Lists & Arrays

Topical Review Session 02

Lists

- Arrays are fixed size in Java, since they can only hold as many elements as they are initialized with.

```
int[] arr = new int[15];
```

  In this case, arr can never hold more than 15 elements, and cannot change this in the future
- List is an interface which describes a new data structure - one where the size is flexible.

```
public interface List<T> { // There are more functions in the actual interface
     void add(T item);     // but these are the big ones.
     T get(int idx);
     void remove(int idx);
     int size();
}
```

Linked Lists

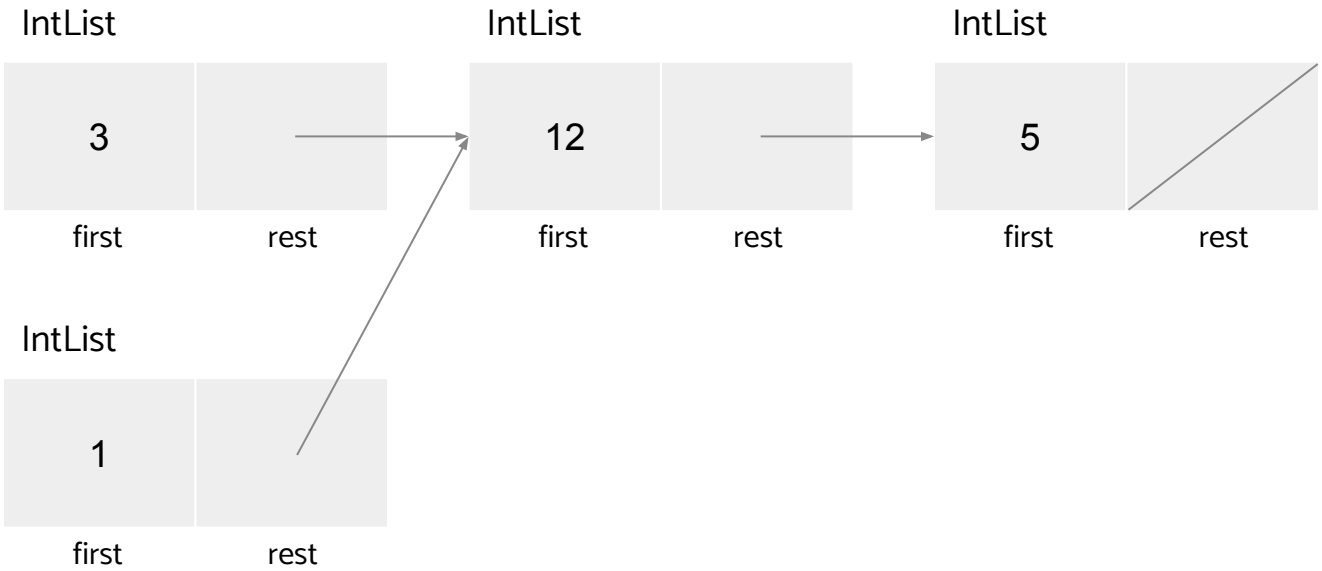
IntLists

- Our first attempt at creating a LinkedList
- It has a recursive structure - each IntList points to another IntList in a modular way

