CSCI 261
Computer Science II
Lists and Complexity

- Take a look at the following code, which adds all elements in the array

```java
int sum = 0;
for(int i = 0; i < n; i++) {
    sum = sum + a[i];
}
```

- Is the simple statement in the loop actually constant-time?
  - Is `+` an $O(1)$ operation?
  - Are array references an $O(1)$ operation?
    - What if we assume elements at larger indices take longer to reference?
    - What if we assume all elements have constant-time reference?
Well, as we know, arrays are just one kind of list

- Arrays are great for some problems, but not all
  - Things are not usually one-size-fits-all
  - What if your algorithm requires *lots* of insertion/deletion?

- *We want options!*

Maybe we can define other kinds of lists that have tradeoffs:

- Other types of lists might have faster insertion and deletion
  - But maybe that would cause element reference to suffer

- Other types of lists might be able to grow and shrink
  - But maybe that would cause insertion/deletion time to suffer
For each type of list, we need to model the time complexity of its common operations

- e.g., construction, insertion, deletion, search, ...

Understanding the complexity of its operations allows us to make the proper choice when programming

- Use the one that'll give you the best performance on average
- One of the most important skills
  - Need to have a fundamental understanding of data structures' (e.g., lists) implementation
Outline

- Java's List Interface
  - ArrayList
  - Aside: Generics
  - Linked List
  - Iterator
- Conclusion
Java's List\(<E>\) Interface

- Java provides a List\(<E>\) interface
  - The diamond notation \(<E>\) allows for "generic" objects to be stored
  - Full API: https://docs.oracle.com/javase/8/docs/api/java/util/List.html

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public E get(int index)</td>
<td>Returns the data in the element at position index</td>
</tr>
<tr>
<td>public E set(int index, E item)</td>
<td>Stores a reference to item in the element at position index. Returns the data formerly at position index</td>
</tr>
<tr>
<td>public int size()</td>
<td>Return the current size of the List</td>
</tr>
<tr>
<td>public boolean add(E item)</td>
<td>Adds a reference to item at the end of the List. Always returns true</td>
</tr>
<tr>
<td>public boolean add(int index, E item)</td>
<td>Adds a reference to item, inserting it before the item at position index</td>
</tr>
<tr>
<td>public int indexOf(E item)</td>
<td>Searches for target and returns the position of the first occurrence, or –1 if it is not in the List</td>
</tr>
<tr>
<td>public E remove(int index)</td>
<td>Removes the entry formerly at position index and returns it</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Java's List Interface (Cont.)

- List hierarchy and implementing classes

```
<<interface>>
List

<<implements>>
AbstractList

<<extends>>
ArrayList
Vector
AbstractSequentialList

<<extends>>
Stack
LinkedList

Not abstract classes (just poorly named)
```
Outline

- Java's List Interface
  - ArrayList
- Aside: Generics
- Linked List
- Iterator
- Conclusion
Motivation: Arrays cannot be resized easily

Designers of Java understood this limitation, and created `ArrayList`

- Can hold any number of objects
- Can hold any type of object (generics)
- Add and remove elements as needed (grow and shrink)
ArrayLists

- To use this class, you must first import it at the top of your file:

```java
import java.util.ArrayList;
```

- ArrayList's declaration and instantiation syntax looks wonky

```java
ArrayList<E> list_name;
list_name = new ArrayList<>();
```

E is the class name for the type of elements we will store.

- To create an ArrayList of Integers, we use:

```java
ArrayList<Integer> peoplesAge = new ArrayList<>();
```

- To create an ArrayList of Dice

```java
ArrayList<Die> diceCollection = new ArrayList<>();
```

**Note:**

- Use ArrayList<Double> to store doubles
- Use ArrayList<Boolean> to store booleans
- Use ArrayList<Integer> to store ints
Useful ArrayList Methods

Hey, these were all specified by the List interface!!!

