SPRING 2021
MOTION

THURSDAY, MARCH 18TH
7:00 PM EST

TEDxBrownU
TEDXBROWNU.COM
Lecture 16

Linked Lists

C → S → 1 → 5
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

```plaintext
LinkedList<HTA> //note generic

Node<HTA> _head  Node<HTA> _tail

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

null

Node<HTA> _next
HTA _data

Node<HTA> _next
HTA _data

Node<HTA> _next
HTA _data

Node<HTA> _next
HTA _data

Node<HTA> _next
HTA _data

Node<HTA> _next
HTA _data

Aalia

Anna

Gil

Marina

Will
```
When to Use Different Data Structures for Collections (1/3)

- **ArrayLists** get their name because they implement Java-FX’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 (2/3)

• Using LinkedList of Nodes, we can construct higher level abstractions to model collections (e.g., NodeList to parallel ArrayList, as well as Stacks, Queues, etc.)

• Each Node instance holds the data for that element in the list
When to Use Different Data Structures for Collections (3/3)

• 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

## Array
- Indexed (explicit access to \(i^{th}\) item)
- If user moves elements during insertion or deletion, their indices will change correspondingly
- Cannot 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
- Can insert or remove nodes anywhere in the list without data movement through the rest of the list

## 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 instead of Arrays
- Java uses an `Array` as the underlying data structure (and does the data shuffling)

Note: don’t usually access items by index in an `ArrayList`; use search/get!
Linked List Implementations

• Find java.util implementation at: [http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html](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:
  
  o difference between MyLinkedList and Java’s implementation is that Java uses something like our MyLinkedList to build a more advanced data structure that implements List
  
  o while there is overlap, there are also differences in the methods provided, their names, and their return types
  
  o you should 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!
Generic Unsorted Singly Linked List (1/3)

• Constructor initializes instance variables
  o _head and _tail are initially set to null
  o _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
public class MyLinkedList<Type> {
    private Node<Type> _head;
    private Node<Type> _tail;
    private int _size;

    public MyLinkedList() {
        _head = null;
        _tail = null;
        _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)
Generic Unsorted 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)
Generic Unsorted 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() {
    //...
}
```
Generic 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() {
        //...
    }

    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 LL 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()`
Ex: A pile of **Books** (1/2)

- Before implementing **LinkedList** internals, let’s see how to use one to model a simple unorganized pile (i.e., set) of **Books**
  - “user” here is another programmer
- The elements in our pile will be of type **Book**
  - each has title, author(s), date and ISBN number
  - we want a list that can store anything that “is a” **Book**
Ex: A pile of **Books** (2/2)

- The **Book** class combines Authors, Titles and ISBNs (International Standard Book Number)

- In our Linked List, the **Node**’s element will be of type **Book**

```
class Book
{
    String _author
    String _title
    int _isbn

    getAuthor()
    getTitle()
    getISBN()
    ...
}
```
public class Book {
    private String _author;
    private String _title;
    private int _isbn;

    public Book(String author, String title, int isbn) {
        _author = author;
        _title = title;
        _isbn = isbn;
    }

    public int getISBN(){
        return _isbn;
    }

    //other mutator and accessor
    //methods elided
}

Book Class

• Constructor stores author, date and ISBN number of Book as instance variables

• For each property, its get method returns that property’s value
  o ex. getISBN() returns _isbn
PileOfBooks Class

• Contains a MyLinkedList of books as underlying data structure—it’s a “thin wrapper”

• Book specializes generic Type

• Instantiating a MyLinkedList is entirely similar to instantiating an ArrayList

```java
public class PileOfBooks {
    private MyLinkedList<Book> _books;
    
    public PileOfBooks() {
        _books = new MyLinkedList<Book>();
    }
    
    //There could be many more methods here!
    //add and search methods on next slide.
}
```
PileOfBooks Class: add And search

• Since PileOfBooks instantiates a MyLinkedList of books, the class has access to MyLinkedList’s methods

• We can make calls to these in the definitions of PileOfBooks’ own methods
  o PileOfBooks’ methods are “wrappers” over the underlying methods from MyLinkedList

```java
public void addBook(Book myBook) {
    _books.add(myBook);
    //Explanation of add to come!
}

public Node<Book> searchBook(Book myBook) {
    return _books.search(myBook);
    //Explanation of search to come!
}

//There could be many more methods here!
```
Ex: MyLinkedList<Book>

```java
MyLinkedList<Book> _books
  Node<Book> _head
  Node<Book> _tail
  int _size = 4

null

Node<Book>
  _next
  Book _element

Book
  _author = "Roald Dahl"
  _title = "The BFG"
  _isbn = 0142410381

Node<Book>
  _next
  Book _element

Book
  _author = "Jon Krakauer"
  _title = "Into The Wild"
  _isbn = 0385486804

Node<Book>
  _next
  Book _element

Book
  _author = "Neal Stephenson"
  _title = "Cryptonomicon"
  _isbn = 0060512806

