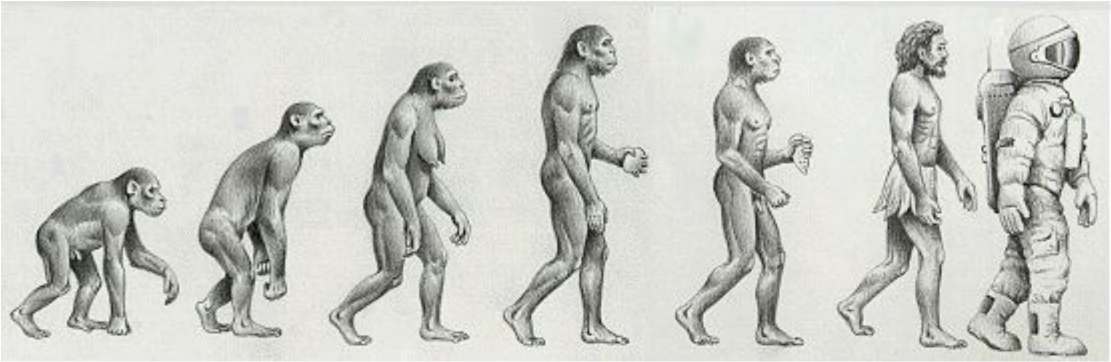


Lecture 4 (Lists 2)

SLLists, Nested Classes, Sentinel Nodes

CS61B, Fall 2024 @ UC Berkeley

Slides credit: Josh Hug

Getting Started: IntList → IntNode

Lecture 4, CS61B, Fall 2024

Getting Started

- `IntList` → `IntNode`
- Creating the SLList Class
- `addFirst` and `getFirst`
- SLLists vs. IntLists

Syntax Improvements:

- Access Control: Public vs. Private
- Nested Classes

`addLast` and `size`

- Creating `addLast` and `size`
- `size` Efficiency, caching

The Empty List

- `addLast` Bug
- Sentinel Nodes
- Invariants

Last Time in 61B: Recursive Implementation of a List

```java
public class IntList {
   public int first;
   public IntList rest;

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

While functional, “naked” linked lists like the one above are hard to use.

- Users of this class are probably going to need to know references very well, and be able to think recursively. Let’s make our users’ lives easier.

Improvement #1: Rebranding and Culling

```java
public class IntNode {
   public int item;
   public IntNode next;

   public IntNode(int i, IntNode n) {
      item = i;
      next = n;
   }

}
```

IntNode is now dumb, has no methods. We will reintroduce functionality in the coming slides.

Not much of an improvement obviously, but this next weird trick will be more impressive.

Creating the SLList Class

Lecture 4, CS61B, Fall 2024

Getting Started

- `IntList` → `IntNode`
- **Creating the SLList Class**
- `addFirst` and `getFirst`
- SLLists vs. IntLists

Syntax Improvements:

- Access Control: Public vs. Private
- Nested Classes

`addLast` and `size`

- Creating `addLast` and `size`
- `size` Efficiency, caching

The Empty List

- `addLast` Bug
- Sentinel Nodes
- Invariants

Coding Demo: Bureaucracy

SLList.java

```
/** An SLList is a list of integers, which hides the terrible truth
  * of the nakedness within. */
public class SLList {









}
```

Coding Demo: Bureaucracy

SLList.java

```java
/** An SLList is a list of integers, which hides the terrible truth
  * of the nakedness within. */
public class SLList {
   public IntNode first;









}
```

Coding Demo: Bureaucracy

SLList.java

```
/** An SLList is a list of integers, which hides the terrible truth
  * of the nakedness within. */
public class SLList {
   public IntNode first;

   public SLList(int x) {

   }





}
```

Coding Demo: Bureaucracy

SLList.java

```java
/** An SLList is a list of integers, which hides the terrible truth
  * of the nakedness within. */
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }







}
```

Coding Demo: Bureaucracy

SLList.java

```
/** An SLList is a list of integers, which hides the terrible truth
  * of the nakedness within. */
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public static void main(String[] args) {


   }



}
```

Coding Demo: Bureaucracy

SLList.java

```
/** An SLList is a list of integers, which hides the terrible truth
  * of the nakedness within. */
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public static void main(String[] args) {
      /** Creates a list of one integer, namely 10 */
      SLList L = new SLList(10);
   }



}
```

Improvement #2: Bureaucracy

```
public class IntNode {
   public int item;
   public IntNode next;

   public IntNode(int i, IntNode n) {
      item = i;
      next = n;
   }

}
```

```
IntList X = new IntList(10, null);
SLList Y = new SLList(10);
```

SLList is easier to instantiate (no need to specify null), but we will see more advantages to come.

```
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }
   ...
}
```

Next: Let’s add addFirst and getFirst methods to SLList.

IntNode is now dumb, has no methods.

addFirst and getFirst

Lecture 4, CS61B, Fall 2024

Getting Started

- `IntList` → `IntNode`
- Creating the SLList Class
- **`addFirst` and `getFirst`**
- SLLists vs. IntLists

Syntax Improvements:

- Access Control: Public vs. Private
- Nested Classes

`addLast` and `size`

- Creating `addLast` and `size`
- `size` Efficiency, caching

The Empty List

- `addLast` Bug
- Sentinel Nodes
- Invariants

Coding Demo: addFirst and getFirst

SLList.java

```
public class SLList {
   public IntNode first;









}
```

Coding Demo: addFirst and getFirst

SLList.java

```java
public class SLList {
   public IntNode first;

   /** Adds x to the front of the list. */
   public void addFirst(int x) {

   }
}
```

Coding Demo: addFirst and getFirst

SLList.java

```
public class SLList {
   public IntNode first;

   /** Adds x to the front of the list. */
   public void addFirst(int x) {
      first = new IntNode(x, first);
   }









}
```

This is how we added to the front of an IntList in the previous lecture.

Coding Demo: addFirst and getFirst

SLList.java

```java
public class SLList {
    public IntNode first;

    /** Adds x to the front of the list. */
    public void addFirst(int x) {
        first = new IntNode(x, first);
    }

    /** Returns the first item in the list. */
    public int getFirst() {

    }




}
```

Coding Demo: addFirst and getFirst

SLList.java

```
public class SLList {
   public IntNode first;

