CS 600.226: Data Structures

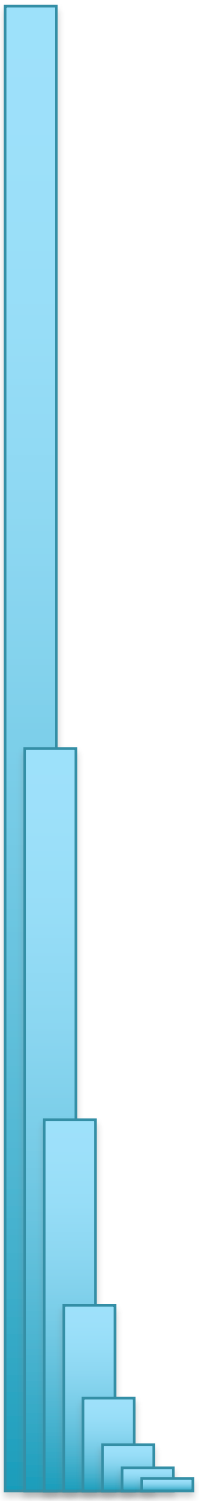
Michael Schatz

Sept 28 2018

Lecture 13. More Lists

Agenda

1. *Questions on HW3*
2. *Introduce HW4*
3. *Lists and More Lists*



Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Assignment 3: Assorted Complexities

Out on: September 21, 2018
Due by: September 28, 2018 before 10:00 pm
Collaboration: None
Grading:

Functionality 60% (where applicable)
Solution Design and README 10% (where applicable)
Style 10% (where applicable)
Testing 10% (where applicable)

Overview

The third assignment is mostly about sorting and how fast things go. You will also write yet another implementation of the Array interface to help you analyze how many array operations various sorting algorithms perform.

Note: The grading criteria now include 10% for unit testing. This refers to JUnit 4 test drivers, not some custom test program you hacked. The problems (on this and future assignments) will state whether you are expected to produce/improve test drivers or not.

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Problem 1: Arrays with Statistics (30%)

Your first task for this assignment is to develop a new kind of `Array` implementation that keeps track of how many read and write operations have been performed on it. Check out the `Statable` interface first, reproduced here in compressed form (be sure to use and read the full interface available in github):

```
public interface Statable {
    void resetStatistics();
    int numberOfReads();
    int numberOfWrites();
}
```

This describes what we expect of an object that can collect statistics about itself. After a `Statable` object has been "in use" for a while, we can check how many read and write operations it has been asked to perform. We can also tell it to "forget" what has happened before and start counting both kinds of operations from zero again.

Make sure to update: length(), get(), and put(); skip the iterator

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Problem 2: All Sorts of Sorts (50%)

You need to write classes implementing BubbleSort and InsertionSort for this problem. Just like our example algorithms, your classes have to implement the SortingAlgorithm interface.

All of this should be fairly straightforward once you get used to the framework. Speaking of the framework, the way you actually "run" the various algorithms is by using the PolySort.java program we've provided as well. You should be able to compile and run it without yet having written any sorting code yourself.

Here's how:

```
$ java PolySort 4000 <random.data

 Algorithm            Sorted?  Size         Reads          Writes         Seconds

 Null Sort            false    4,000        0              0              0.000007
 Gnome Sort           true     4,000        32,195,307     8,045,828      0.243852
 Selection Sort       true     4,000        24,009,991     7,992          0.252085
```

The running times can be a clue, but write up should reflect on more than time

Assignment 3: Due Friday Sept 28 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment03/README.md

Problem 3: Analysis of Selection Sort (20%)

Your final task for this assignment is to analyze the following selection sort algorithm theoretically (without running it) in detail (without using O-notation).

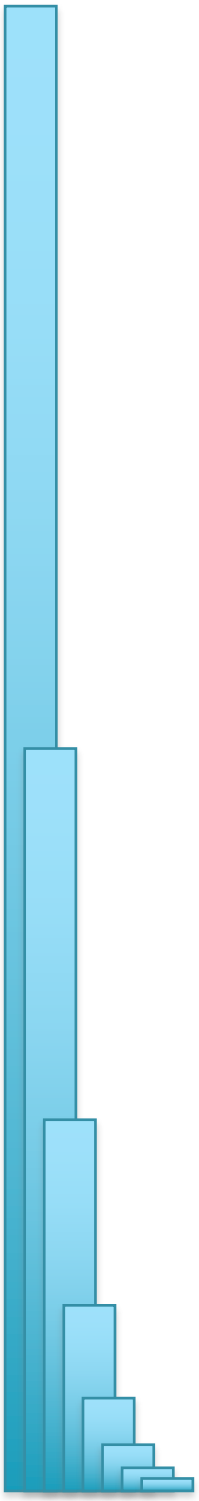
Here's the code, and you must analyze exactly this code (the line numbers are given so you can refer to them in your writeup for this problem):

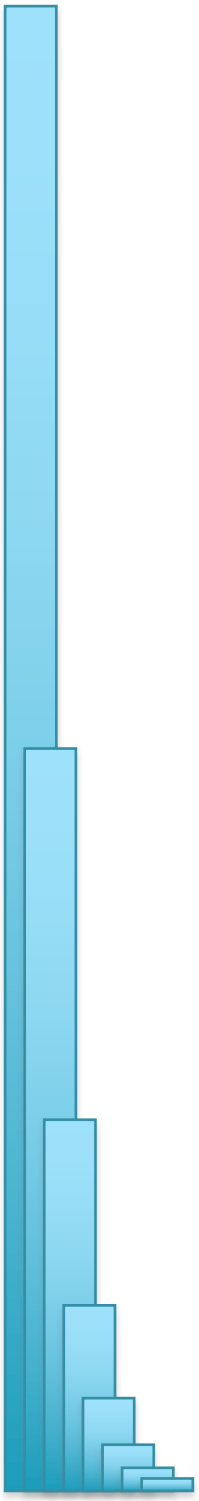
```
 1: public static void selectionSort(int[] a) {
 2:     for (int i = 0; i < a.length - 1; i++) {
 3:         int min = i;
 4:         for (int j = i + 1; j < a.length; j++) {
 5:             if (a[j] < a[min]) {
 6:                 min = j;
 7:             }
 8:         }
 9:         int t = a[i]; a[i] = a[min]; a[min] = t;
10:     }
11: }
```

Work slowly, line-by-line analysis

Agenda

1. *Updates on HW3*
2. *Introduce HW4*
3. *Lists and More Lists*





Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Assignment 4: Stacking Queues

Out on: September 28, 2018
Due by: October 5, 2018 before 10:00 pm
Collaboration: None
Grading:

Packaging 10%,
Style 10% (where applicable),
Testing 10% (where applicable),
Performance 10% (where applicable),
Functionality 60% (where applicable)

Overview

The fourth assignment is mostly about stacks and dequeues. For the former you'll build a simple calculator application, for the latter you'll implement the data structure in a way that satisfies certain performance characteristics (in addition to the usual correctness properties).

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Problem 1: Calculating Stacks (50%)

Your first task is to implement a basic RPN calculator that supports integer operands like 1, 64738, and -42 as well as the (binary) integer operators +, -, *, /, and %. Your program should be called `Calc` and work as follows:

- You create an empty `Stack` to hold intermediate results and then repeatedly accept input from the user. It doesn't matter whether you use the `ArrayStack` or the `ListStack` we provide, what does matter is that those specific types appear only once in your program.
- If the user enters a ***valid integer***, you ***push*** that integer onto the stack.
- If the user enters a ***valid operator***, you ***pop*** two integers off the stack, ***perform*** the requested operation, and ***push*** the result back onto the stack.
- If the user enters the symbol `?` (that's a question mark), you ***print*** the current state of the stack using its `toString` method followed by a new line.
- If the user enters the symbol `.` (that's a dot or full-stop), you ***pop*** the top element off the stack and ***print*** it (only the top element, not the entire stack) followed by a new line.
- If the user enters the symbol `!` (that's an exclamation mark or bang), you exit the program.