Node<Book>
  _next
  Book _element

Book
  _author = "J. R. R. Tolkien"
  _title = "The Hobbit"
  _isbn = 0345339681
```

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!
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 LL’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 LL’s _head to point to newNode
  - if _size is 1, _tail must also point to newNode
  - return newNode

```java
public MyLinkedList() {
    _head = null;
    _tail = null;
    _size = 0;
}

public Node<Type> addFirst(Type el) {
    _size++;
    Node<Type> newNode = new Node<Type>(el);
    newNode.setNext(_head); //previous head
    _head = newNode;
    if (size == 1) {
        _tail = newNode;
    }
    return newNode;
}
```
Constructor and **addFirst** Runtime (2/2)

```java
public MyLinkedList() {
    _head = null;  // 1 op
    _tail = null;  // 1 op
    _size = 0;    // 1 op
}

public Node<Type> addFirst(Type el) {
    ++_size;       // 1 op
    Node<Type> newNode = new Node<Type>(el);    // 1 op
    newNode.setNext(_head);         // 1 op
    _head = newNode;                 // 1 op
    if (_size == 1) {
        _tail = newNode;            // 1 op
    }
    return newNode;                  // 1 op
}
```

→ **addFirst** *(Type el)* is **O(1)**

→ **constructor** is **O(1)**
**addLast Method (1/2)**

- LL’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

```
Node<Type>  // _head
        ^
        |  
Node<Type>  // _tail
        ^
        |  
Node<Type>  // _tail
        ^
        |  
null
```

```
new node
```

Andries van Dam © 2021 3/18/21
addLast Method (2/2)

• **Edge Case**
  o if list is empty, update the \_head and \_tail variables to the newNode

• **General Case**
  o update \_next of current last Node (to which \_tail is pointing – “update \_tail’s \_next”) to new last Node
  o update \_tail to that new last Node
  o new Node’s \_next variable already points to \texttt{null}

```java
public Node<Type> addLast(Type el) {
    Node<Type> newNode = new Node<Type>(el);
    if (_size == 0) {
        _head = newNode;
        _tail = newNode;
    } else {
        _tail.setNext(newNode);
        _tail = newNode;
    }
    _size++;
    return newNode;
}
```
public Node<Type> addLast(Type el) {
    Node<Type> newNode = new Node<Type>(el) // 1 op
    if (_size == 0) {
        _head = newNode; // 1 op
        _tail = newNode; // 1 op
    } else{
        _tail.setNext(newNode); // 1 op
        _tail = newNode; // 1 op
    }
    _size++; // 1 op
    return newNode; // 1 op
}

→ addLast(Type el) is O(1)
size and isEmpty Methods

```java
public int size() {
    return _size;
}

public boolean isEmpty() {
    return _size == 0;
}
```

isEmpty() would return false
size and isEmpty Runtime

```java
public int size() {
    return _size; // 1 op
}

public boolean isEmpty() {
    return _size == 0; // 1 op
}
```

→ 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 (_size == 0) {
        System.out.println("List is empty");
        return null;
    }

    Type removed = _head.getElement();
    _head = _head.getNext();
    _size--;
    if (_size == 0) {
        _tail = null;
    }
    return removed;
}
```
public Type removeFirst() {
    if (_size == 0) {
        // 1 op
        System.out.println("List is empty");  // 1 op
        return null;                         // 1 op
    }

    Type removed = _head.getElement();    // 1 op
    _head = _head.getNext();              // 1 op
    _size--;                             // 1 op
    if (_size == 0) {                    // 1 op
        _tail = null;                     // 1 op
    }
    return removed;                      // 1 op
}

→ removeFirst() is O(1)
Review: Accessing Nodes Via Pointers

head.getNext();

- This does not get the _next field of _head, which doesn’t have such a field, being just a pointer
- Instead, it is the temporary name of a node and getNext() gets that node’s _next field, a pointer to the second node
- What does _tail.getNext() produce?
- What does _tail.getElement() produce?
- note we can access a variable by its unique name, its index, its contents, or here, via a pointer
Lecture Question

Given a Linked List of Nodes,

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

where \_head\ points to node \( A \), what is \_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
    - very inefficient—stay tuned
  - Update `_tail`
  - Last `Node` is thereby garbage-collected!
removeLast Method

- 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

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

Given that _animals is a Singly Linked List of n animals already in the list, what would be printed to the console?

private void testListMethods() {
    _animals.addFirst(new Cat());
    _animals.addFirst(new Fish());
    _animals.addLast(new Dog());
    if(!_animals.isEmpty()) {
        System.out.println("Not Empty");
    }
    System.out.println(_animals.removeLast());
}

A. “Not Empty”, “Dog”
B. no print, “Cat”
C. “Not Empty”, “Fish”
D. no print, “Dog”
Lecture Question

Given that \texttt{animals} is a Singly Linked List of \textit{n} animals, what is \texttt{node} pointing to?