   /** Adds x to the front of the list. */
   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   /** Returns the first item in the list. */
   public int getFirst() {
      return first.item;
   }




}
```

Coding Demo: addFirst and getFirst

SLList.java

```
public class SLList {
    public IntNode first;

    public static void main(String[] args) {
        SLList L = new SLList(15);
        L.addFirst(10);
        L.addFirst(5);
        System.out.println(L.getFirst()); // should print 5
    }









}
```

The Basic SLList and Helper IntNode Class

```java
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   public int getFirst() {
      return first.item;
   }
}
```

```java
public class IntNode {
   public int item;
   public IntNode next;

   public IntNode(int i, IntNode n) {
      item = i;
      next = n;
   }

}
```

Example usage:

```java
SLList L = new SLList(15);
L.addFirst(10);
L.addFirst(5);
int x = L.getFirst();
```

SLLists vs. IntLists

Lecture 4, CS61B, Fall 2024

SLLists vs. IntLists

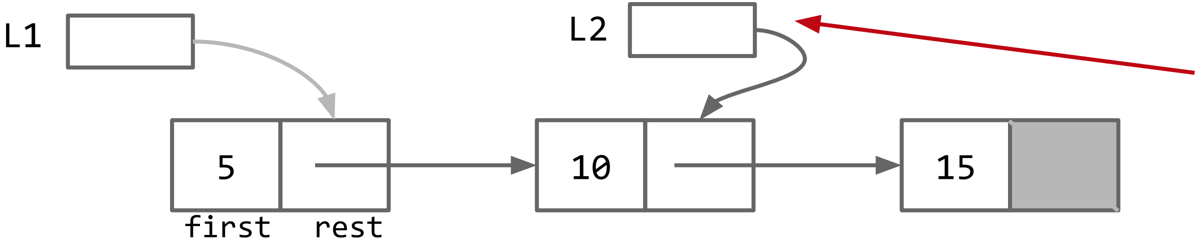
```java
SLList L = new SLList(15);
L.addFirst(10);
L.addFirst(5);
int x = L.getFirst();
```

```java
IntList L = new IntList(15, null);
L = new IntList(10, L);
L = new IntList(5, L);
int x = L.first;
```

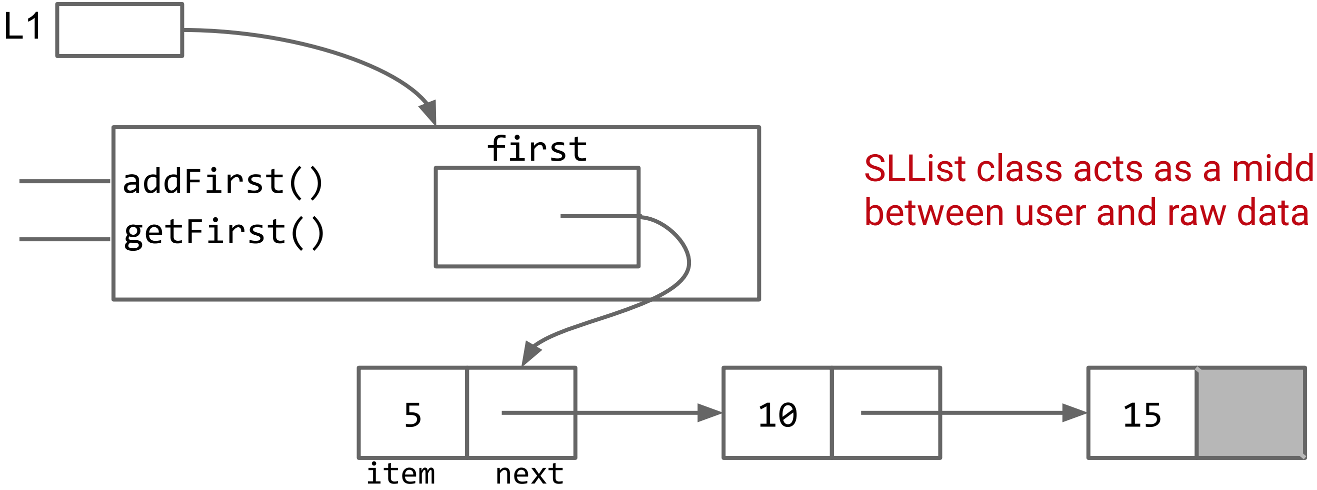
While functional, “naked” linked lists like the `IntList` class are hard to use.

- Users of `IntList` are need to know Java references well, and be able to think recursively.
- `SLList` is much simpler to use. Simply use the provided methods.
- Why not just add an `addFirst` method to the `IntList` class? Turns out there is no efficient way to do this. Try it out and you’ll see it’s hard (and inefficient).

Naked Linked Lists (IntList) vs. SLLists



Naked recursion: Natural for IntList user to have variables that point to the middle of the IntList.



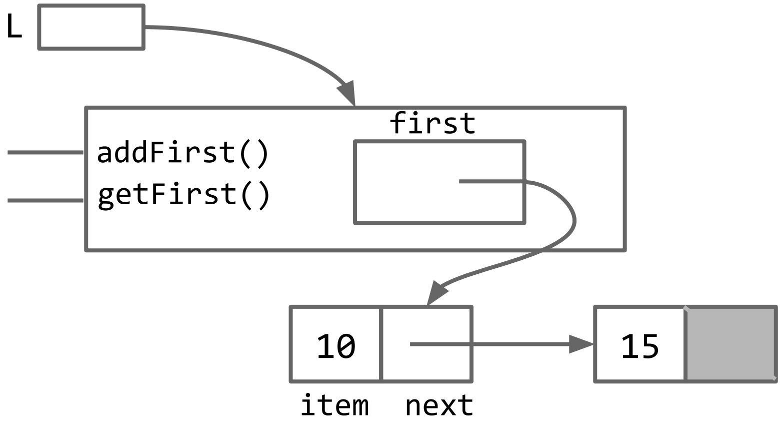
SLList class acts as a middle man between user and raw data structure.

Access Control: Public vs. Private

Lecture 4, CS61B, Fall 2024

The SLList So Far

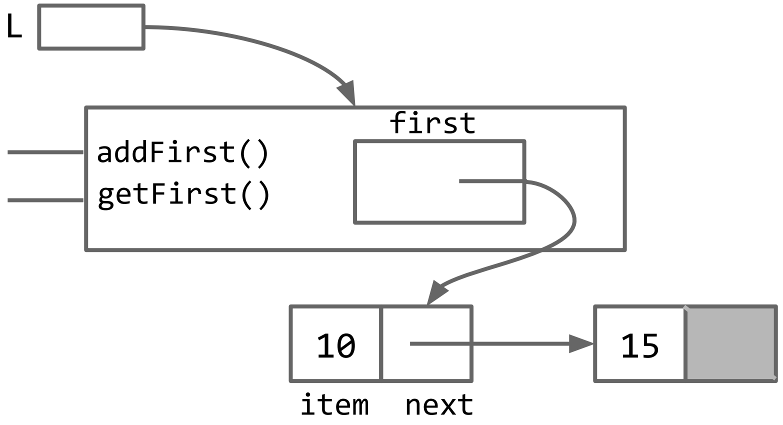
```
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   ...
}
```