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

```
$ java Calc
?
[]
10
?
[10]
20 30
?
[30, 20, 10]
*
?
[600, 10]
+
?
[610]
.
610
!
$
```

```
$ java Calc
? 10 ? 20 30 ? *
? + ? . !
[]
[10]
[30, 20, 10]
[600, 10]
[610]
610
$
```

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Problem 1: Calculating Stacks (50%)

Note that there are a number of error conditions that your program must deal with gracefully for full credit. We'll give you two examples for free, you'll have to figure out any further error conditions for yourself:

- If the ***user enters*** `blah` (or anything else that doesn't make sense for a calculator as specified above) your program ***should make it clear*** that it can't do anything helpful with that input; ***but it should not stop*** at that point.

- If the user requests an operation for which there are ***not enough operands*** on the stack, your program ***should notify the user*** of the problem but ***leave the stack unchanged***; again, it should certainly not stop at that point.

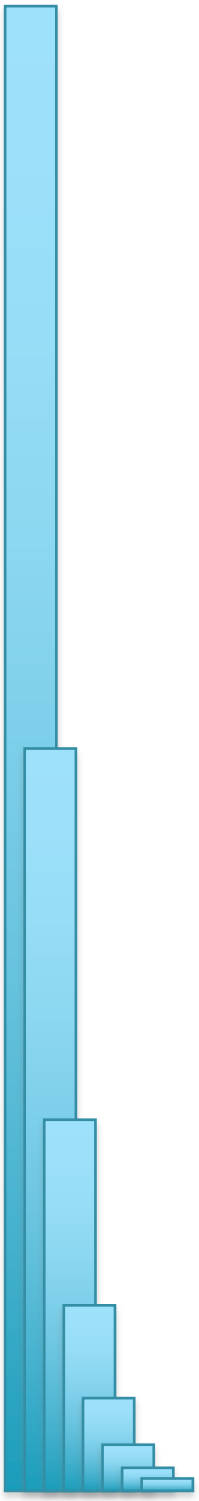
- Of course this means that you'll have to print error messages to the user. ***Error messages must be printed to standard error and not to standard out!*** (Of course, the regular input and output is done through standard in and standard out as usual.)

- Furthermore, ***all error messages must start with the symbol ?*** (that's a question mark) and be followed by a new line!

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

```
$ java Calc
1 2 blah 1.0 3 ?
?Huh?
?Huh?
[3, 2, 1]
+ + ?
[6]
+ + ?
?Not enough arguments.
?Not enough arguments.
[6]
!
$
```



Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Problem 1: Calculating Stacks (50%)

Hints

- Note that we're dealing with ***integers only*** (type Integer in Java) so / stands for integer division and % stands for integer remainder. Both of these should behave in `Calc` just like they do in Java. (The details are messy but worth knowing about, especially regarding modulus.)

- You may find it interesting to read up on ***Autoboxing*** and ***Unboxing*** in Java. It's the reason we can use our generic Stack implementations for Integer objects yet still do arithmetic like we would on regular int variables.

- You'll probably want to use a ***Scanner*** object, the methods hasNext and next, but nothing else for getting the input.

- Only if you're not afraid of learning on your own: You'll be able to use the `matches` method of the String class to your advantage when it comes to checking whether a valid operator was entered. (But you can just as well do it with a bunch of separate comparisons if you don't want to learn about regular expressions.)

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Problem 2: Hacking Growable Dequeues (50%)

Your second task is to implement a generic `ArrayDequeue` class as outlined in lecture. As is to be expected, `ArrayDequeue` must implement the `Dequeue` interface we provided on github.

- Your implementation must be done in terms of an array that grows by doubling as needed. It's up to you whether you want to use a basic Java array or the SimpleArray class you know and love; just in case you prefer the latter, we've once again included it on the github directory for this assignment. Your initial array must have a length of one slot only! (Trust us, that's going to make debugging the "doubling" part a lot easier.)
- Your implementation must support all `Dequeue` operations except insertion in (worst-case) constant time; insertion can take longer every now and then (when you need to grow the array), but overall all insertion operations must be constant amortized time as discussed in lecture.
- You should provide a `toString` method in addition to the methods required by the `Dequeue` interface. A new dequeue into which 1, 2, and 3 were inserted using `insertBack()` should print as [1, 2, 3] while an empty dequeue should print as []

Assignment 4: Due Friday Oct 5 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

Problem 2: Hacking Growable Dequeues (50%)

Testing

- You must write a JUnit 4 test driver for your `ArrayDequeue` class in a file `TestArrayDequeue.java`. Be sure to test all methods and all exceptions as well. Note that it is not enough to have just one test case for each method; there are plenty of complex interactions between the methods that should be covered as well. (And yes, you need to test `toString()` as well.)

Documentation

- Don't forget to add proper javadoc comments for your ArrayDequeue class.
- Running checkstyle will remind you to do this!

Assignment 4: Due Friday Oct 5 @ 10pm

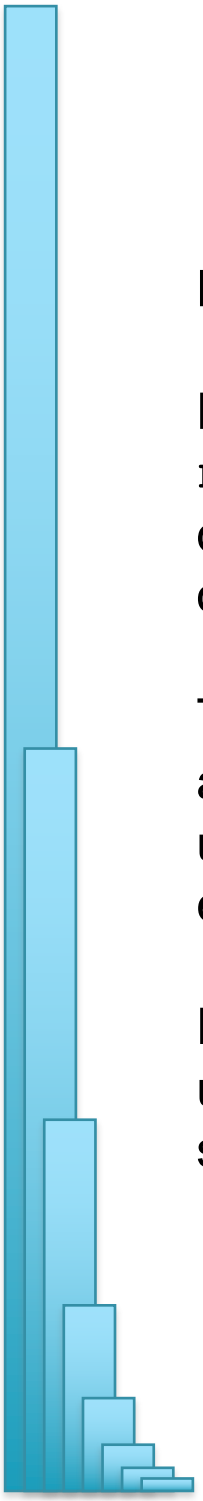
https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment04/README.md

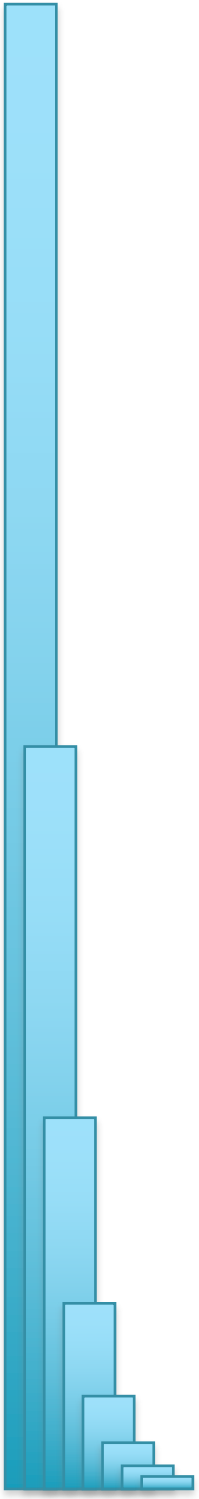
Bonus Problem (5 pts)

Develop an **algebraic specification** for the **abstract data type Queue**. Use `new`, `empty`, `enqueue`, `dequeue`, and `front` (with the meaning of each as discussed in lecture) as your set of operations. Consider unbounded queues only.

The difficulty is going to be modelling the FIFO (first-in-first-out) behavior accurately. You'll probably need at least one axiom with a case distinction using an if expression; the syntax for this in the Array specification for example.

Doing this problem without resorting to Google may be rather helpful for the upcoming midterm. There's no need to submit the problem, but you can submit it if you wish; just include it at the end of your README file.