\begin{verbatim}
curr = _head;
prev = null;
while (curr.getNext().getNext() != null) {
    prev = curr;
    curr = curr.getNext();
}
node = curr.getNext();
\end{verbatim}

A. Nothing useful, throws a \texttt{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
Lecture Question

Given that `_animals` is a Singly Linked List of `n` animals, what will this code fragment do?

```java
//prev and curr initialized to null and _head, respectively
while (curr.getNext() != null) {
    prev = curr;
    curr = curr.getNext();
}
removed = prev.getElement();
System.out.println(removed);
```

A. Nothing useful, throws a `NullPointerException`
B. Prints last animal in list
C. Prints next to last animal in list
D. Prints whole list
search Method

- Let’s think back to our pile of *Books* example – what if we want to find a certain *Book* in the pile of *Books*? What if we want to search by ISBN, author, or title?

- Must compare each *Book* with one we are looking for
  - but in order to do this, we first need a way to check for the equality of two elements!
  - brute force: we can do somewhat better by sorting, much better with sorted arrays, binary trees – stay tuned!
Java’s `Comparable<Type>` interface (1/3)

- Previously we used `==` to check if two things are equal
  - this only works correctly for primitive data types (e.g., `int`), or when we are comparing two variables referencing the exact same object
  - to compare `Strings`, need a different way to compare things

- We can implement the `Comparable<Type>` generic interface provided by Java

- Must define `compareTo` method, which returns an `int`

- Why don’t we just use `==`, even when using something like ISBN, which is an `int`?
  - can treat ISBNs as `ints` and compare them directly, but more generally we implement the `Comparable<Type>` interface, which could easily accommodate comparing `Strings`, such as author or title, or any other property
Java’s `Comparable<Type>` interface (2/3)

- The `Comparable<Type>` interface is specialized (think of it as parameterized) using generics
  
  ```java
  public interface Comparable<Type> {
    public int compareTo(Type toCompare);
  }
  ```

- Call `compareTo` on a variable of same type as specified in implementator of interface (`Book`, in our case)
  - `currentBook.compareTo(bookToFind);`
  - pseudo-code: `currentBook.compareTo(bookToFind)`
Java’s `Comparable<Type>` interface (3/3)

- `compareTo` method must return an `int`
  - **negative** if element on which `compareTo` is called is less than element passed in as the parameter of the search
  - 0 if element is `equal` to element passed in
  - **positive** if element is `greater` than element passed in
  - sign of `int` returned is all-important, magnitude is not and is implementation dependent

- `compareTo` not only used for numerical comparisons—it could be used for alphabetical or geometric comparisons as well—depends on how you implement `compareTo`
“Comparable” Book Class

• Recall format for `compareTo`:
  o `elementA.compareTo(elementB)`

• Book class now implements `Comparable<Book>`
  o this means we can compare books, using `bookA.compareTo(bookB)`

• `compareTo` is defined according to these specifications
  o returns number that is `<0`, `0` or `>0`, depending on the ISBN numbers
  o `<0` if stored `_isbn` `< toCompare parm

```java
public class Book implements Comparable<Book> {
    public Book(String author, String title, int isbn){
        //variable initializations elided
    }

    public int getISBN(){
        return _isbn;
    }

    //other methods elided

    //compare isbn of book passed in to stored one
    @Override
    public int compareTo(Book toCompare){
        return (_isbn - toCompare.getISBN());
    }
}
```
“Comparable” Singly Linked List

- Using keyword `extends` in this way ensures that `Type implements Comparable<Type>`
  - note nested `<`, and that `extends` is used differently from inheritance notation
  - nested `<` to show it modifies `Type` and not the class

- All elements stored in `MyLinkedList` must now have `compareTo` method for `Type`; thus restricts generic

```java
public class MyLinkedList<Type extends Comparable<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) {
        //...
    }
}
```

In our example use Book as `Type`

Just as classes “extend” other classes, interfaces can “extend” other interfaces, thus generic `Type` must “extend” the interface
search Method for MyLinkedList (brute force)

