Use Cases for Different Data Structures for Collections (1/2)

- Many data structures store a collection of elements
- ArrayLists are called that because they implement the List interface and are implemented using Arrays
- Can define a building block called LinkedList, which is better than Arrays for some use cases by avoiding data movement for insertion and deletion

Note: in JavaFX, we use a special collection to represent the children of a Node that acts as an ObservableList – an interface that extends the List interface

Use Cases for Different Data Structures for Collections (2/2)

- Using LinkedList of nodes, can construct higher level abstractions (e.g. a NodeList to parallel ArrayList, Stacks, Queues, etc.)
- How to decide between data structures?
  - choose based on the way data is accessed and stored in your algorithm
  - access and store operations of different data structures can have very different impacts on an algorithm’s overall efficiency

What is a LinkedList? (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
  - your algorithm can take advantage of the fact that elements are stored sequentially, or not
- A LinkedList 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

---

**Array**
- indexed (explicit access to \( n^{th} \) item)
- indices of items in an array do not change dynamically, it is up to the user to reassign index values on insertion or deletion
- cannot change size dynamically

---

**ArrayList**
- indexed (explicit access to \( n^{th} \) item)
- indices of other items are 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

---

**Ex: HTA Linked List**

![Instance Diagram](image)

**LinkedList 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
  o _head and _tail are initially set to null
- addFirst creates first node and sets _head to reference it
- addLast appends a Node to the ends of the list and sets _tail to 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 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) {
    //...
}
```

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> {
    public Node<Type> remove(Type e1) {
        //...
    }
    public Node<Type> search(Type e1) {
        //...
    }
    public int size() {
        //...
    }
    public boolean isEmpty() {
        //...
    }
    public Node<Type> addFirst(Type el) {
        //...
    }
    public Node<Type> addLast(Type el) {
        //...
    }
    public Node<Type> removeFirst() {
        //...
    }
    public Node<Type> removeLast() {
        //...
    }
    public Node<Type> getHead() {
        //...
    }
    public Node<Type> getTail() {
        //...
    }
```
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 Type _element;
    private Node<Type> _next;

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

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

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

    public void setElement(Type element) {
        _element = element;
    }

    public Type getElement() {
        return _element;
    }
}
```

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

- Let's use a Linked List to model a simple pile (i.e. set) of **Books**
- The elements in our pile will be of type **Book**
  - have titles, authors, dates and ISBN numbers
  - 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**

```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
}
```

**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`
**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`

---

**addFirst – empty list**

- If list is empty, `_head` and `_tail` will be `null`
- Create a new `Node<Type>`
- Set new node’s `_next` variable to `null`, which is where current `_head` points in this case
- Set the `_head` and `_tail` variables to the new node

---

**addFirst - non empty**

- Create a new `Node` `_head`
- Set its `_next` variable to current `_head` (in this case, some previously added `Node` that heads list)
- Set `_head` variable to the new `Node`
- `_tail`

---

**Ex: MyLinkedList<Book>**

```
MyLinkedList<Book> _books
null
```

```
Node<Book> _head
Node<Book> _tail
set_size = 4
```

```
Node<Book> _next
Type_element
_ tail
null
```

```
Type_element
_ head:_null
_ tail:_null
```

---

```
new node
```

---

```
new node
```

---

```null
```
Constructor and `addFirst` Method

- **Constructor**
  - initialize instance variables
- **`addFirst` method**
  - increment `_size` by 1
  - create new Node
  - set new Node's `next` pointer to first Node
  - update `_head` to point to new Node
  - if `_size` is 1, `_tail` also points to new Node
  - return new Node

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

public Node<Type> addFirst(Typ e 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

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

public Node<Type> addFirst(Typ e 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) { // 1 op
        _tail = newNode; // 1 op
    }
    return newNode; // 1 op
}
```

→ `constructor` is O(1)

```
public Node<Type> addFirst(Typ e 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) { // 1 op
        _tail = newNode; // 1 op
    }
    return newNode; // 1 op
}
```

→ `addFirst(Typ e)` is O(1)