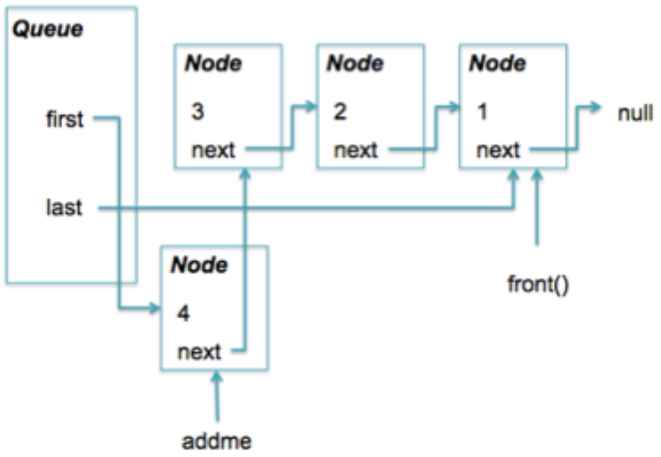




Lists and More Lists

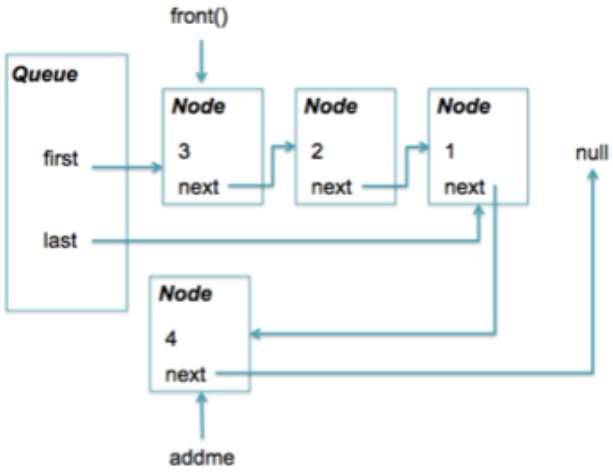
Singly Linked Lists

insertFront



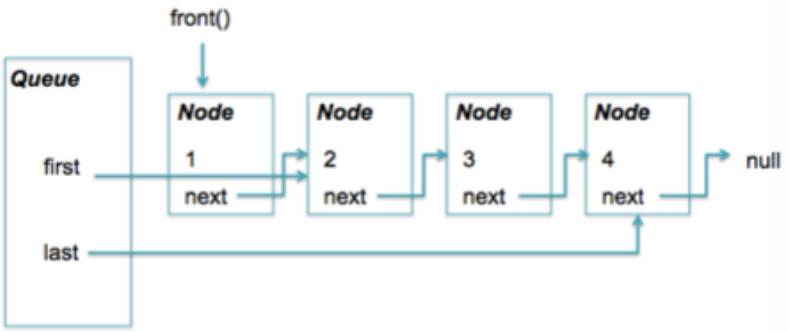
addme.next = first; first = addme;

insertBack



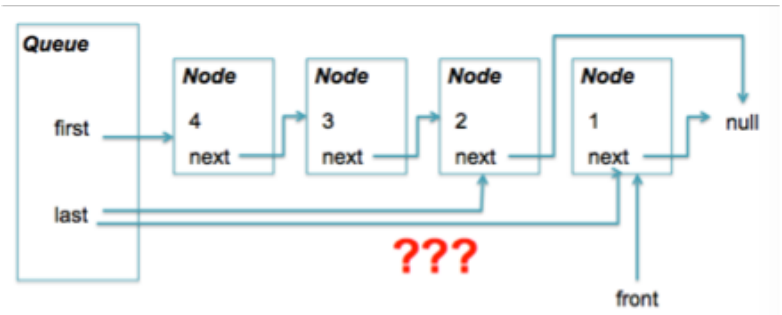
last.next = addme; addme.next = null

removeFront



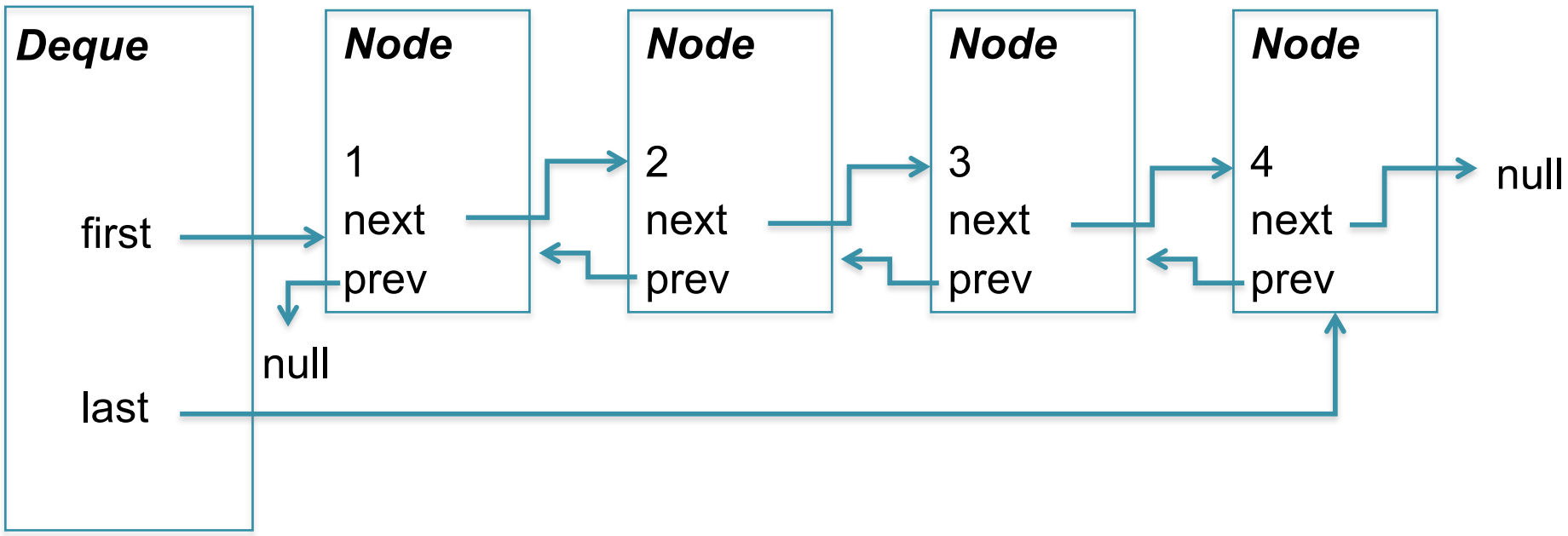
first = first.next;

removeBack



???

Doubly Linked List



Very similar to a singly linked list, except each node has a reference to both the next and previous node in the list

A little more overhead, but significantly increased flexibility: supports insertFront(), insertBack(), removeFront(), removeBack(), insertBefore(), removeMiddle()

List v4

```
public interface Node<T> {
    void setValue(T t);
    T getValue();

    void setNext(Node<T> n);
    void setPrev(Node<T> n);

    void getNext(Node<T> n);
    void getPrev(Node<T> n);
}

public interface List<T> {
    boolean empty();
    int length();

    Node<T> front();
    Node<T> back();

    void insertFront(Node<T> t);
    void insertBack(Node<T> t);

    void removeFront();
    void removeBack();
}
```

List v4

```
public interface Node<T> {
    void setValue(T t);
    T getValue();

    void setNext(Node<T> n);
    void setPrev(Node<T> n);

    void getNext(Node<T> n);
    void getPrev(Node<T> n);
}

public interface List<T> {
    boolean empty();
    int length();

    Node<T> front();
    Node<T> back();

    void insertFront(Node<T> t);
    void insertBack(Node<T> t);

    void removeFront();
    void removeBack();
}
```

Nodes are useful, but is there someway to hide the internals?