- 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 = _head;
    while (curr != null) {
        if (curr.getElement().compareTo(el) == 0) {
            return curr;
        }
        curr = curr.getNext(); //bop pointer
    }
    return null; //got to end of list w/o finding
}
```
public Node<Type> search(Type el) {
    Node<Type> curr = _head;  // 1 op
    while (curr != null) {    // n ops
        if (curr.getElement().compareTo(el) == 0) {  // 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
  ○ 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.

Note: caller of remove can find out if item was successfully found (and removed) by testing for != null
public Type remove(Type itemToRemove){
    if (this.isEmpty()) {
        System.out.println("List is empty");
        return null;
    }
    if (itemToRemove.compareTo(_head.getElement()) == 0) {
        return this.removeFirst();
    }
    if (itemToRemove.compareTo(_tail.getElement()) == 0) {
        return this.removeLast();
    }
    Node<Type> curr = _head.getNext();
    Node<Type> prev = _head;
    while (curr != null) {
        if (itemToRemove.compareTo(curr.getElement()) == 0) {
            prev.setNext(curr.getNext());
            _size--;
            return curr.getElement();
        }
        prev = curr;
        curr = curr.getNext();
    }
    return null;
}

→ remove(Type itemToRemove) is O(n)
Lecture 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, and that prev points to curr's predecessor, what will this code fragment do?

removed = curr.getElement();
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
Ex: A sorted bookshelf

• Faster to find (and remove!) books in a sorted bookshelf

• Use a sorted linked list
  o makes several of our methods somewhat more efficient:
    ▪ search
    ▪ insert
    ▪ delete

• Sort in increasing order
  o maintain sort order when inserting
Example: `MySortedLinkedList<Book>`

- `MySortedLinkedList<Book> _books`
  - `Node<Book> _head`
  - `Node<Book> _tail`
  - `int _size = 4`

- `Node<Book>`
  - `Node<Book> _next`
  - `Book _element`

- `Book`
  - `_author = "Neal Stephenson"`
  - `_title = "Cryptonomicon"`
  - `_isbn = 0060512806`

- `Node<Book>`
  - `Node<Book> _next`
  - `Book _element`

- `Book`
  - `_author = "Roald Dahl"`
  - `_title = "The BFG"`
  - `_isbn = 0142410381`

- `Node<Book>`
  - `Node<Book> _next`
  - `Book _element`

- `Book`
  - `_author = "J. R. R. Tolkien"`
  - `_title = "The Hobbit"`
  - `_isbn = 0345339681`

- `Node<Book>`
  - `Node<Book> _next`
  - `Book _element`

- `Book`
  - `_author = "Jon Krakauer"`
  - `_title = "Into Thin Air"`
  - `_isbn = 0385486804`
Generic Sorted Singly Linked List

- Slightly different set of methods
  - addFirst and addLast replaced by general insert

- Many methods whose signatures are identical to those of the unsorted list will have different implementation because of more efficient loop termination used in search and remove
search Method [For Sorted Linked Lists]

- Must iterate through list until `toFind` is found – only need `curr`

- Compare `toFind` to `curr`’s element
  - if equal, we’re done!

- If `curr`’s element is greater than `toFind`, stop search
  - any following Node’s elements will also be greater since list is sorted
  - note: order of operands dictates sign of test–be careful!

```java
public Node<Type> search(Type toFind){
    Node<Type> curr = _head;
    while (curr != null) {
        // have we found it?
        if (curr.getElement().compareTo(toFind) == 0) {
            return curr;
        }
        // are we past it? Curr’s element > toFind
        else if (curr.getElement().compareTo(toFind) > 0){
            return null;
        }
        // haven’t found it, bop the curr
        curr = curr.getNext();
    }
    return null;
}
```
search Runtime [For Sorted Linked Lists]

```java
public Node<Type> search(Type toFind) {
    Node<Type> curr = _head; // 1 op
    while (curr != null) { // n ops
        if (curr.getElement().compareTo(toFind) == 0) { // 1 op
            return curr; // 1 op
        } else if (curr.getElement().compareTo(toFind) > 0) { // 1 op
            return null; // 1 op
        }
        curr = curr.getNext(); // 1 op
    }
    return null; // 1 op
}
```

While the `else if` statement will typically improve performance because searches don’t usually have to search to the bitter end, it does not affect Big-O run time! (Big-O describes the worst case!)

→ search(Type toFind) is O(n)
What Did Sorting Buy Us?

• Search still $O(n)$—not better worst-case performance, but better for normal cases because can usually stop earlier for items not in the list

• This comes at the cost of having to maintain sort order, i.e., by having to insert “in the right place” (next slide)

• So if an algorithm does a lot of searching compared to insertion and deletion, this efficiency would pay off; it even beats a sorted array because the linear search loop is $O(n)$ for both sorted linked lists and arrays, but inserting/deleting is $O(1)$ for linked lists and $O(n)$ for arrays because of (worst case) data movement
  o we can do much better than $O(n)$ for search: Binary Search in a sorted array and a Linked List as a Binary Tree makes search $O(\log n)$ (stay tuned for Lecture on Trees)

• Conversely, if an algorithm does a lot of adding and deleting compared to searching, sorting wouldn’t pay off since the algorithm couldn’t use the simple $O(1)$ `insertFirst` or `insertLast`...
Comparing Variable Values

• The equality operator “==” compares references
  o checks if two objects point to the same place in memory
  o good for comparing primitive types (integers, doubles, booleans, etc..)

• The “.equals()” method compares content
  o checks if two objects have the same values
  o most Java classes override the .equals() method to be able to compare instances of themselves

• We will use .equals() for Strings, but for more general comparisons use the Comparable<Type> interface
Comparisons