The new Node's `next` variable already points to null

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

`addLast` Method

- **Edge Case**
  - if list is empty, set the `_head` and `_tail` variables to the new Node
- **General Case**
  - Set `_next` variable of current last Node (to which `_tail` is pointing) to new last Node,
    and then set `_tail` to that new last Node
  - The new Node's `_next` variable already points to null

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

22/89

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```

```
Node<Type> [node]
| _head |
| node |
| _tail |
```
**addLast Runtime**

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

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

→ `size()` is O(1)

→ `isEmpty()` is O(1)

**removeFirst Method**

- 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

- 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, set `_tail` to null
- 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;
}
```

removeFirst Runtime

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

→ removeFirst(Type el) is O(1)

removeLast Method

- As with removeFirst, remove Node by removing any references to it
- Pointer-chase to get predecessor to `_tail` and reset predecessor’s `_next` instance variable to null
- Pretty inefficient — stay tuned
- Last Node is thereby garbage-collected!

```java
public Type removeLast() {
    Type removed = null;
    if (_size == 0) {
        System.out.println("List is empty");
        return null;
    } 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) {
            // op the pointers
            prev = curr;
            curr = curr.getNext();
        }
        removed = curr.getElement();
        prev.setNext(null); // unlink last
        _tail = prev;
        _size--;
    }
    return removed;
}
```

removeLast Method

- Edge case(s)
  - can’t delete from an empty list
  - if there is only one Node, null the _head and _tail references
- General case
  - iterate (“pointer-chase”) through list
  - 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");
        return null;
    } else {
        // classic pointer-chasing loop
        Node curr = _head;
        Node prev = null;
        while (curr.getNext() != null) {
            // op the pointers
            prev = curr;
            curr = curr.getNext();
        }
        removed = curr.getElement();
        prev.setNext(null); // unlink last
        _tail = prev;
        _size--;
    }
    return removed;
}
```
removeLast Runtime

```java
public Type removeLast() {
    Type removed = null;
    if (_size == 0) {
        // 1 op
        System.out.println("list is empty");
    } else if (_size == 1) {
        // 1 op
        removed = _head.getElement();
        _head = null;
        _tail = null;
        _size--; // 1 op
    } else {
        Node curr = _head;
        Node prev = null;
        while (curr.getNext() != null) { // 1 op
            prev = curr;
            curr = curr.getNext(); // 1 op
        }
        removed = curr.getElement();
        prev.setNext(null); // 1 op
        _tail = prev; // 1 op
        _size--; // 1 op
    }
    return removed; // 1 op
} // removeLast(Type el) is O(n)
```

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?
- Must compare each Book with the one we are looking for
  - 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, we 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

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);
  }
  ```
- You call compareTo on a variable of same type as specified in implementation of interface (Book, in our case)
Java’s Comparable<Type> interface (3/3)

- `compareTo` method must return an `int`
  - negative if object on which `compareTo` is called is less than object passed in as a parameter
  - 0 if object is equal to object passed in
  - positive if this object is greater than object passed in as a parameter
- `compareTo` not only used for numerical comparisons—it could be used for alphabetical comparisons as well!

“Comparable” Book Class

- Book class now implements `Comparable<Book>`
- `compareTo` is defined according to the specifications
  - returns number that is < 0, 0 or > 0, depending on the ISBN numbers
  - neg if `_isbn` < `compareTo`

```java
public class Book implements Comparable<Book> {
    // variable declarations, e.g. _isbn, elided
    public Book(String author, String title, int isbn)
        // variable initializations elided
        {
            _isbn = isbn;
        }
        public java.util.date getISBN()
        {
            return _isbn;
        }
        // other methods elided
        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
- All elements stored in MyLinkedList must now have `compareTo` method for `Type`

```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 ) {
        //...
    }
    // other methods elided
}
```

Search Method

- 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`
**Search Runtime**

```java
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
  - This will be similar to the `search` algorithm

```java
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) { // pointer-chasing loop to find el
        if (itemToRemove.compareTo(curr.getElement()) == 0) {
            prev.setNext(curr.getNext()); // jump over node
            _size--; // decrement size
            return curr.getElement();
        }
        prev = curr; // if not found, bop pointers
        curr = curr.getNext();
    }
    return null; // return null if itemToRemove is not found
}
```

**remove Method**

- Loop through Nodes until find one whose `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** Runtime