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public ArrayList&lt;E&gt;()</td>
<td>Constructor. Creates a new ArrayList that holds objects of type E</td>
</tr>
<tr>
<td>public boolean add(E e)</td>
<td>Inserts element e to end of this list</td>
</tr>
<tr>
<td>public boolean add(int index, E e)</td>
<td>Inserts element e at the specified position in the list</td>
</tr>
<tr>
<td>public void clear()</td>
<td>Removes all elements from list</td>
</tr>
<tr>
<td>public boolean contains(E e)</td>
<td>Searches for element e in list, returns true if found, false otherwise</td>
</tr>
<tr>
<td>public E get(int index)</td>
<td>Returns the element at given index</td>
</tr>
<tr>
<td>public int indexOf(E e)</td>
<td>Searches for e in the list, returns the index of the first occurrence if found, or -1 if not found.</td>
</tr>
<tr>
<td>public E remove(int index)</td>
<td>Removes the element at given index. Returns the deleted element</td>
</tr>
<tr>
<td>public E set(int index, E e)</td>
<td>Replaces the element at the specified index.</td>
</tr>
<tr>
<td>public int size()</td>
<td>Returns the number elements in the current list</td>
</tr>
<tr>
<td>public String toString()</td>
<td>Returns the String representation of the current list</td>
</tr>
</tbody>
</table>

For the full ArrayList API: [http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)
ArrayList Internals

List<String> dwarves = new ArrayList<>();
ArrayList Internals

List<String> dwarves = new ArrayList<>();
dwarves.add("Bashful");
dwarves.add("Jumpy");
ArrayList Internals

List<String> dwarves = new ArrayList<>();
dwarves.add("Bashful");
dwarves.add("Jumpy");
dwarves.add(1, "Awful");
ArrayList Internals

```java
List<String> dwarves = new ArrayList<>();
dwarves.add("Bashful");
dwarves.add("Jumpy");
dwarves.add(1, "Awful");
dwarves.remove(0);
```
ArrayList Internals

List<String> dwarves = new ArrayList<>();
dwarves.add("Bashful");
dwarves.add("Jumpy");
dwarves.add(1, "Awful");
dwarves.remove(0);
dwarves.set(0, "Doc");

```
List<String> dwarves = new ArrayList<>();
dwarves.add("Bashful");
dwarves.add("Jumpy");
dwarves.add(1, "Awful");
dwarves.remove(0);
dwarves.set(0, "Doc");
```
ArrayList Internals

List<String> dwarves = new ArrayList<>();
dwarves.add("Bashful");
dwarves.add("Jumpy");
dwarves.add(1, "Awful");
dwarves.remove(0);
dwarves.set(0, "Doc");
dwarves.add("Snorty");
dwarves.add("Sneezy");
dwarves.add("Coughy");
dwarves.add("Snoozy");
dwarves.add("Grumpy");
dwarves.add("Dopey");
dwarves.add("Happy");
dwarves.add("Frumpy");
ArrayList Internals

List<String> dwarves = new ArrayList<>();
dwarves.add("Bashful");
dwarves.add("Jumpy");
dwarves.add(1, "Awful");
dwarves.remove(0);
dwarves.set(0, "Doc");
dwarves.add("Snorty");
dwarves.add("Sneezy");
dwarves.add("Coughy");
dwarves.add("Snoozy");
dwarves.add("Grumpy");
dwarves.add("Dopey");
dwarves.add("Happy");
dwarves.add("Frumpy");
dwarves.add("Jittery");

size: 11
capacity: 20
Details of ArrayList

- Some details you can now appreciate:
  - Can also instantiate like this: `List<E> my_list = new ArrayList<>();`
  - Initially `my_list` is "empty," but is backed by an underlying array
    - So it takes up storage even though it's "empty"
  - Built using arrays
    - Unless otherwise specified when constructing, has an initial capacity of 10
  - Size reallocation
    - Can add 11th element, but not before array size doubles automatically!
    - In fact, will always double capacity when adding new element would exceed capacity

- Let's try writing our own ArrayList to see how it's really implemented!
  (Code on course page)
Aside: Generic Types

- But the real ArrayList can hold *any type* of object
  - You just have to specify it in the angle brackets
  - Examples:

```java
List<String> dwarves = new ArrayList<>();
List<Integer> my_fave_numbers = new ArrayList<>();
```

- Let's talk generics. A simple example:
  - A class Pair that manages a pair of the same objects
Let's update our ArrayList to support generics!
Analysis of ArrayList

- Interested in analyzing commonly used methods
  - get() and set() are $O(1)$
  - add(E new_item)
    - Always adds to tail of the list: $O(1)$
    - May need to reallocate space: Seems like $O(n)$, but actually amortized $O(1)$
      - Reallocation every $n$ inserts, allows us to insert $n$ more items with $O(1)$ cost
      - Averages out to $O(1)$
  - add(int index, E new_item)
    - May need to reallocate space: $O(1)$ amortized again, but...
    - May need to shift elements right. Worst case, shift all elements: $O(n)$
Analysis of ArrayList

- Interested in analyzing commonly used methods
  - `remove(int index)`
    - May need to shift n elements left: $O(n)$
  - `indexOf(E item)`
    - Linear search: $O(n)$
  - `toString()`
    - Need to traverse the entire array: $O(n)$

- [Implement these in lab...]
To Summarize...