```java
int x = 5;
int y = 5;
x == y;
true
```

Primitive types assigned to a variable are contained in that variable, hence 
“==” compares the contents

```java
String s1 = new String("CS15 is so fun!");
String s2 = new String("CS15 is so fun!");
s1 == s2
false
```

Because “==” compares references for non-primitives, 
therefore use `s1.equals(s2);`
**insert method**

- Once again, 2 phases:
  - find right place to insert with a search loop, keeping track of current and previous nodes
  - break the chain to insert
- Unlike insertion into unsorted linked list, there is one correct spot in sorted list for new node
- End iteration if current node’s value is greater than new node’s value—break the chain and insert there!
- Update `_next` pointers of new node and previous node
insert Method
[for Sorted Linked Lists]

• Edge case
  o if list is empty, all we have to do is reset _head/_tail

• General case
  o iterate over lists until curr’s element is greater than newItem
  o need loop’s prev, so we can re-link list to integrate the new node
  o or if not found, special case

```java
public Node<Type> insert(Type newItem){
    Node<Type> toAdd = new Node<Type>(newItem); //node for newItem
    if (this.isEmpty()) {
        _head = toAdd;
        _tail = toAdd;
        return _head;
    }
    else {
        Node<Type> curr = _head;
        Node<Type> prev = null;
        while (curr != null) {
            //pointer-chasing iterator
            if (curr.getElement().compareTo(newItem) < 0) {
                prev = curr;
                curr = curr.getNext();
            }
            else { //found the spot! Update two pointers
                toAdd.setNext(curr); //always do this, but...
                if (prev != null) {
                    //prev is null only at front
                    prev.setNext(toAdd);
                } else {
                    _head = toAdd; //insert at front of list
                }
                _size++
                return toAdd;
            }
            prev.setNext(toAdd); //not found, insert node at end
        }
        _tail = toAdd;
        _size++;
        return toAdd;
    }
}
```

- Edge case
  - if list is empty, all we have to do is reset _head/_tail

- General case
  - iterate over lists until curr’s element is greater than newItem
  - need loop’s prev, so we can re-link list to integrate the new node
  - or if not found, special case
public Node<Type> insert(Type newItem) {
    Node<Type> toAdd = new Node<Type>(newItem); // 1 op
    if (this.isEmpty()) { // 1 op
        _head = toAdd; // 1 op
        _tail = toAdd; // 1 op
        return _head;
    } else { // 1 op
        Node<Type> curr = _head; // 1 op
        Node<Type> prev = null; // 1 op
        while (curr != null) { //pointer-chasing iterator // n ops
            if (curr.getElement().compareTo(newItem) < 0) { // 1 op
                prev = curr; // 1 op
                curr = curr.getNext(); // 1 op
            } else { // 1 op
                toAdd.setNext(curr); // 1 op
                if (prev != null) { // 1 op
                    prev.setNext(toAdd); // 1 op
                } else { // 1 op
                    _head = toAdd; // 1 op
                }
                _size++; // 1 op
            }
            prev = curr; // 1 op
        }
        return toAdd; // 1 op
    }
}

→ insert(Type newItem) is O(n)

Note: O(n) is to find the point of insertion
**remove Method**  
[for Sorted Linked Lists]  

- Loop through nodes until an `_element` matches `itemToRemove` is found  
  - since list is sorted, we can end loop early—stay tuned  

- Re-link predecessor of node (again using a previous node) to successor of node (its `_next` reference)  

- With no more reference to node, it is garbage collected at the termination of the method
**remove Method [for Sorted Linked Lists]**

- **Edge Case(s)**
  - if list is empty, return null
  - if `itemToRemove` is the `_head`/`_tail`, use same code as `removeFirst/removeLast` in `MyLinkedList`

- **General case**
  - iterate over list until either:
    - `itemToRemove` is found (equals `curr`'s element), so reset `next` pointer in `prev` node and return found item
    - or if `curr`'s element is greater than `itemToRemove`, it can’t be in the list, hence return null – early exit!
    - or we reach end of list, so return null