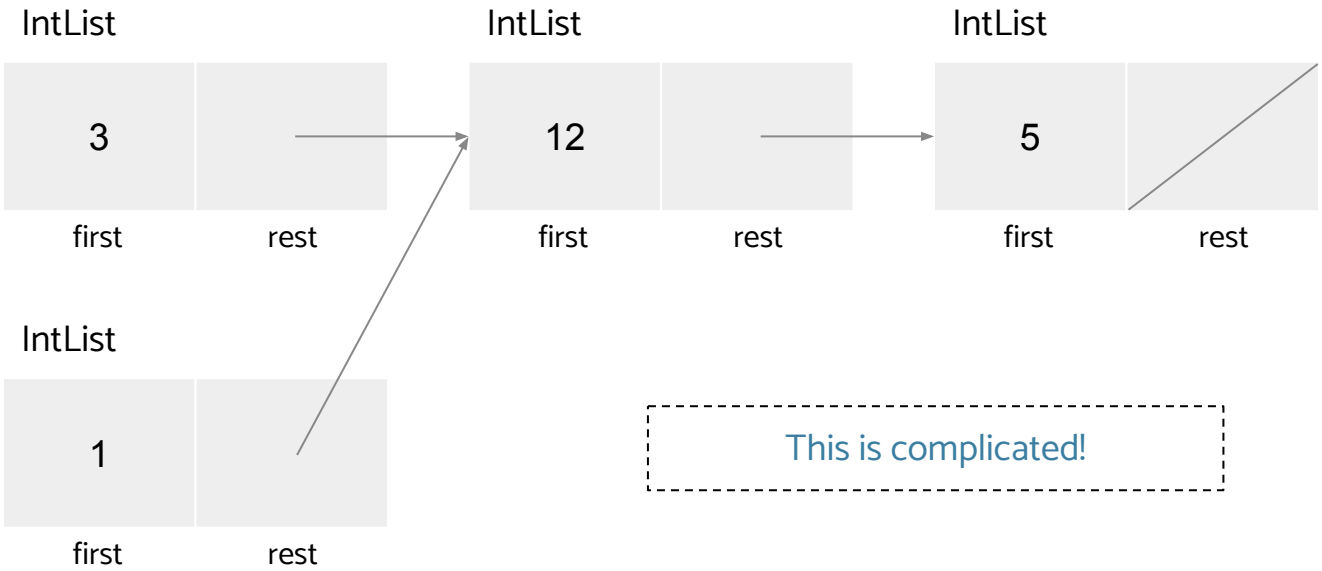
```
public IntList {
     int first;
     IntList rest;
     public IntList(int first, IntList rest) {
          this.first = first;
          this.rest = rest;
     }
}
```

- Functions written with IntLists can be a little messy since the user has to directly interact with each node of the list.

IntLists


IntList
3
first
rest
IntList
12
first
rest
IntList
5
first
rest
IntList
1
first
rest

IntLists


IntList
3
first
rest
IntList
12
first
rest
IntList
5
first
rest
IntList
1
first
rest
This is complicated!

SLLists

- What if we try to hide the complexity of IntLists in a **wrapper**, which handles all the operations for us?

```
public class SLList<T> {
     Node first;
     int size;

     private class Node { ... }
}
```

- The SLList class has an internal Node class that is very similar to an IntList - each Node contains a single value and points to the next item on the list
- The SLList class itself contains functions from the List interface so that a programmer can interact with the list rather than directly with individual nodes, which means it encourages **abstraction**

Node Class

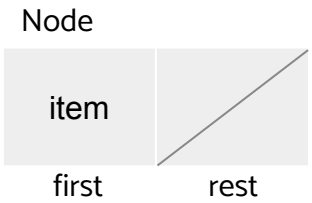
```
private class Node { // Class needs to be private so that outside users can't mess with it
     T item; // Node can hold any type now, as long as the whole list is consistent
     Node next;

     public Node(T item, Node next) {
          this.item = item;
          this.next = next;
     } // This class looks mostly identical to IntList
}
```

```
Node n = new Node(item, null);
```

Node Class

```
private class Node { // Class needs to be private so that outside users can't mess with it
     T item; // Node can hold any type now, as long as the whole list is consistent
     Node next;

     public Node(T item, Node next) {
          this.item = item;
          this.next = next;
     } // This class looks mostly identical to IntList
}
```

- `Node n = new Node(item, null);`



SLList Class

```
public class SLList<T> {
     private class Node {...}
     Node first;
     int size;
     public void addFirst(T item) {
          this.first = new Node(item, first);
          this.size += 1;
     }
}
```

- `SLList<Integer> list = new SLList<>();`

SLList Class

```
public class SLList<T> {
     private class Node {...}
     Node first;
     int size;
     public void addFirst(T item) {
          this.first = new Node(item, first);
          this.size += 1;
     }
}
```

- `SLList<Integer> list = new SLList<>();`

SLList

SLList Class

```
public class SLList<T> {
     private class Node {...}
     Node first;
     int size;
     public void addFirst(T item) {
          this.first = new Node(item, first);
          this.size += 1;
     }
}
```

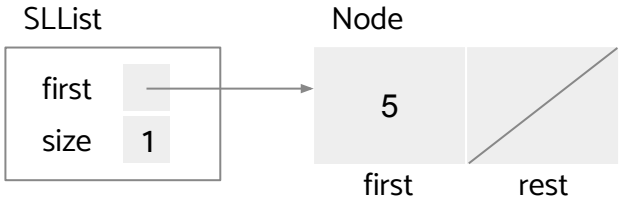
- `list.addFirst(5);`

SLList

SLList Class

```
public class SLList<T> {
     private class Node {...}
     Node first;
     int size;
     public void addFirst(T item) {
          this.first = new Node(item, first);
          this.size += 1;
     }
}
```