List v4

```
public interface Node<T> {
    void setValue(T t);
    T getValue();

    void setNext(Node<T> n);
    void setPrev(Node<T> n);

    void getNext(Node<T> n);
    void getPrev(Node<T> n);
}

public interface List<T> {
    boolean empty();
    int length();

    Node<T> front();
    Node<T> back();

    void insertFront(Node<T> t);
    void insertBack(Node<T> t);

    void removeFront();
    void removeBack();
}
```

```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
    // simplified interface
    int length();

    Position<T> insertFront(T t);
    Position<T> insertBack(T t);
    void insertBefore(Position<T> t);
    void insertAfter(Position<T> t);

    void removeAt(Position<T> p);
}
```

List v4

"I am a position and while you can hold on to me, you can't do anything else with me!"

```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
    // simplified interface
    int length();

    Position<T> insertFront(T t);
    Position<T> insertBack(T t);
    void insertBefore(Position<T> t);
    void insertAfter(Position<T> t);

    void removeAt(Position<T> p);
}
```

Inserting at front or back creates the Position objects.

If you want, you could keep references to the Position objects even in the middle of the list

Pass in a Position, and it will remove it from the list

```
    void getNext(Node<T> n);
    void getPrev(Node<T> n);

    t);
    );

    void removeFront();
    void removeBack();
}
```

List v4

```
public class NodeList<T> implements List<T>
    private static class Node<T> implements Position<T> {
        Node<T> next;
        Node<T> prev;
        T data;
        List<T> owner; // reference back to the List it came from
    }

    private Node<T> front;
    private Node<T> back;
    private int elements;
    public int length() { return this.elements; }

    public Position<T> insertFront(T t) {
       ...
    }

    public Position<T> insertBack(T t) {
       ...
    }

    public void remoteAt(Position<T> p) {
       ...
    }
}
```

Public Position interface,
but nested (private static) Node Implementation

List v4

```
public String toString () {
  String s ="[";
  Node<T> n = this.front;
  while (n != null) {
    s += n.data.toString();
    if (n.next != null) {
      s +="";
    }
    n = n.next;
  }
  s +="]";
  return s;
}
```

```
public static void main(String[] args) {
  List l = new NodeList();
  Position a = l.insertFront("Mike");
  System.out.println(l);
  Position b = l.insertBack("Peter");
  System.out.println(l);
  Position c = l.insertFront("Kelly");
  System.out.println(l);

  l.remove(a);
  System.out.println(l);
}
```

```
$ java NodeList
[Mike]
[Mike, Peter]
[Kelly, Mike, Peter]
[Kelly, Peter]
```

How to test the code?

```
public class NodeList<T> implements List<T> {
    private static final class Node<T> implements Position<T> {
        Node<T> next;
        Node<T> prev;
        T data;
        List<T> owner;
        public T get() { return this.data;  }
        public void put(T t) { this.data = t; }
    }
    ...
    public Position<T> front() throws EmptyException { … }
    public Position<T> back() throws EmptyException { … }
    public Position<T> insertFront(T t) { … }
    public Position<T> insertBack(T t) { … }
    public void removeFront() throws EmptyException { … }
    public void removeBack() throws EmptyException { … }
    public Position<T> insertBefore(Position<T> p, T t)
            throws PositionException { … }
    public Position<T> insertAfter(Position<T> p, T t)
        throws PositionException { … }
    public void remove(Position<T> p) throws PositionException { … }
    public String toString() { … }
}
```

How to test the code?

```
public class NodeList<T> implements List<T> {
    private static final class Node<T> implements Position<T> {
        Node<T> next;
        Node<T> prev;
        T data;
        List<T> owner;
        public T get() { return this.data;  }
        public void put(T t) { this.data = t; }
    }
    ...
    public Position<T> front() throws EmptyException { … }
    public Position<T> back() throws EmptyException { … }
    public Position<T> insertFront(T t) { … }
    public Position<T> insertBack(T t) { … }
    public void removeFront() throws EmptyException { … }
    public void removeBack() throws EmptyException { … }
    public Position<T> insertBefore(Position<T> p, T t)
            throws PositionException { … }
    public Position<T> insertAfter(Position<T> p, T t)
        throws PositionException { … }
    public void remove(Position<T> p) throws PositionException { … }
    public String toString() { … }
}
```

TestList.java (1)

```
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

public class TestList {
    List<String> list ;

    @Before
    public void setupList () {
        list = new NodeList<String>();
    }

    @Test
    public void testEmptyList () {
        assertEquals ("[]", list.toString());
        assertEquals (0, list.length());
    }

    @Test
    public void testInsertFront () {
        list.insertFront("Peter");
        list.insertFront("Paul");
        assertEquals("[Paul, Peter]", list.toString());
        assertEquals(2, list.length());
    }
```

TestList.java (2)

```
@Test
public void testInsertBack () {
    list.insertBack("Peter");
    list.insertBack("Paul");
    assertEquals ("[Peter, Paul]", list.toString ());
    assertEquals (2, list.length ());
}
```

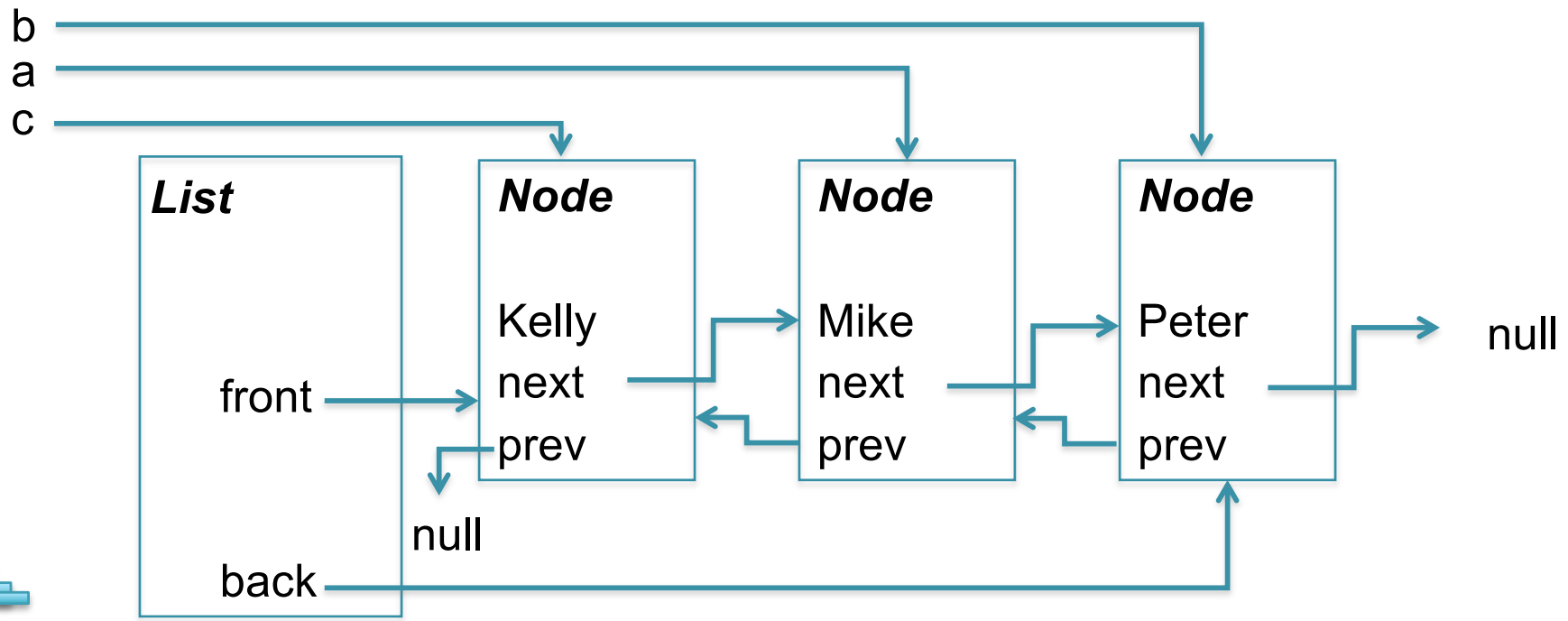
What tests are missing?