```java
public Type remove(Type itemToRemove){
    if (this.isEmpty()) {
        return null;
    }
    if (itemToRemove.compareTo(_head.getElement()) == 0) {
        //elided; same as MyLinkedList’s removeFirst code
    }
    if (itemToRemove.compareTo(_tail.getElement()) == 0) {
        //elided; same as MyLinkedLists’s removeLast code
    }
    Node<Type> curr = _head.getNext();
    Node<Type> prev = _head;
    while (curr != null) {
        if (curr.getElement().compareTo(itemToRemove) == 0) {
            prev.setNext(curr.getNext()); //jump over node
            _size--;
            return curr.getElement(); //curr points to found
        }
        else if (curr.getElement().compareTo(itemToRemove) > 0) {
            return null;
        }
        prev = curr; //bop pointers, iterate
        curr = curr.getNext();
    }
    return null; // End of list, w/o finding it
}
```
public Type remove(Type itemToRemove){
    if (this.isEmpty()) {
        return null;
    }
    if (itemToRemove.compareTo(_head.getElement()) == 0) { // elided; same as MyLinkedLists's removeFirst code
        // 1 op
        return null; // 1 op
    }
    if (itemToRemove.compareTo(_tail.getElement()) == 0) { // elided; same as MyLinkedLists's removeLast code
        // 1 op
        // O(1)
    }
    Node<Type> curr = _head.getNext(); // 1 op
    Node<Type> prev = _head; // 1 op
    while (curr != null) { // n ops
        if (curr.getElement().compareTo(itemToRemove) == 0) { // 1 op
            prev.setNext(curr.getNext()); // 1 op
            _size--; // 1 op
            return curr.getElement(); // 1 op
        } else if (curr.getElement().compareTo(itemToRemove) > 0) { // 1 op
            return null; // 1 op
        }
        prev = curr; // 1 op
        curr = curr.getNext(); // 1 op
    }
    return null; // 1 op
}
Lecture Question

How do sorted and unsorted Linked Lists differ?

A. Sorted linked lists are somewhat more efficient than unsorted linked lists because they are more quickly searched, but are far less efficient for insertion and deletion.

B. Sorted linked lists are less efficient than unsorted linked lists because they are more slowly searched, but are more efficient for insertion and deletion.

C. Sorted linked lists are more efficient than unsorted linked lists because they are more quickly searched and are more efficient for insertion and deletion.

D. Sorted linked lists are somewhat more efficient than unsorted linked lists, but they are more slowly searched and are far less efficient for insertion and deletion.
Unsorted vs. Sorted Singly Linked List (1/3)

• Worst Case and Best Case are the same for sorted and unsorted linked lists, as is the average case if the item is in the list (n/2 on average)

• Runtime advantage of sorted list comes when object is not in list (and at cost of more expensive insert/delete)
  
  o can stop looping if we find an element larger than what we are searching for

  o note: while searching may be on average twice as efficient for this case, big O is still O(n) for worst and average cases
Unsorted vs. Sorted Singly Linked List (2/3)

- LISP, the original AI programming language, and its various dialects, e.g., Scheme, use linked lists as the basic data structure.

- Recursive definition of list: a list member is an atomic element or a sublist.

Source: http://gajon.org/trees-linked-lists-common-lisp/
Example using a sorted singly linked list: dynamic storage allocation using the “free list”

- Need a dynamic collection, and single pointers are memory efficient
- Can search until we find the smallest chunk of memory large enough for what we need
Efficiency of Arrays vs. Linked Lists in Modern Computers (1/2)

• Our discussion has not taken into account memory hierarchies
  o (superfast index registers, not suitable for storing more than frequently-used vars)
  o at the lowest level is the cache, which is very fast but limited in size (may have multiple levels, e.g., L1 at 32KB, L2 at 256KB and L3 at 2MB for Intel’s i7)
  o next is system memory (aka CPU memory, RAM) – much larger, much slower
  o next is peripheral memory such as flash drives, disk drives, etc. – much slower
  o could even consider cloud storage as a 4th level

• Arrays can be stored and accessed super-efficiently as a whole or in chunks in cache, but when a next element isn’t in cache, a cache “miss” occurs and the next chunk has to be brought in from RAM, overwriting the previous chunk

From Tom Doeppner’s CS33 Lecture:
—... the L1 cache is like grabbing a piece of paper from your desk (3 seconds)
—... the L2 cache is picking up a book from a nearby shelf (14 seconds)
—... main system memory is taking a 4-minute walk down the hall to talk to a friend
—... a hard drive is like leaving the building to roam the earth for one year and three months
http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait/
1. To get her work done, the CPU needs access to lots and lots of data. This data resides in different components that store different quantities of information and require varying amounts of time to access.

2. The CPU registers (index card).

3. The L1 cache (1 page @ 1 inch).

4. The L2 cache (31 pages @ 3 inches).

5. The L3 cache (125 pages @ 16 inches).

6. Main memory (200 books @ 5 feet).

7. Disk/virtual memory (60,000 books @ 950 miles). Roughly a science library in St. Louis.

8. Tertiary storage (100 million books @ 1 million miles). Roughly the Library of Congress on the moon.


Scale: 1 inch per CPU cycle (about a nanosecond) and one page of text per 32K bytes of storage.
Efficiency of Arrays vs. Linked Lists in Modern Computers (2/2)

• Virtual memory is an even older technique where “pages” of program and/or data are stored in peripheral memory but your program doesn’t have to deal with that – page misses/faults cause hardware and the OS to find the proper page in peripheral memory and bring it in.

• Linked list traversal may be much more inefficient than simulating dynamic collections with arrays if data movement or copying arrays into larger arrays can largely be done in cache, because page faults are hugely time-consuming (millisecs vs. microseconds).

• Understanding performance of algorithms and data structures for actual data is complex, hardware-dependent, and is different from Big-O asymptotic analysis – they both have their place.
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\(^1\), need to go in the opposite direction
  o 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”?

\(^1\) like Singly Linked Lists, Doubly Linked Lists can be sorted or unsorted. We only discuss the sorted version here
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
  o 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:
  o find student with the lowest “A” using search
  o 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
Remove method [For Sorted Doubly Linked List]

- This is *pseudo-code* for a `remove` method for a Sorted Doubly Linked List
  - note dot-notation for variables – a common practice for pseudo-code
  - this is incomplete: does not deal with edge cases (element not found, last in list, etc.)

- Other methods are analogous