```
SLList L = new SLList(15);
L.addFirst(10);
```

A Potential SLList Danger

```
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   ...
}
```

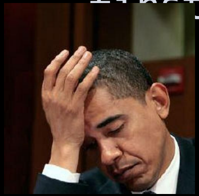


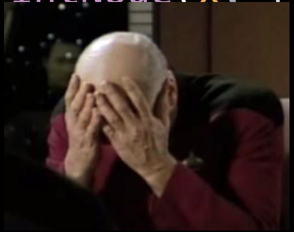
Users of our class might be tempted to try to manipulate our secret IntNode directly in uncouth ways!

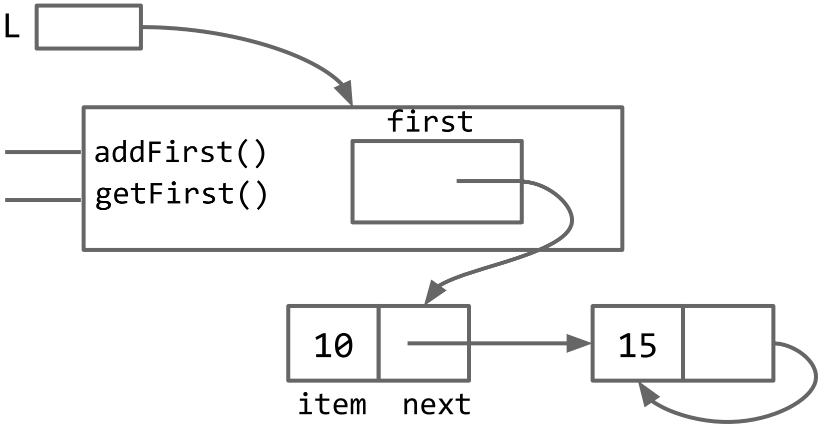
```
SLList L = new SLList(15);
L.addFirst(10);
L.first.next.next = L.first.next;
```

A Potential SLList Danger

```
public class SLList {
    public IntNode first;

    publ            x) {
       f            tNode
    }

    public void addFirst(int x) {
       first = new IntNode(x, first);

}
```








```
SLList L = new SLList(15);
L.addFirst(10);
L.first.next.next = L.first.next;
```

Users of our class might be tempted to try to manipulate our secret IntNode directly in uncouth ways!

Access Control

```
public class SLList {
   public IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   ...
}
```

We can prevent programmers from making such mistakes with the `private` keyword.

Improvement #3: Access Control

```
public class SLList {
   private IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   ...
}
```

Use the **private** keyword to prevent code in other classes from using members (or constructors) of a class.

```
SLList L = new SLList(15);
L.addFirst(10);
L.first.next.next = L.first.next;
```

```
jug ~/Dropbox/61b/lec/lists2
$ javac SLListUser.java
SLListUser.java:8: error: first has private access in SLList
        L.first.next.next = L.first.next;
```

Why Restrict Access?

Hide implementation details from users of your class.

- Less for user of class to understand.
- Safe for you to change private methods (implementation).

Car analogy:

- **Public**: Pedals, Steering Wheel   **Private**: Fuel line, Rotary valve

- Despite the term ‘access control’:
    - Nothing to do with protection against hackers, spies, and other evil entities.

Nested Classes

Lecture 4, CS61B, Fall 2024

Getting Started

- `IntList` → `IntNode`
- Creating the SLList Class
- `addFirst` and `getFirst`
- SLLists vs. IntLists

Syntax Improvements:

- Access Control: Public vs. Private
- **Nested Classes**

`addLast` and `size`

- Creating `addLast` and `size`
- `size` Efficiency, caching

The Empty List

- `addLast` Bug
- Sentinel Nodes
- Invariants

Improvement #4: Nested Classes

Can combine two classes into one file pretty simply.

```
public class SLList {
   public class IntNode {
      public int item;
      public IntNode next;
      public IntNode(int i, IntNode n) {
         item = i;
         next = n;
      }
   }

   private IntNode first;
   public SLList(int x) {
      first = new IntNode(x, null);
   } ...
```

Nested class definition.

Could have made `IntNode` a private nested class if we wanted.

Instance variables, constructors, and methods of `SLList` typically go below nested class definition.

Why Nested Classes?

Nested Classes are useful when a class doesn't stand on its own and is obviously subordinate to another class.

- Make the nested class private if other classes should never use the nested class.

In my opinion, probably makes sense to make `IntNode` a nested private class.

- Hard to imagine other classes having a need to manipulate `IntNodes`.
- If there was some hypothetical strange function like:
  ```
  public IntNode getFrontNode()
  ```
  Then we would need the `IntNode` class to be public.

Static Nested Classes

If the nested class never uses any instance variables or methods of the outer class, declare it static.

- Static classes cannot access outer class's instance variables or methods.
- Results in a minor savings of memory. See book for more details / exercise.

