Lecture 16
Linked Lists
When to Use Different Data Structures for Collections (1/2)

• Many data structures store a collection of elements

• ArrayLists are called that because they implement Java-FX’s List interface (defined soon) and are implemented using Arrays

• We can define a building block called LinkedList, which can be an alternative to ArrayLists in some cases by avoiding data movement for insertion and deletion
When to Use Different Data Structures for Collections (2/2)

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

• 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
What is a **Linked List**? (1/2)

- A collection of linked nodes that form a sequence of elements
  - as with *Arrays* and *ArrayLists*, it can represent an unordered set or an ordered sequence of your data
  - algorithms can take advantage of fact that elements are stored sequentially, or not
- A **Linked List** holds reference to its first node (*head*) and its last node (*tail*)
What is a **Linked List?** (2/2)

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

LinkedList<HTA>

Node<HTA> _head    Node<HTA> _tail

Node<HTA>
Node<HTA> _next
HTA _data

null

Node<HTA>
Node<HTA> _next
HTA _data

Node<HTA>
Node<HTA> _next
HTA _data

Node<HTA>
Node<HTA> _next
HTA _data

Node<HTA>
Node<HTA> _next
HTA _data

Divya
Dan
Sophia
Emily

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

**Array**
- indexed (explicit access to n\(^{th}\) item)
- if user moves elements during insertion or deletion, their indices will change correspondingly
- cannot change size dynamically

**ArrayList**
- indexed (explicit access to n\(^{th}\) item)
- indices of successor items automatically updated following an inserted or deleted item
- can grow/shrink dynamically
- Java uses an Array as the underlying data structure

**LinkedList**
- **not** indexed – in order 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
- use nodes instead of Arrays
- can insert or remove in the middle of list without data movement through the rest of the list

Note: don’t usually access items by index in an AL, you 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, define 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 `List`
  - while there is overlap, there are also differences in the methods provided, their names, and their return types

- `MyLinkedList` is a building block for more specialized versions: `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
Generic Unsorted Singly Linked List (1/3)

• Constructor initializes instance variables
  - _head and _tail are initially set to null

• AddFirst creates first Node and updates _head to reference it

• addLast appends a Node to the ends 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() {
        //...
    }

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

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

    // more on next slide
```

Generic type parameter
Generic Unsorted Singly Linked List (2/3)

- **removeFirst** removes first *Node* and returns element
- **removeLast** removes last *Node* and returns element
- **remove** removes the *Node* containing the element *e1* and returns it

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

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

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

// still more on next slide
```
Generic Unsorted Singly Linked List (3/3)

- **search** finds and returns the Node containing **e1** (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 e1) {
    //...
}
public int size() {
    //...
}
public boolean isEmpty() {
    //...
}
public Node<Type> getHead() {
    //...
}
public Node<Type> getTail() {
    //...
}
```
Generic Singly Linked List Overview

```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 el) {
        //...
    }

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

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

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

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

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

Andries van Dam 201611/01/16
The **Node** Class

- Constructor initializes instance variables `_element` and `_next`
- It’s 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) {
        _next = null;
        _element = element;
    }

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

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

    public Type getElement() {
        return _element;
    }

    public void setElement(Type element) {
        _element = element;
    }
}
```
Ex: A pile of Books (1/2)

• Let’s use a Linked List to model a simple unorganized pile (i.e., set) of Books

• The elements in our pile will be of type Book
  o have titles, authors, dates and ISBN numbers
  o 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

• In our Linked List, the Node’s element will be of type 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>
</tbody>
</table>

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

Andries van Dam © 201611/01/16
**Book Class**

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

- For each property, its `get` returns that property’s value
  - `getISBN()` returns `_isbn`

```java
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 accessor methods elided
}
```
PileOfBooks Class

• Contains a MyLinkedList of books as underlying data structure – it’s a thin wrapper

• Book is our 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!
}
```
**Ex: MyLinkedList<Book>**

MyLinkedList<Book> _books

Node<Book> _head
Node<Book> _tail
int _size = 4

null

Node<Book>
Node<Book>
Node<Book>
Node<Book>

Book
Book
Book
Book

_author = “Roald Dahl”  
_title = “The BFG”  
isbn = 0142410381

_author = “Jon Krakauer”  
_title = “Into The Wild”  
isbn = 0385486804

_author = “Neal Stephenson”  
_title = “Cryptonomicon”  
isbn = 0060512806

_author = “J. R. R. Tolkien”  
_title = “The Hobbit”  
isbn = 0345339681  17/98

Note: The LinkedList is the instance with _head and _tail in it and the set of linked Nodes distributed in memory.
addFirst – empty list

• If list is empty, _head and _tail will be null
  o show only the list pointers

• Create a new Node<Type>

• Update new node’s _next variable to null, which is where current _head points 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

- Create a new Node
- Update its _next variable to current _head (in this case, some previously added Node that headed list)
- Update _head variable to the new Node
Constructor and **addFirst** Method (1/2)