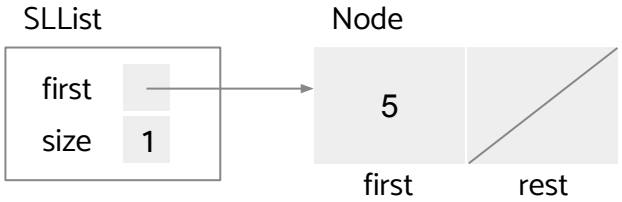
- `list.addFirst(5);`



SLList Class

```
public class SLList<T> {
     private class Node {...}
     Node first;
     int size;
     public void addFirst(T item) {
          this.first = new Node(item, first);
          this.size += 1;
     }
}
```

- `list.addFirst(12);`



SLList Class

```
public class SLList<T> {
     private class Node {...}
     Node first;
     int size;
     public void addFirst(T item) {
          this.first = new Node(item, first);
          this.size += 1;
     }
}
```

- `list.addFirst(12);`

Sentinel Nodes

- When our list is empty, we need an extra check for a lot of functions to avoid null pointer exceptions:

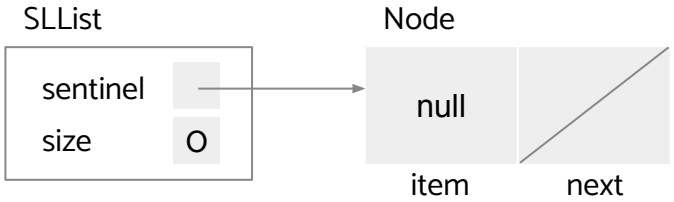
```
public void removeLast() {
     if (first == null) {
          return;
     }
     Node p = first;
     while (p.rest != null) { ... }
     // do other stuff which applies to every case except when the list is empty
}
```

- We like to avoid complexity, so how can we make all SLLists behave the same?
- **Sentinel nodes** are "secret" nodes we add to our implementation invisible to the outside user so that we can perform the same checks on an empty list that we perform on any other list
- No need for that initial null check everywhere!

SLList Class with Sentinel

```
public class SLList<T> {
     private class Node {...}
     Node sentinel;
     int size;
     public SLList() {
          this.sentinel = new Node(null, null);
          this.size = 0;
     }
}
```

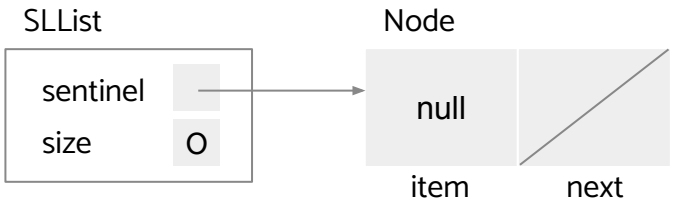
- `SLList<Integer> list = new SLList<>();`



SLList Class with Sentinel

```
public class SLList<T> {
     private class Node {...}
     Node sentinel;
     int size;
     public SLList() {
          this.sentinel = new Node(null, null);
          this.size = 0;
     }
}
```

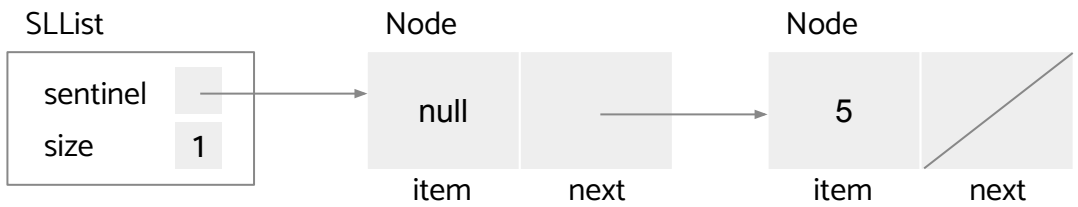
- `list.addFirst(5);`



SLList Class with Sentinel

```
public class SLList<T> {
     private class Node {...}
     Node sentinel;
     int size;
     public SLList() {
          this.sentinel = new Node(null, null);
          this.size = 0;
     }
}
```