```
public class SLList {
    private static class IntNode {
        public int item;
        public IntNode next;
        public IntNode(int i, IntNode n) {
            item = i;
            next = n;
        }
    ...
}
```

We can declare IntNode static, since it never uses any of SLList's instance variables or methods.

Analogy: Static methods had no way to access "my" instance variables. Static classes cannot access "my" outer class's instance variables.

Unimportant note: For private nested classes, access modifiers are irrelevant.

Creating addLast and size

Lecture 4, CS61B, Fall 2024

Getting Started

- `IntList` → `IntNode`
- Creating the SLList Class
- `addFirst` and `getFirst`
- SLLists vs. IntLists

Syntax Improvements:

- Access Control: Public vs. Private
- Nested Classes

`addLast` and `size`

- **Creating `addLast` and `size`**
- `size` Efficiency, caching

The Empty List

- `addLast` Bug
- Sentinel Nodes
- Invariants

Adding More SLList Functionality

To motivate our remaining improvements, and to give more functionality to our `SLList` class, let's add:

- `.addLast(int x)`
- `.size()`

See study guide for starter code!

Recommendation: Try writing them yourself before watching how I do it.

Methods	Non-Obvious Improvements	
`addFirst(int x)`	#1	Rebranding: `IntList` → `IntNode`
`getFirst`	#2	Bureaucracy: `SLList`
	#3	Access Control: `public` → `private`
	#4	Nested Class: Bringing `IntNode` into `SLList`

Coding Demo: addLast

SLList.java

```
public class SLList {
   private IntNode first;

   /** Adds an item to the end of the list. */
   public void addLast(int x) {








   }


}
```

Coding Demo: addLast

SLList.java

```
public class SLList {
   private IntNode first;

   /** Adds an item to the end of the list. */
   public void addLast(int x) {
      IntNode p = first;






   }


}
```

Coding Demo: addLast

SLList.java

```
public class SLList {
   private IntNode first;

   /** Adds an item to the end of the list. */
   public void addLast(int x) {
      IntNode p = first;


      while (p.next != null) {

      }


   }


}
```

Coding Demo: addLast

SLList.java

```java
public class SLList {
   private IntNode first;

   /** Adds an item to the end of the list. */
   public void addLast(int x) {
      IntNode p = first;


      while (p.next != null) {
         p = p.next;
      }


   }


}
```

Coding Demo: addLast

SLList.java

```
public class SLList {
   private IntNode first;

   /** Adds an item to the end of the list. */
   public void addLast(int x) {
      IntNode p = first;

      /* Move p until it reaches the end of the list. */
      while (p.next != null) {
         p = p.next;
      }


   }


}
```

Coding Demo: addLast

SLList.java

```java
public class SLList {
    private IntNode first;

    /** Adds an item to the end of the list. */
    public void addLast(int x) {
        IntNode p = first;

        /* Move p until it reaches the end of the list. */
        while (p.next != null) {
            p = p.next;
        }

        p.next = new IntNode(x, null);
    }


}
```

Coding Demo: addLast

SLList.java

```
public class SLList {
   private IntNode first;

   public static void main(String[] args) {
      SLList L = new SLList(15);
      L.addFirst(10);
      L.addFirst(5);
      L.addLast(20);
      System.out.println(L.getFirst()); // should print 5
   }
}
```

Running this code doesn't actually prove that our addLast works. More about testing in a later lecture.

[Java visualizer]()

Coding Demo: size

SLList.java

```
public class SLList {
   private IntNode first;

   public int size() {

   }



}
```

Coding Demo: size

SLList.java

```
public class SLList {
   private IntNode first;

   public int size() {

   }




}
```

Writing a recursive size method is tricky, because SLList itself is not recursive.

The size method doesn't take in any arguments, so calling size recursively is strange. What is the base case? How do you call size recursively to get closer to the base case?

Coding Demo: addLast

SLList.java

```
public class SLList {
   private IntNode first;


   private static int size(IntNode p) {





   }

   public int size() {

   }
}
```

Solution: Write a private static helper method that takes in an extra argument to help with recursion.

(This can be static because we don't need to reference first.)

Coding Demo: addLast

SLList.java

```
public class SLList {
   private IntNode first;

   /** Returns the size of the list that starts at IntNode p. */
   private static int size(IntNode p) {






   }

   public int size() {

   }
}
```

Coding Demo: addLast

SLList.java

```
public class SLList {
   private IntNode first;

   /** Returns the size of the list that starts at IntNode p. */
   private static int size(IntNode p) {
      if (p.next == null) {

      }


   }

   public int size() {

   }
}
```

Coding Demo: addLast

SLList.java

```
public class SLList {
   private IntNode first;

   /** Returns the size of the list that starts at IntNode p. */
   private static int size(IntNode p) {
      if (p.next == null) {
         return 1;
      }


   }

   public int size() {

   }
}
```

Coding Demo: addLast

SLList.java

```java
public class SLList {
   private IntNode first;

   /** Returns the size of the list that starts at IntNode p. */
   private static int size(IntNode p) {
      if (p.next == null) {
         return 1;
      }

      return 1 + size(p.next);
   }

   public int size() {

   }
}
```

Coding Demo: addLast

SLList.java

```java
public class SLList {
   private IntNode first;

   /** Returns the size of the list that starts at IntNode p. */
   private static int size(IntNode p) {
      if (p.next == null) {
         return 1;
      }

      return 1 + size(p.next);
   }

   public int size() {
      return size(first);
   }
}
```

Private Recursive Helper Methods

To implement a recursive method in a class that is not itself recursive (e.g. SLList):

- Create a private recursive helper method.
- Have the public method call the private recursive helper method.

```
public class SLList {
   private IntNode first;

   private int size(IntNode p) {
      if (p.next == null) {
         return 1;
      }

      return 1 + size(p.next);
   }

   public int size() {
      return size(first);
   }
}
```

`size` Efficiency, caching

Lecture 4, CS61B, Fall 2024

Getting Started

- `IntList` → `IntNode`
- Creating the SLList Class
- `addFirst` and `getFirst`
- SLLists vs. IntLists

Syntax Improvements:

- Access Control: Public vs. Private
- Nested Classes

`addLast` and `size`

- Creating `addLast` and `size`
- **`size` Efficiency, caching**

The Empty List

- `addLast` Bug
- Sentinel Nodes
- Invariants

Efficiency of Size

How efficient is size?

- Suppose size takes 2 seconds on a list of size 1,000.
- How long will it take on a list of size 1,000,000?

a. 0.002 seconds.
b. 2 seconds.
c. 2,000 seconds.
d. 2,000,000 seconds.