As we add functionality,
testing code will become significantly longer
than the implementation
☺

List v4

```
List l = new List<String>();

Position a = l.insertFront("Mike");
Position b = l.insertBack("Peter");
Position c =
l.insertFront("Kelly");

l.removeAt(a);
```

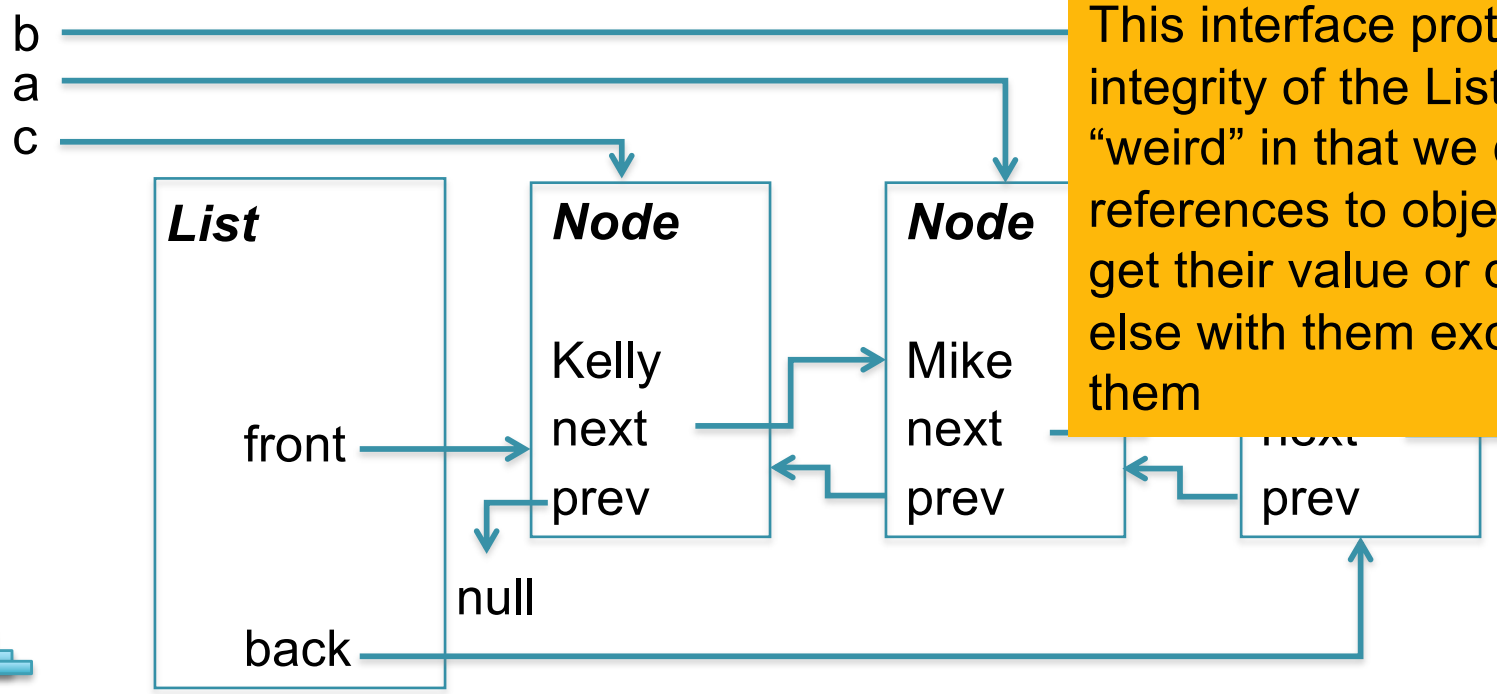
```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



List v4

```
List l = new List<String>();

Position a = l.insertFront("Mike");
Position b = l.insertBack("Peter");
Position c =
l.insertFront("Kelly");

l.removeAt(a);
```

```
public interface Position<T> {
    // empty on purpose
}

public interface List<T> {
     // simplified interface
     int length();

     Position<T> insertFront(T t);
     Position<T> insertBack(T t);

     // TODO: void is temporary
     void removeAt(Position<T> p);
}
```



This interface protects the integrity of the List, but is "weird" in that we can have references to objects but cant get their value or do anything else with them except remove them

List v5

```
public interface Position<T> {
    T get();
    void put(T t);
}

public interface List<T> {
    private static class Node<T> implements Position<T> {
        Node<T> next;
        Node<T> prev;
        T data;
        List<T> owner;
        public T get() { return this.data; }
        public void put(T t) { this.data = t; }
    }
    ...
}
```

List v5

```
public interface Position<T> {
    T get();
    void put(T t);
}

public interface List<T> {
    private static class Node<T> implements Position<T> {
        Node<T> next;
        Node<T> prev;
        T data;
        List<T> owner;
        public T get() { return this.data; }
        public void put(T t) { this.data = t; }
    }
    ...
}
```

Hooray, now we can get/set the value from a Position

Why wouldn't you want to do it this way?

What if you wanted a UniqueList<T> that only stored unique items? This would have to be checked in the UniqueList<T> implementation

List v5 Test

```
@Test
public void testPositionGet() {
    Position<String> p1 = list.insertBack("Peter");
    Position<String> p2 = list.insertBack("Paul");
    assertEquals("[Peter Paul]", list.toString());
    assertEquals("Peter", p1.get());
    assertEquals("Paul", p2.get());
}

@Test
public void testPositionPut() {
    Position<String> p1 = list.insertBack("Peter");
    list.insertBack("Paul");
    assertEquals("[Peter Paul]", list.toString());
    p1.put("Mary");
    assertEquals("Mary", p1.get());
    assertEquals("[Mary Paul]", list.toString());
}
```

What else are we missing?

List v6

```
public interface List<T> {
    ...
    Position<T> front() throws EmptyListException;

    Position<T> back() throws EmptyListException;

    Position<T> next(Position<T> p)
              throws InvalidPositionException;

    Position<T> previous(Position<T> p)
              throws InvalidPositionException;

    boolean hasNext(Position<T> p)
              throws InvalidPositionException;

    boolean hasPrevious(Position<T> p)
              throws InvalidPositionException;

   boolean valid(Position<T> p);
    …
}
```

List v6

```
public interface List<T> {
    ...
    Position<T> front() throws EmptyListException;

    Position<T> back() throws EmptyListException;

    Position<T> next(Position<T> p)
            throws InvalidPositionException;

    Position<T> previous(Position<T> p)
            throws InvalidPositionException;

    boolean hasNext(Position<T> p)
            throws InvalidPositionException;

    boolean hasPrevious(Position<T> p)
            throws InvalidPositionException;
```

Why do we put next() and prev() into the list and not Position?

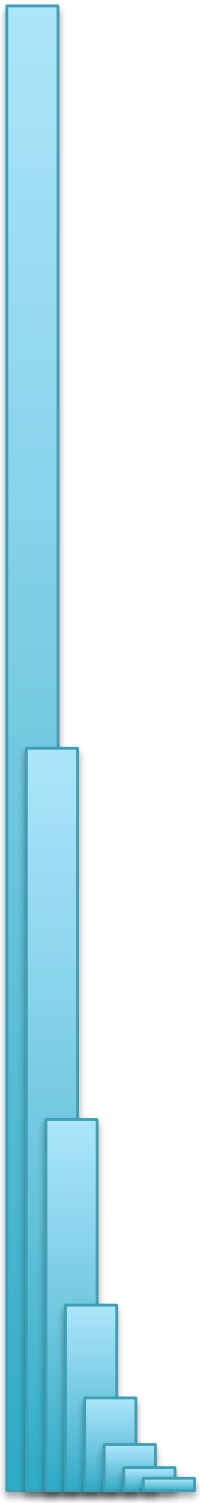
For more complex data structures, like trees or graphs, next() and prev() will be more complicated

List v6 Iterating

```
// Very C++ like
Position<String> current = list.front();
Position<String> last = list.back();
while (current != last) {
    // do whatever we need to do at the current position
    current = list.next(current);
}
```

```
// More Java-like
Position<String> current = list.front();
for (;;) {
    // do whatever we need to do at the current position
    if (list.hasNext(current)) {
        current = list.next(current);
    } else {
        break;
    }
}
```