- **Constructor**
  - initialize instance variables

- **addFirst** method
  - increment **size** by 1
  - create new **Node** (constructor stores **el** in **_element**, null in **_next**)
  - update **newNode**'s **_next** to first **Node** (pointed to by **_head**)
  - update **_head** to point to **newNode**
  - if **size** is 1, **_tail** also points 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);
    _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++;
    Node<Type> newNode = new Node<Type>(el);
    newNode.setNext(_head);
    _head = newNode;
    if (size == 1) {
        _tail = newNode;
    }
    return newNode;
}
```

→ addFirst(Type el) is O(1)
addLast Method (1/2)

• _tail already points to the last Node in the list

• Create a new Node<Type>

• Update _tail’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) to new last `Node`, and then 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 (_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;
}
```
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 first Node’s successor Node via first’s _next

- Node to remove is garbage-collected at the termination of the method
removeFirst Method (2/2)

- Edge case for empty list
  - println is optional, one way to handle error checking; caller should check for null in any case
- Store 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 (how did _head get set?)
- Node to remove is garbage-collected at method’s end

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

→ removeFirst() is O(1)
removeLast Method

- As with removeFirst, remove Node by removing any references to it
- Pointer-chase in a loop to get predecessor to `_tail` and reset predecessor’s `_next` instance variable to null
  - pretty inefficient – stay tuned
- 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

```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--;
    } 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;
}
```
removeLast Method

• Edge case(s)
  o can’t delete from an empty list
  o if there is only one Node, update _head and _tail references to null

• General case
  o iterate (pointer-chase) through list – common pattern
  o 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;
    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;
    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;
    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;
    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;
    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
}
Clicker Question

Given that _animals is a Singly Linked List with \( n \) animals in the list, what would be the total runtime of this method, and what would be printed to the console?

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

A. \( O(1) \); “Not Empty”
B. \( O(1) \); no print
C. \( O(n) \); “Not Empty”
D. \( O(n) \); no print
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
  o but in order to do this, we first need a way to check for the equality of two elements!
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>` 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!

Andries van Dam © 201611/01/16
Java’s `Comparable<Type>` interface (2/3)

- The `Comparable<Type>` interface is specialized (think of it as parameterized) using generics

```
public interface Comparable<Type> {
    public int compareTo(Type toCompare);
}
```

- Call `compareTo` on a variable of same type as specified in implementation of interface (`Book`, in our case)
  - `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

- `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:
  - `elementA.compareTo(elementB)`

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

- `compareTo` is defined according to the specifications
  - returns number that is `< 0, 0` or `> 0`, depending on the ISBN numbers
  - neg. if `_isbn < toCompare`
“Comparable” Singly Linked List

- Using keyword `extends` in this way ensures that `Type` implements `Comparable<Type>`
  - Note nested `<`, and that `extends` is used differently.

- All elements stored in `MyLinkedList` must now have `compareTo` method for `Type`
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 = _head;
    
    while (curr != null) {
        if (curr.getElement().compareTo(el) == 0) {
            return curr;
        }
        curr = curr.getNext(); //bop pointer
    }
    return null;
}
```
search Runtime

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
  - will be similar to the `search` algorithm.
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 an 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`

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)
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 more efficient:
    ▪ search
    ▪ insert
    ▪ delete

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

```java
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 54/98
```
Generic Sorted Singly Linked List

• Slightly different set of methods
  o 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

- Compare `toFind` to `curr`'s element
  - if `==`, 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;
}
```
public Node<Type> search(Type toFind) {
    Node<Type> curr = _head;
    while (curr != null) {
        // 1 op
        // n ops
        if (curr.getElement().compareTo(toFind) == 0) {
            return curr;
        } // 1 op
        else if (curr.getElement().compareTo(toFind) > 0) { // 1 op
            return null;
        } // 1 op
        curr = curr.getNext(); // 1 op
    } // 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 lot of searching compared to insertion and deletion, this efficiency would pay off; conversely, if an algorithm does a lot of adding and deleting compared to searching, it wouldn’t since the algorithm would use the simple $O(1)$ `insertFirst` or `insertLast`...
**insert method**

- Once again, iterate through nodes with a `while` loop, keeping track of current and previous nodes.
- Unlike insertion into unsorted linked list, there is one correct spot in 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**
  - 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

```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 null only at front of list
                    prev.setNext(toAdd);
                } else {
                    _head = toAdd;
                }
                _size++
                return toAdd;
            }
            prev.setNext(toAdd); //not found, insert node at end
            _tail = toAdd;
            _size++;
            return toAdd;
        }
    }
}
```
**insert** Runtime [for Sorted Linked Lists]

```java
public Node<Type> insert(Type newItem) {
    Node<Type> toAdd = new Node<Type>(newItem); // 1 op
    if (this.isEmpty()) {
        // 1 op
        _head = toAdd;
        _tail = toAdd;
        return _head;
    }
    else {
        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;
                curr = curr.getNext(); // 1 op
            }
            else { // 1 op
                toAdd.setNext(curr);
                if (prev != null) {
                    prev.setNext(toAdd); // 1 op
                }
                else { // 1 op
                    _head = toAdd;
                }
                _size++; // 1 op
                return toAdd; // 1 op
            }
        }
        prev.setNext(toAdd); // 1 op
        _tail = toAdd; // 1 op
        _size++; // 1 op
        return toAdd; // 1 op
    }
}
```