```java
public class SLList {
   private IntNode first;

   private int size(IntNode p) {
      if (p.next == null) {
         return 1;
      }

      return 1 + size(p.next);
   }

   public int size() {
      return size(first);
   }
}
```

Improvement #5: Fast size()

Your goal:

- Modify SLList so that the execution time of size() is always fast (i.e. independent of the size of the list).

(video viewers only, time is too tight in class to think carefully about this)

```
public class SLList {
   private IntNode first;

   public SLList(int x) {
      first = new IntNode(x, null);
   }

   public void addFirst(int x) {
      first = new IntNode(x, first);
   }

   private int size(IntNode p) {
      if (p.next == null)
         return 1;
      return 1 + size(p.next);
   }

   public int size() {
      return size(first);
   }
}
```

Coding Demo: Fast size

SLList.java

```java
public class SLList {
    private IntNode first;
    private int size;

}
```

Instead of re-calculating size on demand every time size is called, we'll keep track of the current size in a private variable.

Then, we'll update the size variable every time the list is changed.

This variable is redundant (we could calculate the size from the list), but will save us time.

Coding Demo: Fast size

SLList.java

```java
public class SLList {
   private IntNode first;
   private int size;

   public SLList(int x) {
      first = new IntNode(x, null);

   }



}
```

Coding Demo: Fast size

SLList.java

```
public class SLList {
   private IntNode first;
   private int size;

   public SLList(int x) {
      first = new IntNode(x, null);
      size = 1;
   }



}
```

New line added to existing function.

Coding Demo: Fast size

SLList.java

```
public class SLList {
   private IntNode first;
   private int size;

   /** Adds x to the front of the list. */
   public void addFirst(int x) {
      first = new IntNode(x, first);

   }








}
```

Coding Demo: Fast size

SLList.java

```
public class SLList {
   private IntNode first;
   private int size;

   /** Adds x to the front of the list. */
   public void addFirst(int x) {
      first = new IntNode(x, first);
      size += 1;
   }



}
```

New line added to existing function.

Coding Demo: Fast size

SLList.java

```java
public class SLList {
   private IntNode first;
   private int size;

   /** Returns the first item in the list. */
   public int getFirst() {
      return first.item;
   }









}
```

No modification needed. getFirst doesn't change the size of the list.

Coding Demo: Fast size

SLList.java

```
public class SLList {
   private IntNode first;
   private int size;

   /** Adds x to the end of the list. */
   public void addLast(int x) {

      IntNode p = first;

      /** Move p until it reaches the end of the list. */
      while (p.next != null) {
         p = p.next;
      }

      p.next = new IntNode(x, null);
   }
}
```

Coding Demo: Fast size

SLList.java

```
public class SLList {
   private IntNode first;
   private int size;

   /** Adds x to the end of the list. */
   public void addLast(int x) {
      size += 1;
      IntNode p = first;

      /** Move p until it reaches the end of the list. */
      while (p.next != null) {
         p = p.next;
      }

      p.next = new IntNode(x, null);
   }
}
```

New line added to existing function.

Coding Demo: Fast size

SLList.java

```java
public class SLList {
    private IntNode first;
    private int size;

    public int size() {
        return size;
    }
























}
```

Improvement #5: Fast size()

Solution: Maintain a special size variable that **caches** the size of the list.

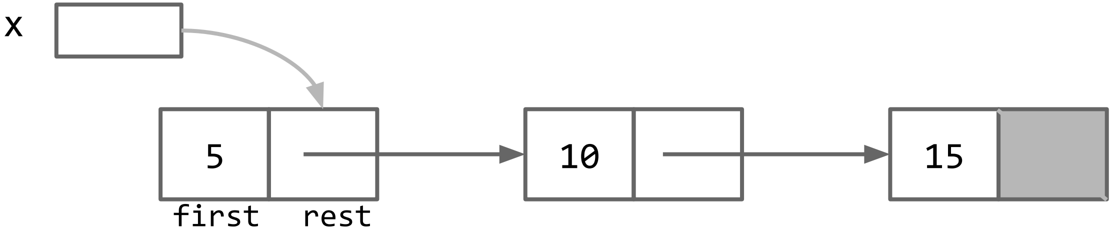
- Caching: putting aside data to speed up retrieval.

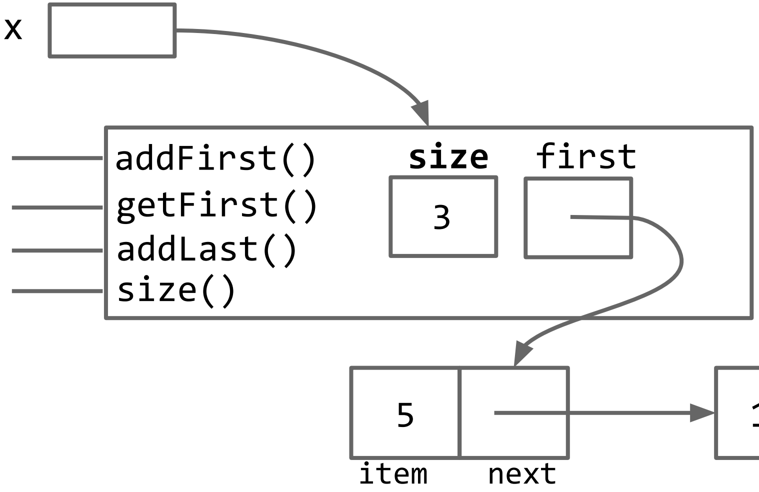
TANSTAAFL: There ain't no such thing as a free lunch.

- But spreading the work over each add call is a net win in almost any circumstance.

http://www.ensler.us/ensler.us/images/nolnchsmalla.jpg

Naked Linked Lists (IntList) vs. SLLists





SLList class acts as a middle man between user and the naked recursive data structure. Allows us to store meta information about entire list, e.g. **size**.

The Empty List and the `addLast` Bug

Lecture 4, CS61B, Fall 2024

Coding Demo: The Empty List

SLList.java