<table>
<thead>
<tr>
<th>Signature</th>
<th>Average/Worst-Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public E get(int index)</td>
<td>(O(1))</td>
</tr>
<tr>
<td>public E set(int index, E item)</td>
<td>(O(1))</td>
</tr>
<tr>
<td>public int size()</td>
<td>(O(1))</td>
</tr>
<tr>
<td>public boolean add(E item)</td>
<td>(O(1)) amortized</td>
</tr>
<tr>
<td>public boolean add(int index, E item)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>public int indexOf(E item)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>public E remove(int index)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>public String toString()</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

Can we do better than linear time?
To Summarize... (Cont.)

- Use ArrayLists when...
  - Lists need to grow/shrink
  - Elements usually added to the tail
  - Elements can be accessed randomly
  - When certain elements need to be updated over time

- Don't use ArrayLists when...
  - Elements usually added to the head or random position
  - Element deletions are abundant
Outline

- Java's List Interface
- ArrayList
- Aside: Generics
  - Linked List
- Iterator
- Conclusion
Linked Lists

- Problem:
  - Insertion and deletion are slow with ArrayLists
    - Lots of shuffling required

- Insight:
  - All the shuffling is due to the usage of arrays as the backing store
  - What if we don't use arrays as the backing store?
    - Allow individual elements to "float about" in memory
      - Instead of being in contiguous order
    - Each element only knows the next element (singly-linked list)
    - Could this help with performance?
Linked Lists (Cont.)

- Class demo
  - Need five volunteers
  - Line up in front of class

- Demonstrate
  - ArrayList insertion/removal
  - LinkedList insertion/removal
A List Node Object

- A **Node** is what we'll call an individual element in a Linked List
  - Stores a generic data element
  - Stores a reference to the next Node

- A **SinglyLinkedList** manages the collection of Nodes
  - It only needs to know:
    - The head Node
    - The size of the linked list
The Nested Node\(<E>\) Class

```java
public class SinglyLinkedList\(<E>\) {

/** A Node is a building block of linked lists! */
private static class Node\(<E>\) {
    private E data;
    private Node\(<E>\) next;

    public Node\(E new_data\) {
        data = new_data;
        next = null;
    }

    public Node\(E new_data, Node\(<E>\) next_node\) {
        data = new_data;
        next = next_node;
    }
}
}
```
Aside: Nested Classes

- Outer class vs. inner (nested) class
  - Outer class must always be public
  - Inner class can be public, protected, private

- Visibility
  - Outer class can access any instance variable and method in inner class
    - Even if they're private!
Aside: Nested Classes

Visibility (cont.)

• If inner class is `private`:
  - Its instances can only be created from the outer class and from within itself!
  - Inner class can access all instance variables (even private ones!) in the outer class

• If inner class is `static`:
  - Inner class cannot access instance variables in the outer class
  - External classes can instantiate it without having created an object of inner class
    – When in doubt, make inner classes static.
Example Usage (from within SinglyLinkedList)

Node<String> brad = new Node<String>("Brad");
Node<String> david = new Node<String>("David");
Node<String> adam = new Node<String>("Adam");
Node<String> sam = new Node<String>("Sam");
Example Usage (from within SinglyLinkedList)

Node<String> brad = new Node<String>("Brad");
Node<String> david = new Node<String>("David");
Node<String> adam = new Node<String>("Adam");
Node<String> sam = new Node<String>("Sam");
brad.setNext(david);
david.setNext(adam);
adam.setNext(sam);
//traverse through the nodes!
Node<String> currNode = brad;
while (currNode != null) {
    System.out.println(currNode.toString());
    currNode = currNode.next;
}

Output:
Brad
//traverse through the nodes!
Node<String> currNode = brad;
while (currNode != null) {
    System.out.println(currNode.toString());
    currNode = currNode.next;
}

Output:
Brad
David
Example Usage (from within SinglyLinkedList)

```java
//traverse through the nodes!
Node<String> currNode = brad;
while (currNode != null) {
    System.out.println(currNode.toString());
    currNode = currNode.next;
}

Output:
Brad
David
Adam
```

Example Usage (from within SinglyLinkedList)

```java
//traverse through the nodes!
Node<String> currNode = brad;
while (currNode != null) {
    System.out.println(currNode.toString());
    currNode = currNode.next;
}
```

Output:
Brad
David
Adam
Sam
Example Usage (from within SinglyLinkedList)

```java
//traverse through the nodes!
Node<String> currNode = brad;
while (currNode != null) {
    System.out.println(currNode.toString());
    currNode = currNode.next;
}
```

Output:
Brad
David
Adam
Sam

![Diagram of linked list traversal]
Wouldn't it be nice to have a "wrapper class" that encapsulates nodes and methods for traversal, insertion, etc.?

