Algorithmic Bias in Health Care

- Algorithm used by health care providers to screen patients for 'high-risk care' is biased
- Favors certain patients over others with same health burden
- Screenings and automatic enrollment determined by patient health-care spending
- Reinforces systemic racial bias
- Black patients were much sicker at a given level of the algorithm's predicted risk

Sources:
https://www.theverge.com/2019/10/24/20929337/care-algorithm-study-race-bias-health
https://news.uchicago.edu/story/health-care-prediction-algorithm-biased-against-black-patients-study-finds

Consequences

- "Almost every large healthcare system is using it, as well as institutions like insurers
- 70 Million patients affected
- Time and resources not being spent on highest need patients
- Algorithm reduced the proportion of black patients receiving extra help from almost 50% to less than 20%
- Algorithm could be easily remedied
  - looking at subsets of data i.e. only emergency room costs.
  - considering additional data: chronic condition flare ups

In what ways does the medical screening algorithm mirror and reinforce societal trends?
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
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

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

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
• 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
• 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
  - _head and _tail are initially set to null
  - _size set to 0
- addFirst creates first Node and updates _head to reference it
- addLast appends a Node to the end of the list and updates _tail to reference it

```java
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 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 el and returns it

```java
public Node<Type> removeFirst() {
    //...
}
public Node<Type> removeLast() {
    //...
}
public Node<Type> remove(Type el) {
    //...
}
// still more on next slide
```

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() {
    //...
}
```
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 getNext()
  - 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
  - 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 (International Standard Book Number)

  - 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()  

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

  - For each property, its **get** method returns that property's value
    - **getISBN()** returns _isbn

---

**Book Class**

```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 mutator and accessor methods elided

---

**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.
  - *PileOfBooks*’ methods are "wrappers" over the underlying methods from *MyLinkedList*.

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

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

//There could be many more methods here.

---

**Ex: MyLinkedList<Book>**

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

- addFirst method
  - _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)

- 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 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

```java
Node<Type> newNode = new Node<Type>(el)
```

**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 = \_next tail’s \_next) to new last Node
  - update \_tail to that new last Node
  - new Node’s \_next variable already points to null

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

**addLast Runtime**

```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;
}
```

\[ \text{addLast}(\text{Type } el) \in O(1) \]
**size and isEmpty Methods**

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

(Empty) 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
  - if printin 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

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)

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
- What does _tail getNext() produce?
- What does _tail getElement() produce?
**TopHat Question**

Given a Linked List, 

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

where A is the _head_, 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
  - pretty inefficient—stay tuned
- Update _tail_
- Last Node is thereby garbage-collected!

```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) {
            prev = curr;
            curr = curr.getNext();
        }
        removed = curr.getElement();
        curr.setNext(null); //unlink last
        tail = prev;
        _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
removeLast Method

```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;
        _size--;
    } else {
        // classic pointer-chasing loop
        Node curr = _head;
        Node prev = null;
        while (curr.getNext() != null) {
            prev = curr;
            curr = curr.getNext();
        }
        removed = curr.getElement();
        prev.setNext(null);  // unlink last
        _tail = prev;  // update _tail
        _size--;
    }
    return removed;
}
```

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

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

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

removeLast Runtime

```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;
        _size--;
    } else {
        Node curr = _head;
        Node prev = null;
        while (curr.getNext() != null) {
            prev = curr;
            curr = curr.getNext();
        }
        removed = curr.getElement();
        prev.setNext(null);
        _tail = prev;
        _size--;
    }
    return removed;
}
```