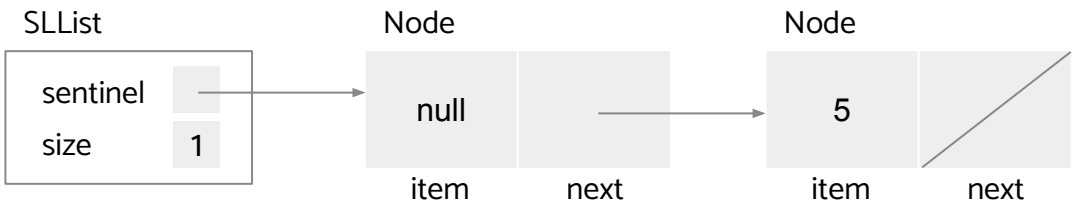
- `list.addFirst(5);`



SLList Class with Sentinel

```
public class SLList<T> {
     private class Node {...}
     Node sentinel;
     int size;
     public SLList() {
          this.sentinel = new Node(null, null);
          this.size = 0;
     }
}
```

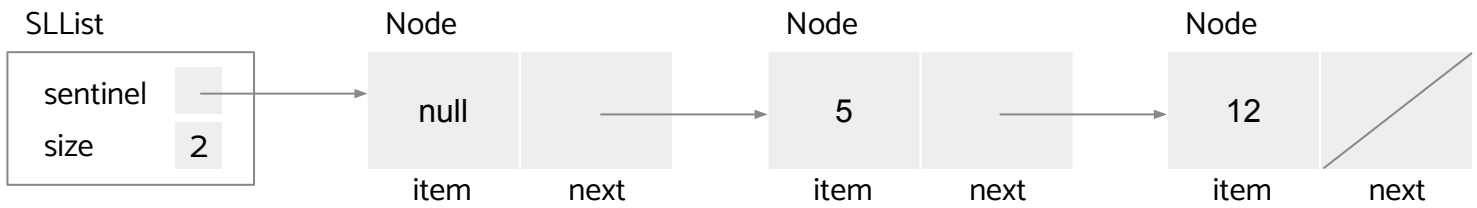
- `list.addLast(12);`



SLList Class with Sentinel

```
public class SLList<T> {
     private class Node {...}
     Node sentinel;
     int size;
     public SLList() {
          this.sentinel = new Node(null, null);
          this.size = 0;
     }
}
```

- `list.addLast(12);`

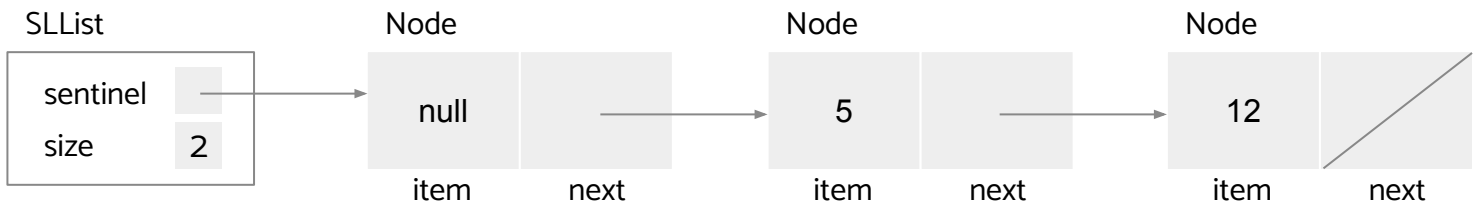


SLList Class with Sentinel

```
public class SLList<T> {
     private class Node {...}
     Node sentinel;
     int size;
     public SLList() {
          this.sentinel = new Node(null, null);
          this.size = 0;
     }
}
```

addLast will take increasingly long as the list gets longer :(

- `list.addLast(12);`



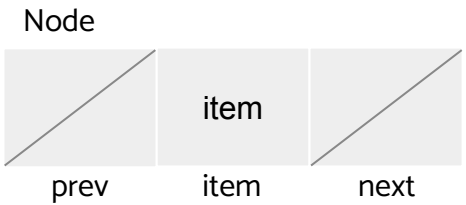
DLLists

- Our new problem is that inserting or removing from the end of a SLList takes a very long time as our list grows longer
- One solution: have a second pointer that points to the end of the list
    - Add is much faster
    - Since we don't have access to the second to last node remove is still very slow
- Better solution: have every Node point to both the previous Node and the next Node
    - Including the sentinel node, so we have fast access to the back
    - Now we have a kind of circular structure
    - Add & remove are both much faster