```java
public class SLList {
    private IntNode first;
    private int size;

    public SLList(int x) {
        first = new IntNode(x, null);
        size = 1;
    }

    /** Creates an empty SLList. */
    public SLList() {


    }
}
```

Adding a second constructor for empty lists (in addition to the existing constructor for making lists with 1 element).

Coding Demo: The Empty List

SLList.java

```
public class SLList {
   private IntNode first;
   private int size;

   public SLList(int x) {
      first = new IntNode(x, null);
      size = 1;
   }

   /** Creates an empty SLList. */
   public SLList() {

      size = 0;
   }
}
```

Adding a second constructor for empty lists (in addition to the existing constructor for making lists with 1 element).

Coding Demo: The Empty List

SLList.java

```
public class SLList {
    private IntNode first;
    private int size;

    public SLList(int x) {
        first = new IntNode(x, null);
        size = 1;
    }

    /** Creates an empty SLList. */
    public SLList() {
        first = null;
        size = 0;
    }
}
```

Adding a second constructor for empty lists (in addition to the existing constructor for making lists with 1 element).

Coding Demo: The Empty List

SLList.java

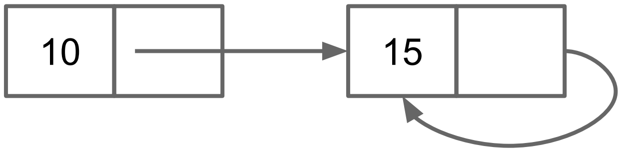
```
public class SLList {
   private IntNode first;
   private int size;

   public static void main(String[] args) {
      SLList L = new SLList();
      L.addFirst(10);
      L.addFirst(5);
      L.addLast(20);
      System.out.println(L.getFirst()); // should print 5
   }



}
```

[Java visualizer]

Improvement #6a: Representing the Empty List

Benefits of `SLList` vs. `IntList` so far:

- Faster `size()` method than would have been convenient for `IntList`.
- User of an `SLList` never sees the `IntList` class.
  - Simpler to use.
  - More efficient `addFirst` method (see exercises).
  - Avoids errors (or malfeasance):



Another benefit we can gain:

- Easy to represent the empty list. Represent the empty list by setting `first` to null. Let's try!
- We'll see there is a **very** subtle bug in the code. It crashes when you call addLast on the empty list.

Coding Demo: The addLast Bug

SLList.java

```
public class SLList {
   private IntNode first;
   private int size;

   public static void main(String[] args) {
      SLList L = new SLList();
      L.addLast(20); // program crashes!
   }







}
```

Java visualizer

Coding Demo: The addLast Bug

SLList.java

```
public class SLList {
   private IntNode first;
   private int size;

   /** Adds x to the end of the list. */
   public void addLast(int x) {
      size += 1;
      IntNode p = first;

      /** Move p until it reaches the end of the list. */
      while (p.next != null) {
         p = p.next;
      }

      p.next = new IntNode(x, null);
   }
}
```

The NullPointerException traceback identifies this line as where the program crashed.

How Would You Fix addLast?

Your goal:

- Fix addLast so that we do not get a null pointer exception when we try to add to the back of an empty `SLList`:

```
SLList s1 = new SLList();
s1.addLast(5);
```

```
public class SLList {
    private IntNode first;
    private int size;

    public SLList() {
        first = null;
        size = 0;
    }

    public void addLast(int x) {
        size += 1;
        IntNode p = first;
        while (p.next != null) {
            p = p.next;
        }

        p.next = new IntNode(x, null);
    }
}
```

See study guide for starter code if you want to try on a computer.

One Solution

One possible solution:

- Add a special case for the empty list.

But there are other ways...

```java
public void addLast(int x) {
    size += 1;

    if (first == null) {
        first = new IntNode(x, null);
        return;
    }

    IntNode p = first;
    while (p.next != null) {
        p = p.next;
    }

    p.next = new IntNode(x, null);
}
```

Sentinel Nodes

Lecture 4, CS61B, Fall 2024

Tip For Being a Good Programmer: Keep Code Simple

As a human programmer, you only have so much working memory.

- You want to restrict the amount of complexity in your life!
- Simple code is (usually) good code.
  - Special cases are not ‘simple’.

```
public void addLast(int x) {
    size += 1;

    if (first == null) {
        first = new IntNode(x, null);
        return;
    }

    IntNode p = first;
    while (p.next != null) {
        p = p.next;
    }

    p.next = new IntNode(x, null);
}
```

addLast's Fundamental Problem

The fundamental problem:

- The empty list has a null **`first`**. Can't access `first.next`!

Our fix is a bit ugly:

- Requires a special case.
- More complex data structures will have many more special cases (gross!!)

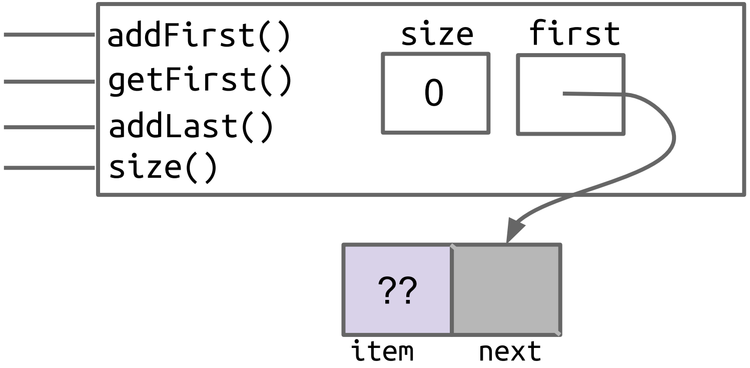
How can we avoid special cases?

- Make all `SLLists` (even empty) the "same".