```
// Very Java-like
Position<String> current = list.front();
while (list.valid(current)) {
    // do whatever we need to do at the current position
    current = list.next(current);
}
```



Iterator Interface

```
public interface Iterator<T> {
    boolean valid();
    void next(); // next element, not necessarily "next"
    T get();     // get ok, but put may break invariants
}
```

```
public interface List<T> {
    ...
    Iterator<T> forwardIterator();
    Iterator<T> backwardIterator();
    ...
}


Iterator<String> i = list.forwardIterator();
while (i.valid()) {
    String e = i.get();
    // do whatever with the element e
    i.next();
}
```

Iterator Interface

```
private static class NodeListIterator<T> implements
Iterator<T> {
    private Node<T> current;
    private boolean forward;

    NodeListIterator(Node<T> start, boolean forward) {
        this.current = start;
        this.forward = forward;
    }

    public boolean valid() {
        return this.current != null;
    }

    public void next() {
        this.current = this.forward ? this.current.next
                                    : this.current.prev;
    }

    public T get() {
        return this.current.get();
    }
}
```

Iterator Interface

```
private static class NodeListIterator<T> implements
Iterator<T> {
    private Node<T> current;
    private boolean forward;

    NodeListIterator(Node<T> start, boolean forward) {
        this.current = sta
        this.forward = for
    }

    public boolean valid()
        return this.curren
    }

    public void next() {
        this.current = this.forward ? this.current.next
                                    : this.current.prev;
    }

    public T get() {
        return this.current.get();
    }
}
```

Ternary operator:

```
If (this.forward) {
      this.current = this.current.next;
} else {
      this.current = this.current.prev;
}
```

Iterator Interface

```
public Iterator<T> forwardIterator() {
    return new NodeListIterator<T>(this.front, true);
}

public Iterator<T> backwardIterator() {
    return new NodeListIterator<T>(this.back, false);
}

@Test
public void testForwardIterator() {
    list.insertBack("Peter");
    list.insertBack("Paul");
    list.insertBack("Mary");
    String[] expected = {"Peter", "Paul", "Mary"};
    int current = 0;
    Iterator<String> i = list.forwardIterator();
    while (i.valid()) {
        String e = i.get();
        assertEquals(expected[current], e);
        i.next();
        current += 1;
    }
    assertEquals(3, current);
}
```

Iterator Interface

```
public Iterator<T> forwardIterator() {
    return new NodeListIterator<T>(this.front, true);
}

public Iterator<T> backwardIterator() {
    return new NodeListIterator<T>(this.back, false);
}

@Test
public void testBackwardIterator() {
    list.insertBack("Peter");
    list.insertBack("Paul");
    list.insertBack("Mary");
    String[] expected = {"Mary", "Paul", "Peter"};
    int current = 0;
    Iterator<String> i = list.backwardIterator();
    while (i.valid()) {
        String e = i.get();
        assertEquals(expected[current], e);
        i.next();
        current += 1;
    }
    assertEquals(3, current);
}
```

Java Iterators

http://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html

java.util

Interface Iterator<E>

All Known Subinterfaces:
ListIterator<E>, XMLEventReader

All Known Implementing Classes:
BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner

```
public interface Iterator<E>
```

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since:
1.2

See Also:
`Collection`, `ListIterator`, `Enumeration`

Method Summary	
boolean	**hasNext()** Returns `true` if the iteration has more elements.
E	**next()** Returns the next element in the iteration.
void	**remove()** Removes from the underlying collection the last element returned by the iterator (optional operation).

Java Iterators

http://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html

java.util

Interface Iterator<E>

All Known Subinterfaces:

ListIterator<E>, XMLEventReader

All Known Implementing Classes:

BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner

```
for (MyType obj : list) {
    System.out.print(obj);
}
```

ffer from

fined

Since:

1.2

See Also:

Collection, ListIterator, Enumeration

Will become even more important for more complex data structures to simplify certain operations like printing every element in sorted order

void	remove() Removes from the underlying collection the last element returned by the iterator (optional operation).

Java Iterable

https://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html

Interface Iterable<T>

Type Parameters:

`T` - the type of elements returned by the iterator

All Known Subinterfaces:

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Collection<E>, Deque<E>, DirectoryStream<T>, List<E>, NavigableSet<E>, Path, Queue<E>, SecureDirectoryStream<T>, Set<E>, SortedSet<E>, TransferQueue<E>

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BatchUpdateException, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DataTruncation, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, RowSetWarning, SerialException, ServiceLoader, SQLClientInfoException, SQLDataException, SQLException, SQLFeatureNotSupportedException, SQLIntegrityConstraintViolationException, SQLInvalidAuthorizationSpecException, SQLNonTransientConnectionException, SQLNonTransientException, SQLRecoverableException, SQLSyntaxErrorException, SQLTimeoutException, SQLTransactionRollbackException, SQLTransientConnectionException, SQLTransientException, SQLWarning, Stack, SyncFactoryException, SynchronousQueue, SyncProviderException, TreeSet, Vector

---

```
public interface Iterable<T>
```

Implementing this interface allows an object to be the target of the "foreach" statement.

Since:

1.5

Method Summary

Methods

Modifier and Type	Method and Description
`Iterator<T>`	`iterator()` Returns an iterator over a set of elements of type T.

Method Detail

iterator

```
Iterator<T> iterator()
```

Returns an iterator over a set of elements of type T.

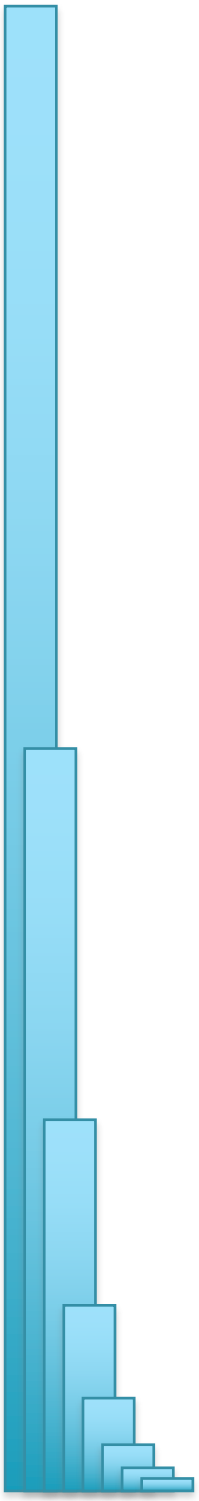
Returns:

an Iterator.

List v7

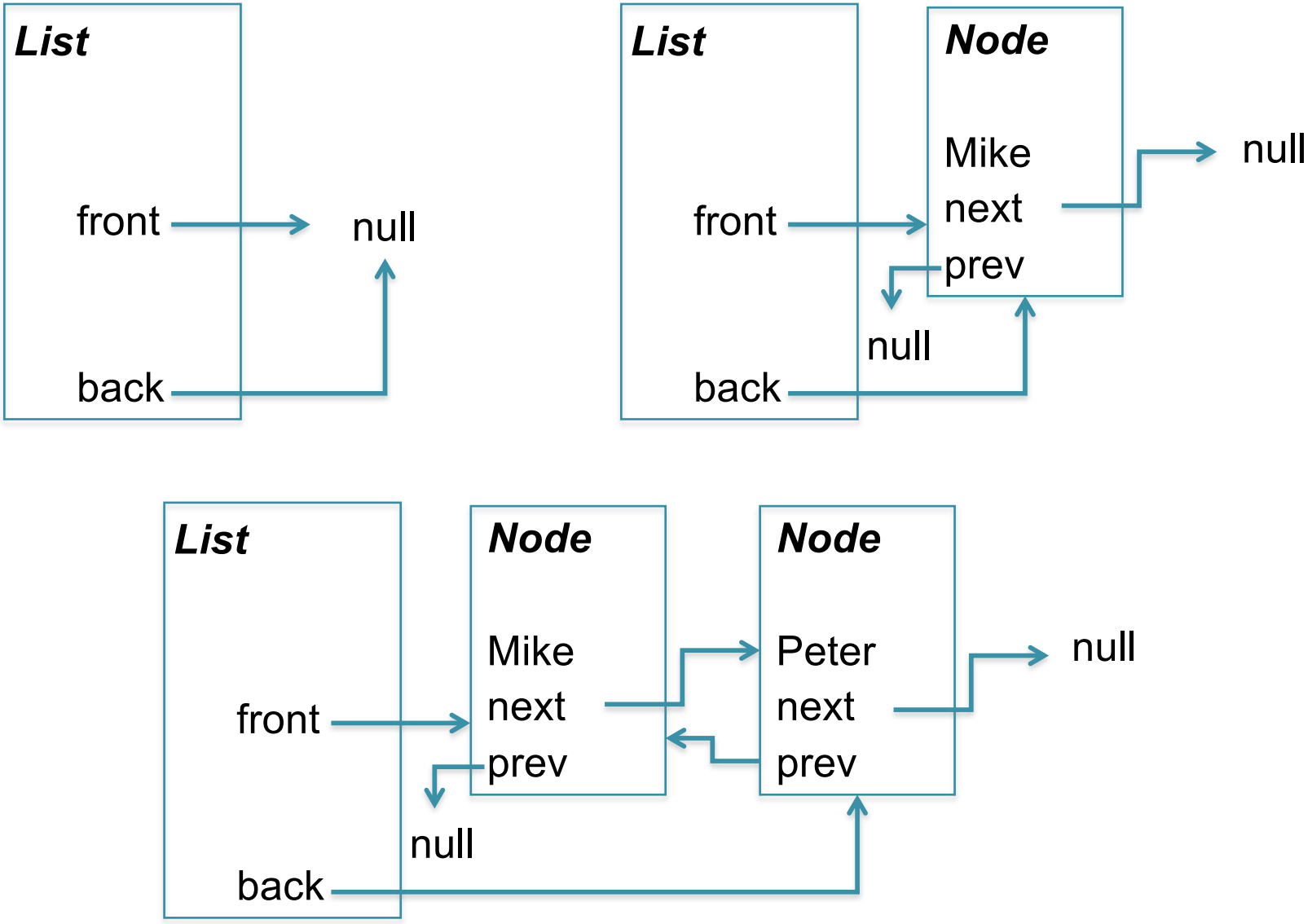
```
public interface List<T> extends Iterable<T> {
    ...
    private static class NodeListIterator<T>
                                        implements Iterator<T> {
    ...
}
```

