Let's Write the Add Methods

- From the **MyList<E>** interface, we need:
  - `public void add(int index, E item):` insert item into indexed position  \( O(?) \)
  - `public boolean add(E item):` add item to tail  \( O(?) \)

- To implement those, we'll write a couple more *helper* methods:
  - `private void addFirst(E item)`:  
    - Create Node with given item stored in it  
    - Add it to the *head* of the list
  - `private void addAfter(Node<E> target, E item)`:  
    - Create Node with given item stored in it  
    - Add it after *target* node
### Linked List (LL) vs ArrayList

<table>
<thead>
<tr>
<th>Method</th>
<th>Linked List (LL)</th>
<th>ArrayList</th>
<th>Big Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>public E get(int index)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>LL Loses</td>
</tr>
<tr>
<td>public E set(int index, E item)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>LL Loses</td>
</tr>
<tr>
<td>public boolean add(E item)</td>
<td>$O(n)$</td>
<td>$O(1)$ amortized</td>
<td>LL Loses</td>
</tr>
<tr>
<td>public boolean add(int index, E item)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>LL wins if adding to head</td>
</tr>
<tr>
<td>public int indexOf(E item)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>Tie</td>
</tr>
<tr>
<td>public E remove(int index)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>LL wins if removing from head</td>
</tr>
<tr>
<td>public boolean remove(E item)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>Tie</td>
</tr>
</tbody>
</table>
Outline

- Java's List Interface
- ArrayList
  - Aside: Generics
- Linked List
  - Optimizations:
    - Tail
    - Iterator
    - Doubly Linked Nodes
- Conclusion
Slowness Problems (1)

- SinglyLinkedList is slow for the these kinds of common workloads:
  1. The 1-arg `add(E item)`

```java
/**
 * Adds [1, ..., n] to the list.
 * This method is slower than O(n)
 */
public MyList<Integer> populate(int n) {
    MyList<Integer> list = new SinglyLinkedList<>();
    for (int i = 0; i < n; i++) {
        list.add(i + 1);
    }
    return list;
}
```

- (Problem is that `add()` must traverse from head to the tail each time!)
  - T(n) for populate?
Currently we need to traverse the list just to find the *tail node*.

- Maybe a SinglyLinkedList should point to its head *and* tail nodes!

Update our code!

```java
public class SinglyLinkedList<E> implements MyList<E> {
    private Node<E> head;  /** head of the list */
    private Node<E> tail;  /** tail of the list */
    private int size;

    public SinglyLinkedList() {
        this.head = this.tail = null;
        this.size = 0;
    }

    // TODO: update add and remove to deal with tail
}
```
**Can we do better?**

<table>
<thead>
<tr>
<th>Method</th>
<th>Linked List (LL)</th>
<th>ArrayList</th>
<th>Big Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>public E get(int index)</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>LL Loses</td>
</tr>
<tr>
<td>public E set(int index, E item)</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>LL Loses</td>
</tr>
<tr>
<td>public boolean add(E item)</td>
<td>(O(1)) with tail</td>
<td>(O(1)) amortized</td>
<td>Tie / Win (no reallocation)</td>
</tr>
<tr>
<td>public boolean add(int index, E item)</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>LL wins if adding to head</td>
</tr>
<tr>
<td>public int indexOf(E item)</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>Tie</td>
</tr>
<tr>
<td>public E remove(int index)</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>LL wins if removing from head</td>
</tr>
<tr>
<td>public boolean remove(E item)</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>Tie</td>
</tr>
</tbody>
</table>
Outline

- Java's List Interface
- ArrayList
  - Aside: Generics
- Linked List
  - Optimizations:
    - Tail
    - Iterator
    - Doubly Linked Nodes
- Conclusion
Iterators

- **Motivation:** get and set() are slow

```java
MyList<Integer> list = new SinglyLinkedList<>();

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

// the complexity of the following loop?
int sum = 0;
for (int i = 0; i < list.size(); i++) {
    sum += list.get(i); // get() isn't O(1)
}
```

- **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
Loop iteration (i == 2)

- Cost: 3
- Total cost: $1 + 2 + 3$
Iterators (Cont.)

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

![Diagram of linked list with loop iteration example]
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
  - Like add-to-tail, it 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 `get(...)`
Summary of SinglyLinkedList<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, also keep a reference to the tail!
Don't use `SinglyLinkedLists` when...

- Deletions and updates are abundant in your application
- Items are usually randomly accessed
  - e.g., get a random item from the list
  - e.g., add new item to the middle of list
  - e.g., delete an item from the middle of the list
  - What is the complexity of binary search over a LinkedList?
    - Hint: it's not $O(\log n)$!
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 links to head)

- 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?