```plaintext
remove(Type t)
Node n = search(t)
set n._prev’s _next variable to n._next
set n._next’s _prev variable to n._prev
//n is now garbage collected
return n’s _element
```

```
Node<Type>
Node<Type> _prev
Node<Type> _next
Type _element
```

```
Node<Type> _prev
Node<Type> _next
Type _element
garbage-collected
```

```
Node<Type> _prev
Node<Type> _next
Type _element
```

...
public class Node<Type> {
    private Type _element;
    private Node<Type> _next;
    private Node<Type> _prev; // New!

    public Node(Type element) {
        _element = element;
        _next = null;
        _prev = null;
    }

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

    public Node<Type> getNext() {
        return _next;
    }

    /* standard mutator and accessor method
    for _element are elided.*/

    public void setPrev(Node<Type> prev) {
        _prev = prev;
    }

    public Node<Type> getPrev() {
        return _prev;
    }
}

Node [For Doubly Linked Lists]
Lecture Question

What methods do Doubly Linked Lists Nodes have that Singly Linked Lists Nodes do not have?

A. getNext(), setNext()
B. getElement(), setElement()
C. getPrev(), setPrev()
D. getLocation(), setLocation()
## [Sorted] Singly Linked vs Doubly Linked

<table>
<thead>
<tr>
<th>Single</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>• To look for a node <em>(search, insert, delete)</em> <strong>must</strong> start at the beginning and go to the next node <em>n</em> times → no random access!</td>
<td>• Same functionality as Single</td>
</tr>
</tbody>
</table>
| • Dynamically grow/shrink | • Has a previous pointer `_prev`
| • Can *insert* or *remove* nodes anywhere in the list |   o for `removeLast`, don’t need to loop through entire list to find predecessor
|                     |   o don’t have to maintain separate `_prev` pointer for `search`

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Traverse in both directions

• Requires more memory/storage (more pointers)
Linked List Exercises
How To Build A Node List

• Now that we have a building block, there are a number of methods we can implement to make a higher-level NodeList that implements Java’s List interface (like ArrayList does)
  - note: List interface is very general...

• Main addition List mandates is to support indexing into the NodeList. Let’s write one of the simpler ones:
  - get(int i) method that returns element (Type) at that index
**search** Private Helper Method

- First, define a `search` helper method to return node at a particular index

- Want to use this helper method in the class, but don’t want to expose found nodes publicly; that would violate encapsulation - make helper `private`

- If a provided index is out of bounds, return `null` (print line is an optional error message)

- Otherwise, iterate through list until node at desired index is reached and return that node

```java
public class NodeList<Type> {
    //constructor elided
    private Node<Type> search(int i) {
        if(i < 0 || i >= _size) {
            System.out.println("Invalid index");
            return null;
        }
        Node<Type> curr = _head;
        //for loop stops at i; pointer-chase to i
        for (int counter = 0; counter < i; counter++) {
            curr = curr.getNext();
        }
        return curr;
    }
}
```
private Node<Type> search(int i) {
    if (i >= _size || i < 0) { // 1 op
        System.out.println(“Invalid index”); // 1 op
        return null; // 1 op
    }

    Node<Type> curr = _head; // 1 op
    for (int counter = 0; counter < i; counter++) { // n ops
        curr = curr.getNext(); // 1 op
    }

    return curr; // 1 op
}

→ search(int i) is O(n)
Public Wrapper Method

• Write the publicly accessible wrapper code for the NodeList’s `get` method

  o this shows a very common pattern of “thin wrappers” over private code

```java
//inside NodeList
public Type get(int i) {
    return this.search(i).getElement();
}
```
An Exercise (“CS16-Style”, common job interview question)

- Write a method that reverses the order of a `MyLinkedList`
Solution A

- If list is empty or has 1 node, return list
- Otherwise, create a new list of same type as input list
- Iterate through input list, removing first element each time and adding it as first element of new list