```java
public Type remove(Type itemToRemove)
{
    if (this.isEmpty()) {
        // 1 op
        System.out.println("List is empty");
        return null;
    }
    if (itemToRemove.compareTo(_head.getElement()) == 0) {
        // 1 op
        return this.removeFirst(); // O(1)
    }
    if (itemToRemove.compareTo(_tail.getElement()) == 0) {
        // 1 op
        return this.removeLast(); // O(n) pointer chase till list end
    }
    Node<Type> curr = _head.getNext(); // 1 op
    Node<Type> prev = _head;
    while (curr != null) {
        // n ops
        if (itemToRemove.compareTo(curr.getElement()) == 0) {
            // 1 op
            prev.setNext(curr.getNext());
            // 1 op
            _size--; // 1 op
            return curr.getElement(); // 1 op
        }
        prev = curr;
        // 1 op
        curr = curr.getNext(); // 1 op
    }
    return null; // 1 op
}
```

Ex: remove is O(n)

Ex: A sorted bookshelf

- Faster to find (and remove!) books in a sorted bookshelf
- Use a sorted linked list
  - makes several of our methods more efficient:
    - search
    - insert
    - delete
- Sort in increasing order:
  - maintain sort order when inserting

**Ex: MySortedLinkedList<Book>**

```java
public class MySortedLinkedList<Book> {
    private Node<Book> _head;
    private Node<Book> _tail;
    private int _size;
    public MyLinkedList() {
        // ...
    }
    public Node<Book> insert(Book el) {
        // ...
    }
    public Type remove(Book el) {
        // ...
    }
    public Node<Book> search(Book el) {
        // ...
    }
    public int size() {
        // ...
    }
    public boolean isEmpty() {
        // ...
    }
    public Node<Book> getHead() {
        // ...
    }
    public Node<Book> getTail() {
        // ...
    }
}
```

Generic Sorted Singly Linked List

- Slightly different set of methods
  - addFirst and addLast replaced by general insert
- Many methods whose signatures haven't changed will have different implementation because of more efficient loop termination used in search and remove

---

11/3/15
**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
  - we know that 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 search(Type toFind) {
    Node curr = _head;
    while (curr != null) {
        if (toFind.compareTo(curr.getElement()) == 0) {
            return curr;
        } // haven't found it, hop the ptr
        else if (curr.getElement().compareTo(toFind) > 0) {
            curr = curr.getNext();
        }
    } // haven't found it, hop the ptr
    return null;
}
```

**search Runtime** [for Sorted Linked Lists]

```java
public Node search(Type toFind) {
    Node curr = _head;
    while (curr != null) {
        if (toFind.compareTo(curr.getElement()) == 0) {
            return curr;
        } // haven't found it, hop the ptr
        else if (curr.getElement().compareTo(toFind) > 0) {
            curr = curr.getNext();
        } // haven't found it, hop the ptr
    } // haven't found it, hop the ptr
    return null;
}
```

While the else if statement will typically improve performance because 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(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 you do a lot of searching compared to insertion and deletion, this efficiency would pay off; conversely, if you do a lot of adding and deleting compared to searching, it wouldn’t since you’d use the simple O(1) `insertFirst` or `insertLast`...
insert Method
[for Sorted Linked Lists]

- **Edge case**
  - if list is empty, we have to do is reset _head/tail

- **General case**
  - iterate over list until curr's element is greater than newItem
  - need loop's prev, so we can re-link list to integrate the new node

```java
public Node<Type> insert(Type newItem) {
    Node<Type> toAdd = new Node<>(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) {
            if (curr.getElement().compareTo(newItem) < 0) {
                prev = curr;
                curr = curr.getNext();
            } else if (found the spot) {
                toAdd.setNext(curr);
                if (prev != null) {
                    prev.setNext(toAdd);
                } else { // prev is null at front of list
                    _head = toAdd;
                }
                _size++;
                return toAdd;
            }
            prev.setNext(toAdd); // not found, insert node at end
            _tail = toAdd;
            _size++;
            return toAdd;
        }
    }
    return _head;
}
```

```
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 set next pointer in prev and return found item
    - or curr is greater than the rest of the elements in the list, return null
  - or we reach end of list, so return null