```
Iterator<String> i = list.iterator();
while (i.hasNext()) {
    String e = i.next();
    // do something with element e
}


for (String e: list) {
    // do something with element e
}
```



Nulls and Sentinels

Living in a null world

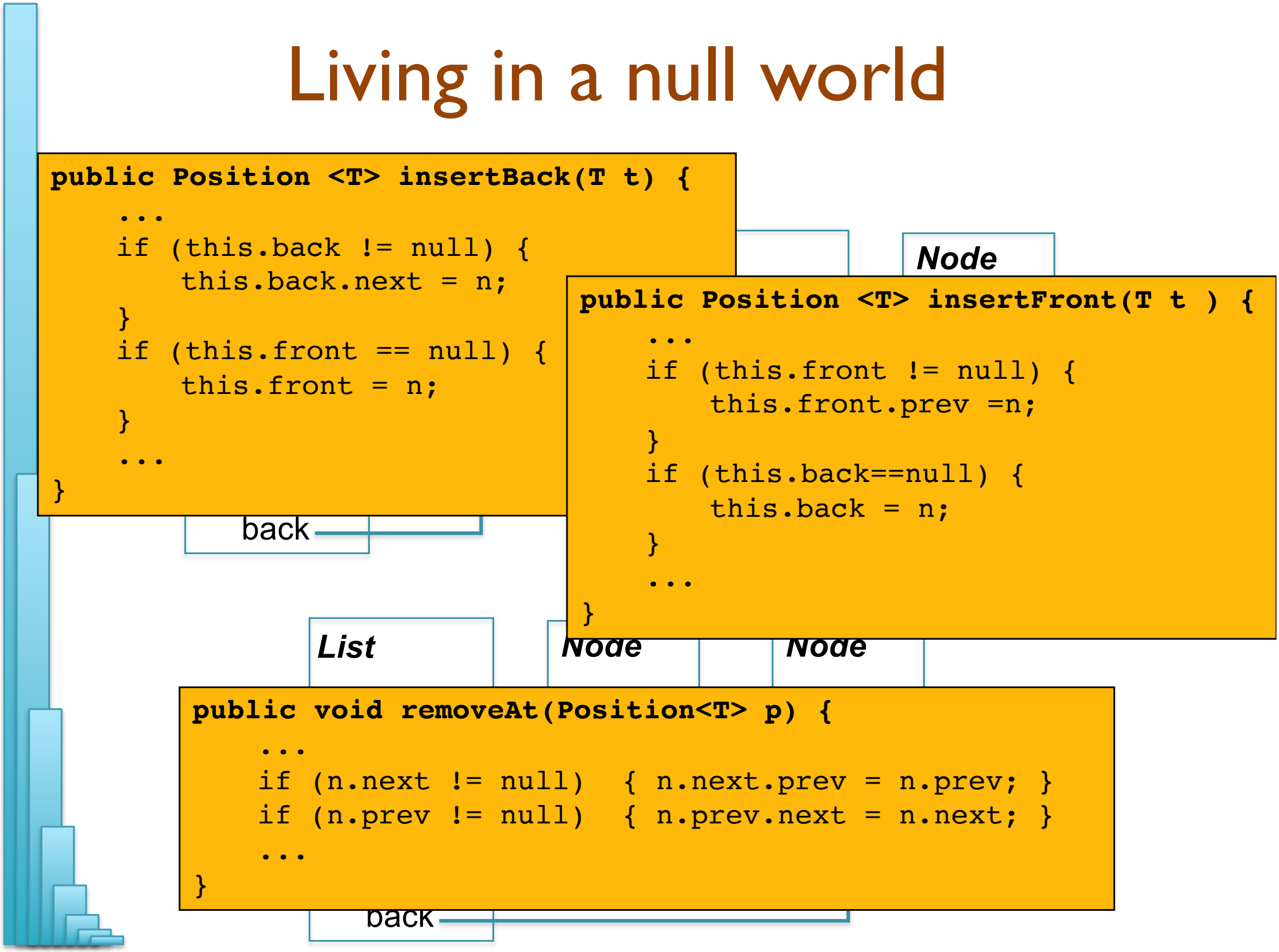


Living in a null world

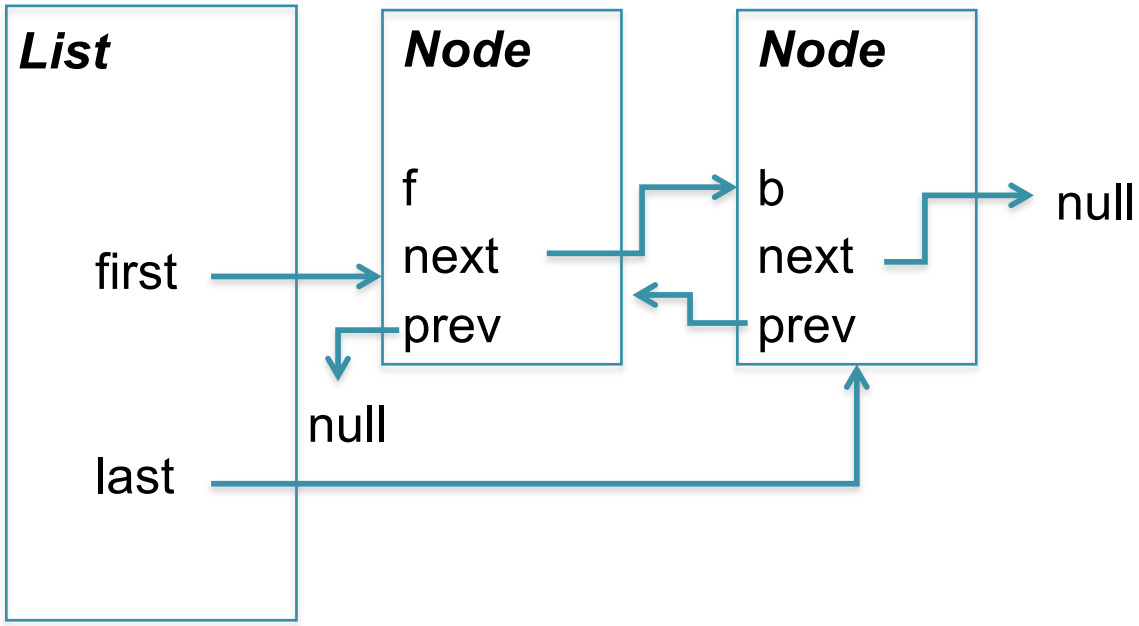
```
public Position <T> insertBack(T t) {
    ...
    if (this.back != null) {
        this.back.next = n;
    }
    if (this.front == null) {
        this.front = n;
    }
    ...
}
```