```java
public MyLinkedList<Type> reverse(MyLinkedList<Type> toReverse) {
    if (toReverse.size() < 2) {
        return toReverse;
    }
    MyLinkedList<Type> newList = new MyLinkedList<Type>();
    int origSize = toReverse.size();
    while (newList.size() < origSize) {
        newList.addFirst(toReverse.removeFirst());
    }
    return newList;
}
```
Solution B (1/2)

• Is there a better way?

• First algorithm reversed in $O(n)$ time
  o but it wasn’t “in-place” – (had to create a new list)

• Can write a method within MyLinkedList that reverses itself without creating new nodes
  o still $O(n)$ but in-place and therefore more efficient
Solution B (2/2)

- Keep track of previous, current, and next node
- While current node isn’t null, iterate through nodes, resetting node pointers in reverse
- In doing so, must be careful not to delete any references further on in the list
- Finally, set the _head pointer to what had been the last node (held in the prev variable)
- If the list is empty curr will be null, so the loop will never begin and _head will continue to point to null

```java
public void reverse() {
    Node<Type> prev = null;
    Node<Type> curr = _head;
    Node<Type> next = null;
    _tail = _head; // set tail to head

    while (curr != null) {
        next = curr.getNext();
        curr.setNext(prev);
        prev = curr;
        curr = next;
    }

    _head = prev;
}
```
Solution B Walkthrough (1/15)

prev = null
next = null
prev = null

_A_

curr

_B_

curr

_C_

 curr

next = null
prev = null
Solution B Walkthrough (2/15)

prev = null
curr
next = null
_tail = _head;

A --- B --- C
Solution B Walkthrough (3/15)

while (curr != null) {
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

prev = null
Solution B Walkthrough (4/15)

prev = null

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (5/15)

```java
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (7/15)

```java
while (curr != null) {
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (9/15)

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

A

head

B

curr

prev

C

null

_tail

_while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (10/15)

```java
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
Solution B Walkthrough (11/15)

```java
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
next = null
```
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

next = null
Solution B Walkthrough (13/15)

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (14/15)

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

curr = null
next = null
Solution B Walkthrough (15/15)

```
_head = prev;
```

A _tail

null

B

C

_head

prev

curr = null

next = null
Announcements

- DoodleJump late deadline Friday 3/19!
- Tetris is out!
  - Early hand-in Monday 3/29
  - On-time hand-in Wednesday 3/31
  - Late hand-in Friday 4/2
- Reminder: you cannot use your late days on the Final Project, so the last project to apply them to is Tetris
IT in the News

ft. Socially Responsible Computing!
Labor in the Tech Industry

• Vast disparity in working conditions of highly paid employees (ex. software engineers) and underpaid, overworked employees (ex. assembly workers)

Apple and Google named in US lawsuit over Congolese child cobalt mining deaths

—The Guardian (December 2019)

Workers at the Google office in Dublin, Ireland

Workers mining cobalt for lithium battery-powered smartphones in DRC, Africa
Content Warning: Suicide

• “Foxconn Suicides” → series of suicides linked to low pay and meticulous, grueling workplace conditions at Foxconn electronics manufacturing plant in Shenzhen, China

• Foxconn is a **major manufacturer** of Apple products

Suicide Nets at Foxconn factory
Source: Wall Street Journal
Labor in the Tech Industry

Gig economy + subcontracting → tech companies avoid recognizing workers as employees

- **Child labor in DRC cobalt mines**
  - Tech companies filed to have the suit dismissed, arguing they are **not responsible**
  - Since they don’t own cobalt mines and use **intermediary suppliers**, specific sites cannot be identified as sources

- **Proposition 22**
  - 2020 CA initiative permitting companies to treat ride-share and delivery drivers as **independent contractors**, not employees
  - 5 major gig companies spent over **$200 million** to pass Prop 22!
    - *why:* employees are entitled to fair wages and labor protections; contractors are not → Uber, Lyft etc. save $$$

- **Content Moderation**
  - Facebook **outsources** content moderation to India, Philippines
“The activity of content moderation just doesn’t fit into Silicon Valley’s self-image. Certain types of activities are very highly valued and glamorized—product innovation, clever marketing, engineering … the nitty-gritty world of content moderation doesn’t fit into that”

—Paul Barrett, director of NYU’s Center for Business and Human Rights
Labor movements within tech

• Tech industry is hostile towards organized labor
  o Google has fired workers for “organizing other employees”, today only 700 out of 260,000 employees of Alphabet are unionized
  o Amazon has fought unionization attempts for years, most recently in Bessemer, Alabama where workers are voting on whether to form the first Amazon union in the U.S.

• Why Unions?
  o Demands for policies in support for women and people of color have led to changes at tech companies
    • Ex. Employees spoke up against firings of Timnit Gebru and Margaret Mitchell
  o Employees can advocate companies to monitor abusive, racist and sexist content on their platforms
    • Ex. Twitter banned Trump after 350 employees wrote a letter demanding an investigation