
dequued student

tail of queue

2
3
4

head of queue
Our Queue

- Again use a wrapper for a contained `MyLinkedList`. As with Stack, we’ll hide most of MLL’s functionality and provide special methods that delegate the actual work to the MLL.

- Contain a `MyLinkedList` within `Queue` class
  - `enqueue` will add to the end of `MyLinkedList`
  - `dequeue` will remove the first element in `MyLinkedList`

```java
public class Queue<Type> {
    private MyLinkedList<Type> list;

    public Queue() {
        this.list = new MyLinkedList<>();
    }

    // Other methods elided
}
```
enqueue

• Just call list’s addLast method – delegation

• This will add node to end of list

```java
public void enqueue(Type newNode) {
    this.list.addLast(newNode);
}
```
dequeue

- We want first node in list
- Use list's `removeFirst` method – delegation

```java
public Type dequeue() {
    return this.list.removeFirst();
}
```

- What if list is empty? There will be nothing to dequeue!
- Our MyLinkedList class's `removeFirst()` method returns `null` in this case, so dequeue does as well
isEmpty() and size()

- As with **Stacks**, very simple methods; just delegate to **MyLinkedList**

  ```java
  public int size() {
      return this.list.size();
  }
  
  public boolean isEmpty() {
      return this.list.isEmpty();
  }
  ```
In order from head to tail, a queue contains the following: bostonRob, russell, parvati, ozzy. We remove each person from the queue by calling dequeue() and then immediately push() each dequeued person onto a stack.

At the end of the process, what is the order of the stack from top to bottom?

A. bostonRob, russell, parvati, ozzy
B. bostonRob, ozzy, russell, parvati
C. ozzy, parvati, russell, bostonRob
D. It's random every time.
Announcements

• Tetris deadlines
  o early handin: Saturday 11/13
  o on-time handin: Monday 11/15
  o late handin: Wednesday 11/17

• Final section (about algorithms!) this week 😞
  o complete the mini-assignment and send to your TAs prior

• DoodleJump check-ins
  o if you still haven’t completed a DJ check-in, sign up for one here
  o sorry for the delay; the number of no-shows and reschedules has led to an increased time burden on our TAs – please show up!
Topics in Socially-Responsible Computing

Forensic Architecture
Using tech to detect human rights violations

- Forensic Architecture — interdisciplinary research agency based at Goldsmiths, University of London
- Use architectural / software techniques to investigate human rights violations around the world
- Typically pulling data from public sources (Twitter, YouTube, etc) and analyzing/compiling it
- Investigations have been used in international courts of law as evidence + exhibited in galleries (the Whitney, Institute of Contemporary Arts – London (ICA), Tate Britain, etc)

3D model of a detention center in Myanmar. Credit: Forensic Architecture
Triple Chaser Investigation

- “Triple Chaser” — specific form of tear gas being used on the US border
- Tear gas sales are private so impossible to figure out when they are being used! Only clear when they show up in photos
- Very few images online!
  - team of artists within Forensic Architecture made 3D models of cannisters and create more images of them to train model
  - used a game engine to simulate the cannisters being worn down
- Trained machine learning algorithm on synthetic images of tear gas cannisters to identify them in future evidence

Credit: Forensic Architecture
Battle of Ilovaisk Investigation

- Town of Ilovaisk — border region between Ukraine and Russia: allegations that Russian army had joined in fight for control of it in 2015

- Forensic Architecture put together a case to be heard in European Court of Human Rights

- Similarly trained classifiers to identify specific tanks that were only used by the Russian army
  - built 3D models of what they looked like
  - brought together evidence of 300 Russian military vehicles in the region

Credit: Forensic Architecture
CS for Social Change

• Lachlan Kermode
  • 1st year Brown PhD in Modern Culture and Media, advisor of Socially Responsible Computing Program
  • Forensic Architecture’s sole Software Researcher before coming to Brown
• CS 1951I: CS for Social Change: option to work on project training machine learning classifiers for ongoing investigation called “Environmental Racism in Death Valley, Louisiana”
More reading that may be of interest!

- “Torture and Detention in Myanmar” — Forensic Architecture
- “Triple-chaser” — Forensic Architecture
- “Battle of Ilovaisk” — Forensic Architecture
- CSCI 1951I: CS for Social Change
- “SOFTWARE RECONNAISANCE: An interview with Lachlan Kermode” — The College Hill Independent