+ removeLast() is O(n)
TopHat Question

Given that \texttt{animals} is a Singly Linked List of \texttt{n} animals already in the list, 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");
    }
    System.out.println(_animals.removeLast());
}
```

A. "Not Empty", "Dog"
B. no print, "Cat"
C. "Not Empty", "Fish"
D. no print, "Dog"

TopHat Question

Given that \texttt{animals} is a Singly Linked List of \texttt{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
  o but in order to do this, we first need a way to check for the equality of two elements!
  o 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
  o this only works correctly for primitive data types (e.g., int), or when we are comparing two variables referencing the exact same object
  o 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?
  o 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
  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)
  o currentBook.compareTo(bookToFind);
  o pseudo-code: currentBook?bookToFind

Java's Comparable<Type> interface (3/3)
• compareTo method must return an int
  o negative if element on which compareTo is called is less than element passed in as the parameter of the search
  o 0 if element is equal to element passed in
  o positive if element is greater than element passed in
  o 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`:
  - `elementA.compareTo(elementB)`
- Book class now implements `Comparable<Book>`
  - This means we can compare books using `bookA.compareTo(bookB)`
- `compareTo` is defined according to these specifications:
  - Returns number that is `<0`, `0`, or `>0`, depending on the ISBN numbers
  - `<0` if stored _isbn `< toCompare parm

```java
public class Book implements Comparable<Book> {
    // variable declarations, e.g. _isbn, elided
    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) {
        //...
    }
    // other methods elided
    // search Method for MyLinkedList (brute force)
    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 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
search Runtime

public Node<Type> search(Type el) {  // 1 op
    Node<Type> curr = _head;  // 1 op
    while (curr != null) {  // 8 ops
        if (curr.getElement().compareTo(el) == 0) {  // 1 op
            return curr;  // 1 op
        }
        curr = curr.getNext();  // 1 op
    }
    return null;  // 1 op
}  // 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

• 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

+ search(Type el) is O(n)
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. Node is GC’d upon return.

```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) {
        // loop to get tail’s predecessor
        return this.removeLast();
    }
    Node<Type> curr = _head.getNext(); // advance to 2nd item
    Node<Type> prev = _head;
    while (curr != null) {
        // pointer-chasing loop to find el.
        if (curr.getElement().compareTo(itemToRemove) == 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
}
```

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

remove Runtime

```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) {
        // O(n) pointer chase till list end
        return this.removeLast();
    }
    Node<Type> curr = _head.getNext(); // advance to 2nd item
    Node<Type> prev = _head;
    while (curr != null) {
        // O(n) pointer chase till list end
        if (curr.getElement().compareTo(itemToRemove) == 0) {
            prev.setNext(curr.getNext());
            _size--;
            // decrement size
            return curr.getElement();
        }
        prev = curr;
        curr = curr.getNext();
    }
    return null;
}
```

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

TopHat Question

Given that \_animals is a Singly Linked List of \( n \) animals, curr points to the node with an animal to be removed from the list, and that prev points to curr’s predecessor, what will this code fragment do?

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
  - makes several of our methods somewhat more efficient:
    - search
    - insert
    - delete
- Sort in increasing order
  - maintain sort order when inserting

Ex: MySortedLinkedList<Book>

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

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) {  // 1 op
  Node<Type> curr = _head;  // N ops
  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
      curr = curr.getNext();  // 1 op
    }  // 1 op
  }
  return null;  // 1 op
}
```

search Runtime [For Sorted Linked Lists]

```java
public Node<Type> search(Type toFind) {  // 1 op
  Node<Type> curr = _head;  // N ops
  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
      curr = curr.getNext();  // 1 op
    }  // 1 op
  }
  return null;  // 1 op
}
```

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, 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
  - Binary Search in a sorted array and a Linked List as a Binary Tree makes search O(log n)
    (play 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
  - checks if two objects point to the same place in memory
  - good for comparing primitive types (integers, doubles, booleans, etc.)

- The "equals()" method compares content
  - checks if two objects have the same values
  - 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

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 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 is null only at front
                    prev.setNext(toAdd);
                } else {
                    // head = toAdd;
                    // insert at front of list
                    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

**remove Method**  
*for Sorted Linked Lists*

- Loop through nodes until an _element_ matches `itemToRemove`
  - 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 curr's element is greater than itemToRemove, so return null
    - or we reach end of list, so return null