```
public void addLast(int x) {
   size += 1;

   if (first == null) {
      first = new IntNode(x, null);
      return;
   }

   IntNode p = first;
   while (p.next != null) {
      p = p.next;
   }

   p.next = new IntNode(x, null);
}
```

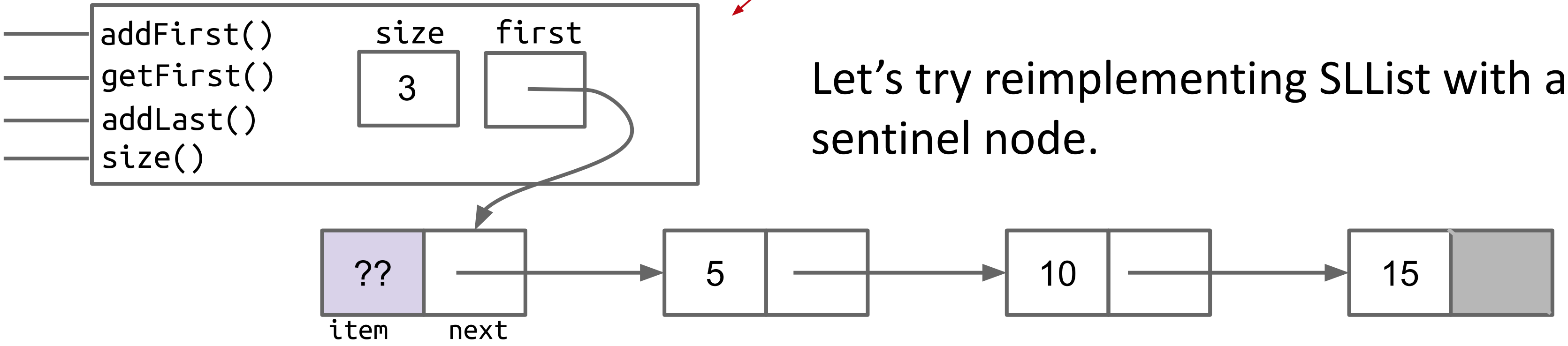
Improvement #6b: Representing the Empty List Using a Sentinel

Create a special node that is always there! Let's call it a "sentinel node".



The empty list is just the sentinel node.

A list with 3 numbers has a sentinel node and 3 nodes that contain real data.



Let's try reimplementing SLList with a sentinel node.

Coding Demo: Sentinel Nodes

SLList.java

```java
public class SLList {

    private IntNode first;
    private int size;

    public SLList(int x) {

        first = new IntNode(x, null);
        size = 1;
    }

    public SLList() {
        first = null;
        size = 0;
    }

}
```

In this demo, we'll be modifying our existing code to account for the sentinel node.

Coding Demo: Sentinel Nodes

SLList.java

```
public class SLList {
   /** The first item (if it exists) is at sentinel.next. */
   private IntNode sentinel;
   private int size;

   public SLList(int x) {

      first = new IntNode(x, null);
      size = 1;
   }

   public SLList() {
      first = null;
      size = 0;
   }

}
```

Renaming first to sentinel.

Coding Demo: Sentinel Nodes

SLList.java

```java
public class SLList {
   /** The first item (if it exists) is at sentinel.next. */
   private IntNode sentinel;
   private int size;

   public SLList(int x) {

      first = new IntNode(x, null);
      size = 1;
   }

   public SLList() {
      sentinel = new IntNode(63, null);
      size = 0;
   }

}
```

The empty list is no longer null. It's the sentinel node (and no other nodes).

63 is a placeholder. We don't care about the sentinel node's value.

Coding Demo: Sentinel Nodes

SLList.java

```java
public class SLList {
   /** The first item (if it exists) is at sentinel.next. */
   private IntNode sentinel;
   private int size;

   public SLList(int x) {
      sentinel = new IntNode(63, null);
      sentinel.next = new IntNode(x, null);
      size = 1;
   }

   public SLList() {
      sentinel = new IntNode(63, null);
      size = 0;
   }

}
```

The single-element list is no longer just one node. It's two nodes: the sentinel node, followed by the node with the single value.

Coding Demo: Sentinel Nodes

SLList.java

```
public class SLList {
   /** The first item (if it exists) is at sentinel.next. */
   private IntNode sentinel;
   private int size;

   /** Adds x to the front of the list. */
   public void addFirst(int x) {
      first = new IntNode(x, first);
      size += 1;
   }

}
```

Need to modify addFirst to be consistent with the sentinel structure.

Coding Demo: Sentinel Nodes

SLList.java

```java
public class SLList {
   /** The first item (if it exists) is at sentinel.next. */
   private IntNode sentinel;
   private int size;

   /** Adds x to the front of the list. */
   public void addFirst(int x) {
      sentinel.next = new IntNode(x, sentinel.next);
      size += 1;
   }


}
```

We need to reassign sentinel.next here, because "the first item is at sentinel.next."

Coding Demo: Sentinel Nodes

SLList.java

```java
public class SLList {
   /** The first item (if it exists) is at sentinel.next. */
   private IntNode sentinel;
   private int size;

   /** Returns the first item in the list. */
   public int getFirst() {
      return first.item;
   }

}
```

Need to modify getFirst to be consistent with the sentinel structure.

Coding Demo: Sentinel Nodes

SLList.java

```
public class SLList {
   /** The first item (if it exists) is at sentinel.next. */
   private IntNode sentinel;
   private int size;

   /** Returns the first item in the list. */
   public int getFirst() {
      return sentinel.next.item;
   }



}
```

Recall: "the first item is at sentinel.next."

If we returned sentinel.item, we would get the placeholder 63, not the true first item of the list.

Coding Demo: Sentinel Nodes

SLList.java

```
public class SLList {
    /** The first item (if it exists) is at sentinel.next. */
    private IntNode sentinel;
    private int size;

    /** Adds x to the end of the list. */
    public void addLast(int x) {
        size += 1;
        IntNode p = first;

        /** Move p until it reaches the end of the list. */
        while (p.next != null) {
            p = p.next;
        }
        p.next = new IntNode(x, null);
    }
}
```

Need to modify addLast to be consistent with the sentinel structure.

Coding Demo: Sentinel Nodes

SLList.java

```
public class SLList {
   /** The first item (if it exists) is at sentinel.next. */
   private IntNode sentinel;
   private int size;

   /** Adds x to the end of the list. */
   public void addLast(int x) {
      size += 1;
      IntNode p = sentinel;