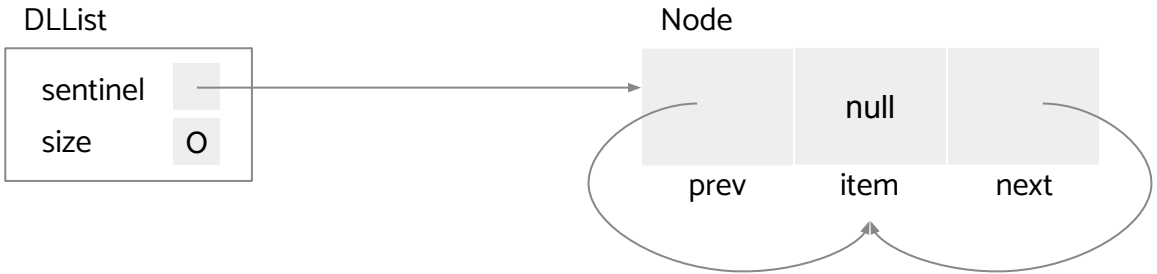
Updated Node Class

```
private class Node {
     T item;
     Node prev, next;
     public Node(T item, Node prev, Node next) {
          this.item = item;
          this.prev = prev;
          this.next = next;
     }
}
```

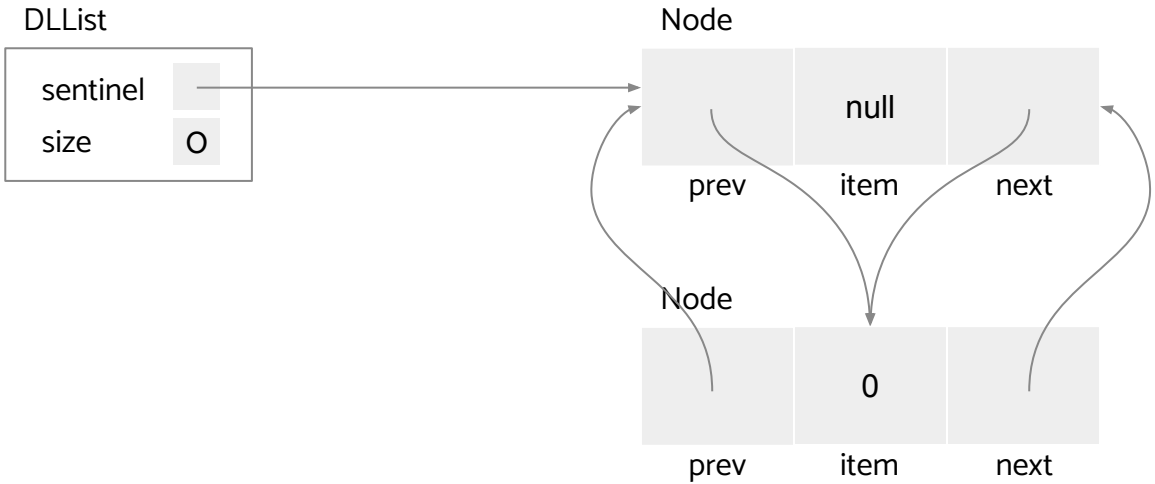
- `Node n = new Node(item, null, null);`



DLLists



DLLists


DLList
sentinel
size
0
Node
null
prev
item
next
Node
0
prev
item
next

Array Lists

It's Party Time

- Post Corona!
- People are vaccinated and Berkeley is back to being lit!
- You decide to throw a hugeeeeeeeee party with N people
    - Every 15 minutes, you give away free popcorn to the nth person, given that person number 0 <= n <= N is chosen randomly (zeroth person is the first person to enter the party, the Nth person is the latest)
    - Don't know how many people are going to show up but want to keep track of those who have already received popcorn
    - Note: N keeps increasing as more and more people show up. No one's leaving the party because it's that good.

What Data Structure Should We Use?

LinkedList

- Can change size dynamically so adding people won't be a problem
- Finding the nth person (to give the popcorn to) will take a long time once we have a lot of people

Arrays

- Size is stuck, so resizing will require copying all the contents each time we add a person + it will be annoying to implement
- Finding the nth person (to give the popcorn to) will be super fast since arrays take no time to access elements

ArrayLists

- Key takeaway: fast lookup **and** insertion
- Implement same functionality as LinkedLists since it implements the List interface

```
public class ArrayList<T> implements List<T> {
     private T[] arr;
     int size;

     private void resize() { ... }
}
```

- Contains an underlying array to keep track of all the data which must be resized occasionally
- Allows you to access any index in constant time (like an Array)
- **Abstraction** at play again - hides all the complexity of indexing and resizing from the outside user