```java
public Type remove(Type itemToRemove) {
    if (this.isEmpty()) {
        return null;
    } else if (itemToRemove.compareTo(_head.getElement()) == 0) {
        // elided; same as MyLinkedList's removeFirst code
    } else if (itemToRemove.compareTo(_tail.getElement()) == 0) {
        // elided; same as MyLinkedList's removeLast code
    } else {
        Node<Type> curr = _head.getNext();
        Node<Type> prev = _head;
        while (curr != null) {
            if (curr.getElement().compareTo(itemToRemove) == 0) {
                prev.setNext(curr.getNext());
                _size--;
                return curr.getElement();
            } else if (curr.getElement().compareTo(itemToRemove) > 0) {
                return null;
            } else {
                prev = curr;
            }
            curr = curr.getNext();
        }
        return null;
    }
    return null; // End of list, no finding it
}
```

TopHat 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 efficiently 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.

remove Runtime [for Sorted Linked Lists]

```java
public Type remove(Type itemToRemove) {
    if (this.isEmpty()) {
        return null;
    } else if (itemToRemove.compareTo(_head.getElement()) == 0) {
        // O(1)
    } else if (itemToRemove.compareTo(_tail.getElement()) == 0) {
        // O(n)
    } else {
        Node<Type> curr = _head.getNext();
        // n ops
        while (curr != null) {
            if (curr.getElement().compareTo(itemToRemove) == 0) {
                // 1 op
                prev.setNext(curr.getNext());
                // 1 op
                _size--;
                return curr.getElement();
            } else if (curr.getElement().compareTo(itemToRemove) > 0) {
                // 1 op
                return null;
            } else {
                prev = curr;
                // 1 op
            }
            curr = curr.getNext();
        }
        return null;
    }
    return null; // End of list, no finding it
}
```

Again, the **else if** statement will often improve performance, but it does not affect Big-O run time.
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)
  - can stop looping if we find an element larger than what we are searching for
  - 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

Unsorted vs. Sorted Singly Linked List (3/3)

- 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
Our discussion has not taken into account memory hierarchies
• (superfast index registers, not suitable for storing more than frequently-used vars)
• 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)
• next is system memory (aka CPU memory, RAM) – much larger, much slower
• next is peripheral memory such as flash drives, disk drives, etc., – much slower
• 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

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 (milliseconds 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?
  - 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\(^1\), need to go in the opposite direction
    - e.g., we sort CS15 students on their final grades in ascending order:
      find the lowest numeric grade that will be recorded as an "A". We then ask: who has a lower grade but is close to the "A" cut-off, i.e., in the grey area, and therefore should be considered for "benefit of the doubt"?

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

```
remove(t)
  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._element
```

Node [For Doubly Linked Lists]

```
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;
  }

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

TopHat 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

Single
- To look for a node (search, insert, delete) must start at the beginning and go to the next node n times → no random access!
- Dynamically grow/shrink
- Can insert or remove nodes anywhere in the list

Double
- Same functionality as Single
- Has a previous pointer _prev
  - for removeLast, don't need to loop through entire list to find predecessor
  - don't have to maintain separate prev pointer for search
- 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
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) { // 1 op
            System.out.println("Invalid index"); // 1 op
            return null; // 1 op
        } // 1 op

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

        return curr; // 1 op
    }
}
```

→ `search(int i)` is O(n)

Write the publicly accessible wrapper code for the `NodeList`’s `get` method. This shows a very common pattern of “thin wrappers” over private code.

```java
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
  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

Solution B (1/2)

- Is there a better way?
- First algorithm reversed in O(n) time
  - 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
  - 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)

![Diagram of head, tail, A, B, C with prev = null, curr = B, next = C]

Solution B Walkthrough (2/15)

![Diagram of head, tail, A, B, C with prev = null, curr = A, next = B]
prev = null

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

_solution B Walkthrough (3/15)_

prev = null

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

_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)_

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 (9/15)

Solution B Walkthrough (10/15)

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

Announcements

• DoodleJump early deadline tonight! Late deadline on Saturday
• Please add your Banner ID to your TopHat account!