```java
public Node<Type> remove(Type itemToRemove) {
    if (this.isEmpty()) {
        return null;
    } else {
        Node<Type> curr = _head;
        Node<Type> prev = null;
        while (curr != null) {
            if (curr.getElement().compareTo(itemToRemove) == 0) {
                prev = curr; // copy pointers
                itemToRemove = curr.getElelement(); // jump over node
                _size--;
                return curr.getNext(); // curr points to first
            } else if (curr.getElement().compareTo(itemToRemove) > 0) {
                return curr; // loop pointers, items
                curr = curr.getNext();
            } return null;
        }
    }
    return null;
}
```
remove Runtime [for Sorted Linked Lists]

```java
public Type remove(Type itemToRemove)
{
    if (this.isEmpty()) {
        return null;
    }
    if (itemToRemove.compareTo(_head.getElement()) == 0) {
        // elided; same as MyLinkedList's removeFirst code  // O(1)
    }
    if (itemToRemove.compareTo(_tail.getElement()) == 0) {
        // elided; same as MyLinkedList's removeLast code  // O(n)
    }
    Node<Type> curr = _head.getNext();
    Node<Type> prev = _head;
    while (curr != null) {
        if (curr.getElement().compareTo(itemToRemove) == 0) {
            prev.setNext(curr.getNext());
            _size--; // 1 op
            return curr.getElement();
        } else if (curr.getElement().compareTo(itemToRemove) > 0) {
            // O(n)
            return null; // 1 op
        }
        prev = curr;
        curr = curr.getNext();
    }
    return null; // 1 op
}
```

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

Doubly Linked List (1/3)

- Is there an easier/faster way to get to the previous node for ease in removing a node?
  - with Doubly Linked Lists, nodes have references both to next and previous nodes
  - can traverse list both backwards and forwards — Linked List still stores reference to front of the list with _head and back of the list with _tail
  - 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, you want to be able to go in the opposite direction
  - 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?

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
  - we’d have to 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 leave B+/A-grey area

1 Like Singly Linked Lists, Doubly Linked Lists can be sorted or unsorted. We only discuss the sorted version here.
Remove method  [for Sorted Doubly Linked List]

- This is pseudo-code for a remove method for a Sorted Doubly Linked List
  - note that we are using 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

Node  [for Doubly Linked Lists]

```java
public class Node<Type> {
    private Type _element;
    private Node<Type> _next;
    private Node<Type> _prev;

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

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

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

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

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

Summary of Linked Lists

- We’ve introduce four different implementations of Linked Lists
- They 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 will 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?

- You might use a Linked List if...
  - do not need random access
  - need constant time insertion/removal at head or tail of list
  - often need to insert elements into the interior of the list
  - do not know how many elements you will store a priori

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 the 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 the 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
**Private Helper Method Runtime**

```java
private Node<Type> search(int i) {
    if (i >= _size || i < 0) {
        System.out.println("Invalid index");
        return null;
    }
    Node<Type> curr = _head;
    for (int counter = 0; counter < i; counter++) {
        curr = curr.getNext();
    }
    return curr;
}
```

+ search(int i) is O(n)

---

**Public Wrapper Method**

- Finally, let's write the publicly accessible wrapper code for the **NodeList's get method**.

```java
public Type get(int i) {
    return this.search(i).getElement();
}
```

- this shows a very common pattern of "thin wrappers" over private code

---

**An Exercise**

("CS16-Style", common job interview question)

- Write a method that reverses the order of a **MyLinkedList**

```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;
}
```

- 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

---

**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
Solution B (1/2)

- Is there a better way?
- First algorithm reversed in \( O(n) \) time
  - but it wasn't in place – (we had to create a new list)
- We can write a method within `MyLinkedList` that reverses itself without creating new nodes
  - still \( O(n) \) but in place and therefore more efficient

```
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;
}
```

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

Announcements

- DoodleJump due dates
  - early hand-in is tomorrow at 11:59pm
  - on-time hand-in is Friday at 10:00 pm
  - late hand-in is Sunday at 10:00 pm

- Tetris Design Checks go out on Thursday
  - This time we will not be as lenient with rescheduling