      /** Move p until it reaches the end of the list. */
      while (p.next != null) {
         p = p.next;
      }
      p.next = new IntNode(x, null);
   }
}
```

Start p at the sentinel, and move p until it reaches the end of the list.

Sentinel always exists, so the addLast bug doesn't apply anymore.

Coding Demo: Sentinel Nodes

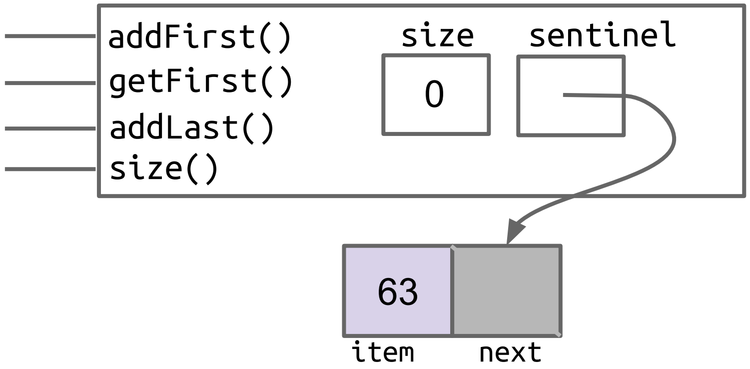
SLList.java

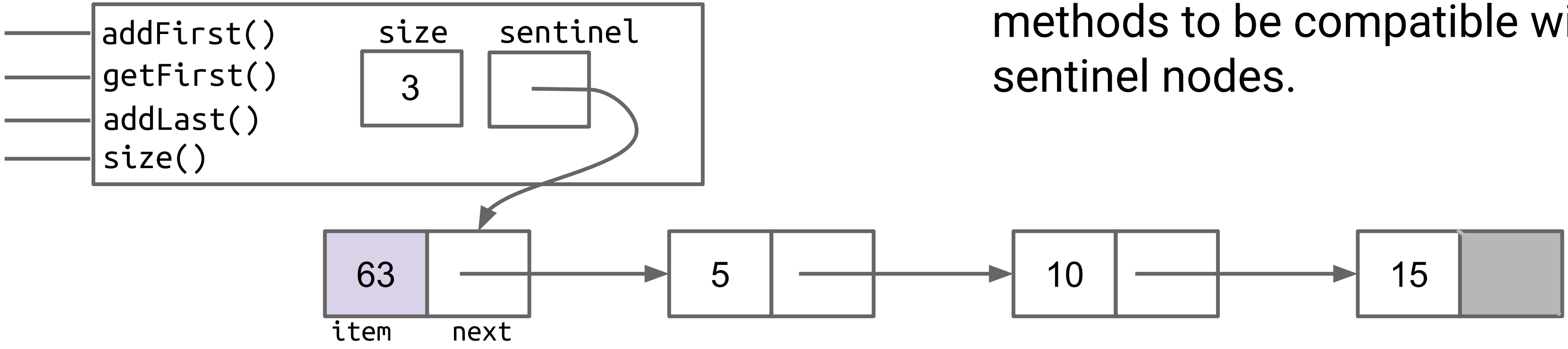
```
public class SLList {
   /** The first item (if it exists) is at sentinel.next. */
   private IntNode sentinel;
   private int size;

   public static void main(String[] args) {
      SLList L = new SLList();
      L.addLast(20);
      System.out.println(L.size()); //should print out 1
   }
}
```

[Java visualizer]

Sentinel Node

The sentinel node is always there for you.





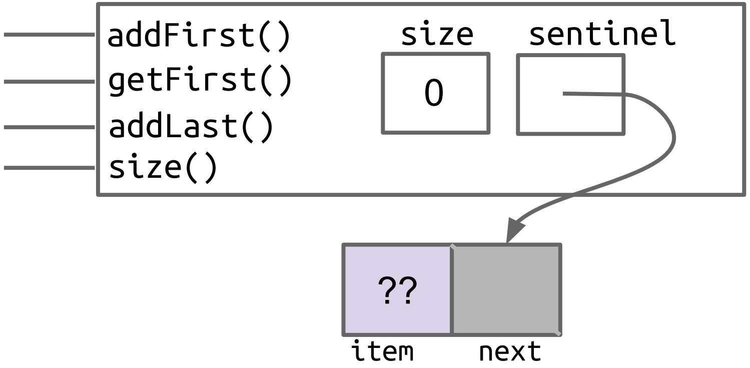
Notes:

- I’ve renamed `first` to be `sentinel`.
- `sentinel` is never null, always points to sentinel node.
- Sentinel node’s `item` needs to be some integer, but doesn’t matter what value we pick.
- Had to fix constructors and methods to be compatible with sentinel nodes.

addLast (with Sentinel Node)

Bottom line: Having a sentinel simplifies our `addLast` method.

- No need for a special case to check if `sentinel` is null (since it is never null).




```java
public void addLast(int x) {
    size += 1;

    if (sentinel == null) {
        sentinel = new IntNode(x, null);
        return;
    }

    IntNode p = sentinel;
    while (p.next != null) {
        p = p.next;
    }

    p.next = new IntNode(x, null);
}
```

Invariants

Lecture 4, CS61B, Fall 2024

Invariants

An invariant is a condition that is guaranteed to be true during code execution (assuming there are no bugs in your code).

An `SLList` with a sentinel node has at least the following invariants:

- The `sentinel` reference always points to the sentinel node.
- The first node (if it exists), is always at `sentinel.next`.
- The `size` variable is always the total number of items that have been added.

Invariants make it easier to reason about code:

- Can assume they are true to simplify code (e.g. `addLast` doesn't need to worry about nulls).
- Must ensure that methods preserve invariants.

Summary

Methods	Non-Obvious Improvements	
`addFirst(int x)`	#1	Rebranding: `IntList` → `IntNode`
`getFirst`	#2	Bureaucracy: `SLList`
`size`	#3	Access Control: `public` → `private`
`addLast(int x)`	#4	Nested Class: Bringing `IntNode` into `SLList`
	#5	Caching: Saving `size` as an `int`.
	#6	Generalizing: Adding a `sentinel` node to allow representation of the empty list.

Today: Just an Introduction

If today felt like a lot to digest, don’t worry.

The `LinkedListDeque` class that you’ll build in project 1A will give you practice so that you can deeply understand the ideas from this week’s lectures.