→ `insert(Type newItem)` is O(n)
**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`), so reset next pointer in `prev` node and return found item
    - or if `curr` is greater than `itemToRemove`, it can’t be in the list, hence return `null`
  - 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 MyLinkedList’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; // 1 op
    } // 1 op
    if (itemToRemove.compareTo(_head.getElement()) == 0) { // 1 op
        // elided; same as MyLinkedList’s removeFirst code // 0(1)
    } // 1 op
    if (itemToRemove.compareTo(_tail.getElement()) == 0) { // 1 op
        // elided; same as MyLinkedList’s removeLast code // 0(n)
    } // 1 op
    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
        } // 1 op
        else if (curr.getElement().compareTo(itemToRemove) > 0) { // 1 op
            return null; // 1 op
        } // 1 op
        prev = curr; // 1 op
        curr = curr.getNext(); // 1 op
    } // 1 op
    return null; // 1 op
}

Again, the else if statement will often improve performance, but it does not affect Big-O run time

→ remove(Type itemToRemove) is O(n)
Clicker Question

How do sorted and unsorted lists differ?

A. Sorted lists are more efficient than unsorted lists and are more easily searched, but their insertion takes longer.
B. Sorted lists are less efficient than unsorted lists and are less easily searched, but their insertion takes less time.
C. Sorted lists are more efficient than unsorted lists and are more easily searched and their insertion takes less time.
D. Sorted lists are more efficient than unsorted lists, but are less easily searched and their insertion takes longer.
• 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., sort students on their final grades. find the lowest numeric grade that will be recorded as an “A”. now ask: who is close to getting an “A”, i.e., borderline?

\(^1\) 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
  - could build our own *specialized search* method, which would scan from the _head_ and be, at a minimum, O(n)

- It is trivial 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
Remove method [For Sorted Doubly Linked List]

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

• Other methods are analogous

```python
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
    return n’s Type
```
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;
    }

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

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

    /*Mutator and accessor method for _element are elided.*/

}
Clicker 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()
Summary of Linked Lists

- Linked Lists can be grouped into two categories:
  - Unsorted vs. Sorted Linked List
- Unsorted Linked List has two different implementations with identical method signatures and return types
  - Unsorted Singly vs. Unsorted Doubly Linked List
- Sorted Linked List also has two different implementations with identical method signatures and return types
  - Sorted Singly vs. Sorted Doubly Linked List
- Implementation matters! While an Unsorted Singly Linked List and an Unsorted Doubly Linked List have the same functionality, they have different runtimes for their methods.
  - pick implementation based on how you expect to use the data structure
Circular Doubly Linked Lists

- No beginning or end
- Example: Rolodex
- In operating systems, these structures are called rings
When to use a Linked List?

- Might use a Linked List if...
  - do not need random access (based on an index)
  - often need to insert elements into or delete from the interior of a sorted list
  - do not know how many elements will be stored beforehand (then still need to trade-off using List vs. ArrayList)
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)
  o 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:
  o 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;
    }
}
```
search Private Helper Method Runtime

```java
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

```java
class Node<Type>

    Type _element
    Node<Type> _next

_class

_type

_head

_next

_type

_tail

_next
```

```
_node

_next

_type

_tail

_next
```

1. Node<Type>
   Type _element
   _next

2. Node<Type>
   Type _element
   _next

3. Node<Type>
   Type _element
   _next

4. Node<Type>
   Type _element
   _next

5. Node<Type>
   Type _element
   _next

6. Node<Type>
   Type _element
   _next

7. Node<Type>
   Type _element
   _next

8. Node<Type>
   Type _element
   _next
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 \textit{in-place} – (had to create a new list)

• Can write a method \textit{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
curr
next = null
prev = null
Solution B Walkthrough (2/15)

```
prev = null
curr
next = null
_tail = _head;
prev = null
```

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

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

Andries van Dam © 201611/01/16
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (5/15)

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

A

B

C

_head

_tail

curr

prev

null

next
Solution B Walkthrough (6/15)

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

A

B

C

null

_prev

_next

_head

_tail
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (8/15)

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

- head
- tail
- prev
- curr
- next
- null
Solution B Walkthrough (9/15)

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

null

_head

_tail

A -> B -> C

prev curr next
Solution B Walkthrough (10/15)

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

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

prev
curr
next = null

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

null
_head
_tail
A
B
C

Andries van Dam © 201611/01/16
94/98
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;
}
null

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

_solution B Walkthrough (14/15)_

_andries van dam © 2016_
Solution B Walkthrough (15/15)

A

B

C

null

_next = null

_prev

_curr = null

_next = null

_andhead = prev;

_andhead = prev;

_andhead = prev;

Andries van Dam © 201611/01/16
Announcements

• DoodleJump due dates
  - on-time hand-in is today at 11:59 pm
  - late hand-in is Saturday at 10:00 pm

• We are now in MacMillan 117!