Naive ArrayList

```
public class ArrayList {
     private int[] items; // Uses array as backend
     private int size; // Represents number of items added - not size of underlying array
     public ArrayList() {
          this.items = new int[100];
          this.size = 0;
     }
     public void addLast(int x) { // Can add to the end just like LinkedList
          items[size] = x;
          size += 1;
     }
     public void removeLast() {
          items[size - 1] = null; // We need to set this to null so that Java can garbage
          size -= 1;              // collect by removing the pointer to what was there
     }
     public int get(int i) { return items[i]; } // So fast!
}
```

Naive ArrayList

```
public class ArrayList {
     private int[] items;
     private int size;
     public ArrayList() {
          this.items = new int[100];
          this.size = 0;
     }
     public void addLast(int x) {
          items[size] = x;
          size += 1;
     }
     public void removeLast() {
          items[size - 1] = null;
          size -= 1;
     }
     public int get(int i) { return items[i]; } // So fast!
}
```

What happens when you add the 101st element?

Need to resize our underlying array by some factor.
We will need to make some changes to addLast.

Resizing

- Idea: check for size when adding and removing elements
- Resize internal array to twice/half the current array size
- Will only have to resize every so often
- Let's proceed with assumption that `ArrayList` begins with array of size 8

Resize

```
int[] piArr = new int[] {3, 1, 4, 1, 5, 9, 2, 6, 5, 3}; // 10 elements
ArrayList<Integer> piList = new ArrayList<>();

for (int i = 0; i < 8; i++) {
     piList.addLast(piArr[i]); // Let's populate our ArrayList without making it resize
}
```

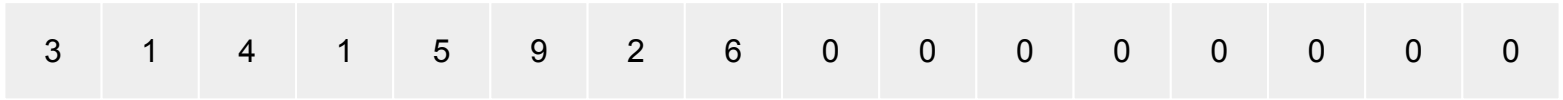
Resize

```
int[] piArr = new int[]{3, 1, 4, 1, 5, 9, 2, 6, 5, 3}; // 10 elements
ArrayList<Integer> piList = new ArrayList<>();

for (int i = 0; i < 8; i++) {
     piList.addLast(piArr[i]);
}

piList.addLast(piArr[8]); // Call will first check whether size == items.length
                          // It is, so we will commence resizing
```

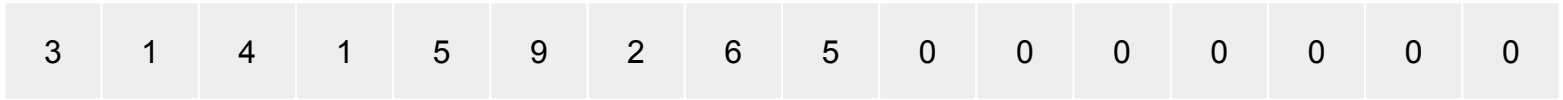
Resize

1. Make a new array of size `size * FACTOR`, in this case factor is 2:

2. Copy over the old contents:



3. Add the new element to the end:



Linked Lists vs Array Lists

LinkedLists

Pros

- Can quickly add and remove items from ends of the list
- Space efficient - no space wasted on nodes that don't contain anything
- No need for any sort of resizing

Cons

- Accessing items can be very slow since you need to traverse the full list
- Each node is an independent data structure - will take up some extra space versus primitives (like arrays)

ArrayLists

Pros

- Can access any value very fast due to underlying array structure
- Since underlying structure is a primitive type, it takes up less space than having a bunch of nodes when there are large numbers of items

Cons

- Resizing can be unintuitive and difficult to implement
- Significantly slower when you encounter resizing
- Can take up a lot of unnecessary space (memory) after resizing with empty elements

When To Use Which?

LinkedLists

- Accessing, adding, and removing the first/last values
- Adding/removing values in arbitrary positions that are not at the end (slightly better performance)

Example: An alphabetical list as applicants submit applications for a job offer

ArrayLists

- Frequent indexing into values that are not first/last
- Really large volume of values (lots of nodes can take up a lot of space)

Example: A large library catalog where each book is tracked with an index number

...More examples on the worksheet!