```
public Position <T> insertFront(T t ) {
    ...
    if (this.front != null) {
        this.front.prev =n;
    }
    if (this.back==null) {
        this.back = n;
    }
    ...
}
```

```
public void removeAt(Position<T> p) {
    ...
    if (n.next != null)  { n.next.prev = n.prev; }
    if (n.prev != null)  { n.prev.next = n.next; }
    ...
}
```

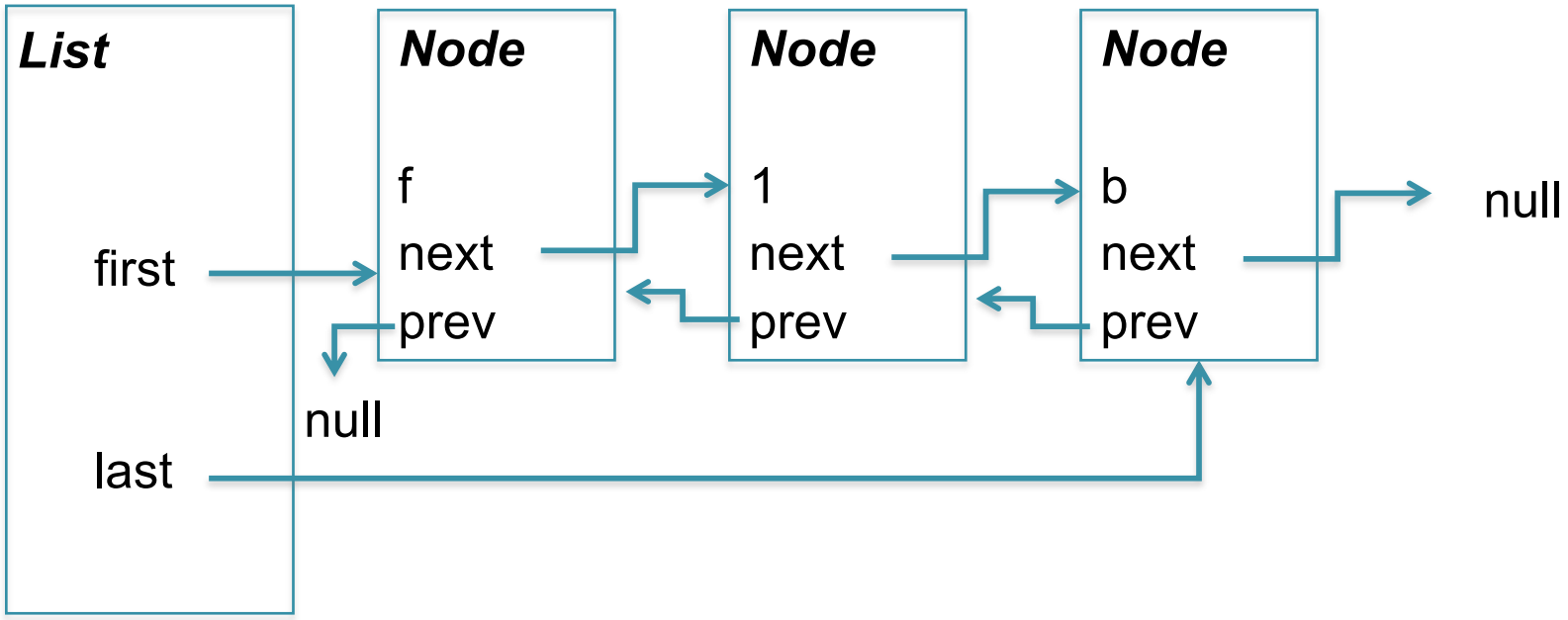


Doubly Linked List with Sentinels



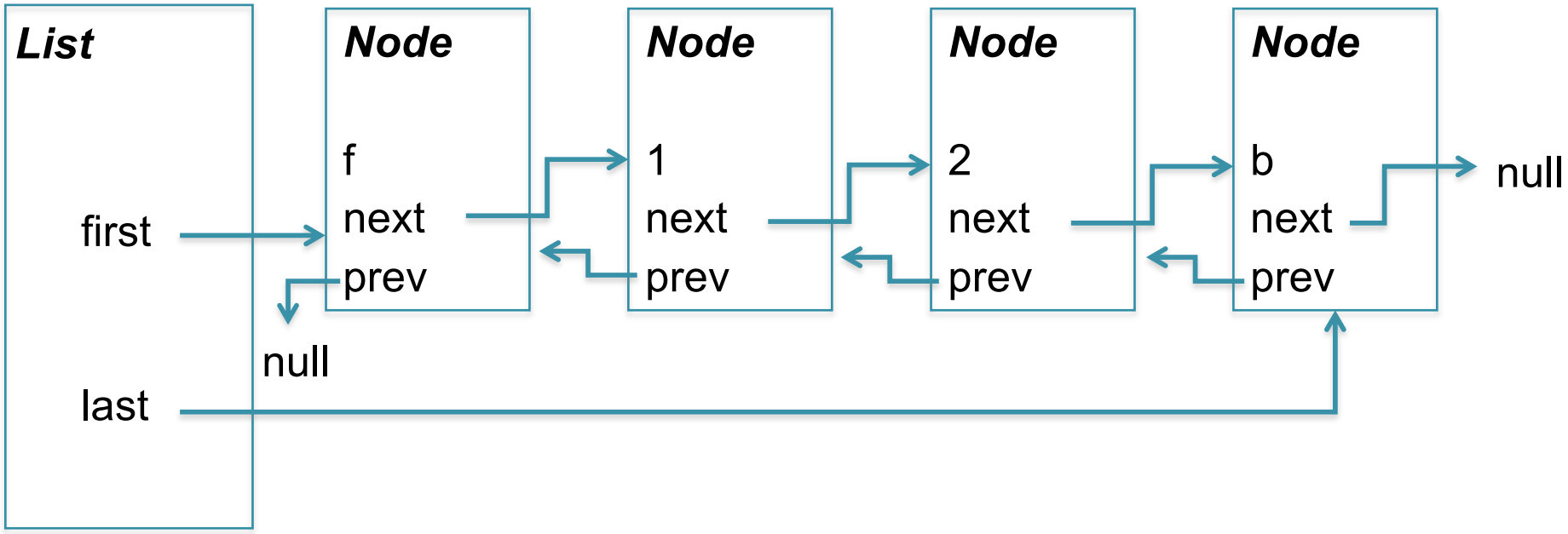
An “empty” list is initialized with the special front and back sentinels

Doubly Linked List with Sentinels



The user data is placed in between sentinels

Doubly Linked List with Sentinels



For the cost of a tiny bit of extra memory, the code gets significantly simpler!

Next Steps

1. Work on HW3

2. Check on Piazza for tips & corrections!