We'll next define the `SinglyLinkedList<E>` Class

- Recall that it needs to know:
  - The head Node
  - The size of the linked list

- Again, it should implement the Java `List<E>` interface
  - But we don't have time for all of them, so we'll leave off the `implements` keyword
  - *(Open up BlueJ...)*
First, Some Helper Methods

- The following methods will make our lives easier later...
  - Some are `private` because we don't want them to be called from outside the class!

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public void addFirst(E item)</code></td>
<td>Adds given item to the head of the list</td>
</tr>
<tr>
<td><code>private void addAfter(Node&lt;E&gt; node, E item)</code></td>
<td>Adds given item after the given node</td>
</tr>
<tr>
<td><code>private E removeAfter(Node&lt;E&gt; node)</code></td>
<td>Removes the node after the given node. Returns the element that was removed, or null</td>
</tr>
<tr>
<td><code>private E removeFirst()</code></td>
<td>Removes the head node. Return the element that was removed, or null.</td>
</tr>
<tr>
<td><code>private Node&lt;E&gt; getNode(int index)</code></td>
<td>Returns the node at the specified position, or null if it doesn't exist.</td>
</tr>
</tbody>
</table>
Now Some Methods from the List Interface

- Now we can implement a few common methods from the `List` interface
  - Here's a refresher...
  - We'll implement them in the following order, too

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public E get(int index)</code></td>
<td>Returns the data in the element at position <code>index</code></td>
</tr>
<tr>
<td><code>public E set(int index, E item)</code></td>
<td>Stores a reference to <code>item</code> in the element at position <code>index</code>. Returns the data formerly at position <code>index</code></td>
</tr>
<tr>
<td><code>public int size()</code></td>
<td>Return the current size of the List</td>
</tr>
<tr>
<td><code>public boolean add(int index, E item)</code></td>
<td>Adds a reference to <code>item</code>, inserting it before the item at position <code>index</code></td>
</tr>
<tr>
<td><code>public boolean add(E item)</code></td>
<td>Adds a reference to <code>item</code> at the end of the List. Always returns true</td>
</tr>
<tr>
<td><code>public int indexOf(E item)</code></td>
<td>Searches for <code>target</code> and returns the position of the first occurrence, or -1 if it is not in the List</td>
</tr>
<tr>
<td><code>public E remove(int index)</code></td>
<td>Removes the entry formerly at position <code>index</code> and returns it</td>
</tr>
</tbody>
</table>
Time Complexity Analysis: get(..)

- **Worst and average case:** $O(n)$
  - Worse than ArrayList and arrays
  - Culprit: need to traverse list from head to find the element

```java
public E get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("index: " + index);
    }
    // get the node at the specified index, then return its data portion
    Node<E> node = getNode(index);
    return node.data;
}

private Node<E> getNode(int index) {
    if (index < 0 || index >= size) {
        return null;
    }
    Node<E> current = head; // start iterating from the head node
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current;
}
```
Time Complexity Analysis: set(.)

- Worst and average case: $O(n)$
  - Worse than ArrayList and arrays too
  - Culprit: need to traverse list from head to find the element
Time Complexity Analysis: \( \text{add}(\ . \ . \) )

- **Best case:**
  - Adding to the head: \( O(1) \)

- **Worst case:**
  - Adding to the tail: \( O(n) \)

- **Average case:**
  - Adding somewhere in the middle: \( O(n) \)

- **Culprit:** need to traverse list to find the place to add

- The complexity of \( \text{remove}() \) is exactly the same as \( \text{add}() \)
### Summary of SinglyLinkedList

<table>
<thead>
<tr>
<th>Signature</th>
<th>Average/Worst-Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public E get(int index)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>public E set(int index, E item)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>public int size()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public boolean add(E item)</td>
<td>$O(n)$ -- But we can bring this down to $O(1)$</td>
</tr>
<tr>
<td>public boolean add(int index, E item)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>public boolean addFirst(E item)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public int indexOf(E item)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>public E remove(int index)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>public String toString()</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Summary of SinglyLinkedListList<E>

- Generally speaking: Use **ArrayLists** when possible
  - Except for the following reasons

- Use **SinglyLinkedLists** when...
  - Elements usually **added** or **removed** from **toward the head** of the list
    - Optimization: Adding/removing the last element can be done in O(1)-time
    - In addition to head, keep a reference to the last element!
  - Lists need to grow/shrink
    - Sure, but ArrayLists handle that too
  - Many things using the list needs to walk all the elements anyway
    - *e.g.*, computing the average, print all the names
    - Sure, but walking all elements are fast with ArrayLists too
Don't use `SinglyLinkedLists` when...

- Elements usually added to a random position
- Deletions and updates are abundant
- Elements are usually randomly accessed
  - e.g., print the 10th name
- Space is a constraint
  - Each Node stores *two items*: reference to next Node and the data
Outline

- Java's List Interface
- ArrayList
- Aside: Generics
- Linked List
- Iterator
- Conclusion
Iterators

- Motivation: Consider the complexity of the loop

```java
List<Integer> my_list = new LinkedList<Integer>();

//
// add n numbers (code omitted)
//

//what's the complexity of the following loop?
int sum = 0;
for (int i = 0; i < my_list.size(); i++) {
    sum += my_list.get(i);
}
```

- Analysis: $O(n^2)$
  - Let's see why
Iterators (Cont.)

- Loop iteration (i == 0)
  - Cost: 1
  - Total cost: 1
Iterators (Cont.)

- Loop iteration (i == 1)
  - Cost: 2
  - Total cost: 1 + 2
Iterators (Cont.)

- Loop iteration (i == 2)
  - Cost: 3
  - Total cost: $1 + 2 + 3$
Iterators (Cont.)

- Loop iteration (i == 3)
  - Cost: 4
  - Total cost: $1 + 2 + 3 + 4$
Iterators (Cont.)

- Loop iteration (i == 4 == n)
  - Cost: 0 (terminated)
  - Total cost: $1 + 2 + 3 + 4 = 10$
Iterators (Cont.)

- Observation
  - We're just traversing all elements in the linked list
  - Seems silly to have to start over each time

- Insight
  - In real-life, we'd just put our finger on where we left off
  - Start from the finger position for next consecutive \texttt{get(...)}

- Important: Java Iterators
  - Gives us the "finger"
  - Allows for fast sequential access to retrieve the next element
  - In fact, the \texttt{List} interface requires its implementation!
The *Iterator*<E*> Interface

- Important methods listed, but the full API is found here:
  - [https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean hasNext()</td>
<td>Returns true if the underlying collection has more elements, false otherwise.</td>
</tr>
<tr>
<td>public E next()</td>
<td>Returns the next item in the collection, or throws NoSuchElementException when all elements have been accessed.</td>
</tr>
</tbody>
</table>
Iterator<E> Usage Example

```java
import java.util.Iterator;
import java.util.ListIterator;

//...

List<Integer> my_list = new SinglyLinkedList<Integer>();

//
// add n numbers (code omitted)
//

int sum = 0;
Iterator<Integer> itr = my_list.iterator();
while (itr.hasNext()) {
    sum += itr.next();
}
```

What kind of data is held in the collection?

Check out Java's List interface. This method must be implemented.
Outline

- Java's List Interface
- ArrayList
- Aside: Generics
- Linked List
- Iterator
- Conclusion
Conclusion

- There are many types lists (all of which implement the same interface)
  - Others include:
    - Doubly-Linked List (Nodes also link to the previous node)
      - Important: Java's own `LinkedList` class is a doubly-linked list!
    - Circular Linked List (tail Node links to head node)

- Understand that certain situations call for different choices
  - Different lists have tradeoffs on performance
  - This is important to your maturing as a CSer!
Conclusion (Cont.)

- Understand why generics are necessary
  - And how to write classes with support of generic types

- Finally, know what Iterators are all about!
  - A standard Java interface for accessing elements in a data structure

- Next time: What are some useful data structures built on Lists?
Exam I
- 50 min, half-page notes, calculator allowed
- Review Monday (See review guide)
- Covers up to, and including O(f(n))

Hwk 3: get started !!

Today: List<E> interface and the ArrayList<E> implementation