Advanced Object-oriented Programming

Lecture 26

Class Invariants
A good program is a correct program.

Recall from Lecture 7 how a **class invariant** expressed (a part of) the correctness of our implementation of a queue (using two pointers, to the start and end of the queue):

\[ 0 \leq \text{startIndex} \]

which ensures that the `next()` method doesn’t throw `ArrayIndexOutOfBoundsException`. 
Class Invariants and Correctness

A good program is a correct program.

Recall from Lecture 7 how a class invariant expressed (a part of) the correctness of our implementation of a queue (using two pointers, to the start and end of the queue):

\[ 0 \leq \text{startIndex} \]

which ensures that the next() method doesn’t throw ArrayIndexOutOfBoundsExceptions.
A good program is a correct program.

Recall from Lecture 7 how a class invariant expressed (a part of) the correctness of our implementation of a queue (using two pointers, to the start and end of the queue):

$$0 \leq \text{startIndex}$$

which ensures that the next() method doesn’t throw ArrayIndexOutOfBoundsExceptions.
A **class invariant** is some property that is always true of all instances of the class in question.

Strictly speaking, we should speak of a **class invariant for a class C**.

i.e., some statement, probably talking about the fields of the class, that might be true or false.

We’re getting a bit formal now, so we have to allow for pathological cases like ‘2 + 2 = 4’, which is a statement that will always be true for any class.
A class invariant is some property that is always true of all instances of the class in question.

Strictly speaking, we should speak of a class invariant for a class C.

i.e., some statement, probably talking about the fields of the class, that might be true or false.

We’re getting a bit formal now, so we have to allow for pathological cases like ‘2 + 2 = 4’, which is a statement that will always be true for any class.
A **class invariant** is some **property** that is always true of all instances of the class in question.

Strictly speaking, we should speak of a class invariant for a class C.

i.e., some statement, probably talking about the fields of the class, that might be true or false.

We’re getting a bit formal now, so we have to allow for pathological cases like ‘2 + 2 = 4’, which is a statement that will always be true for any class.
A **class invariant** is some property that is always true of all instances of the class in question.

Strictly speaking, we should speak of a class invariant *for a class C*. i.e., some statement, probably talking about the fields of the class, that might be true or false.

We’re getting a bit formal now, so we have to allow for pathological cases like ‘2 + 2 = 4’, which is a statement that will always be true for any class.
Example of a Statement

Recall the **Point** class:

```java
class Point {
    private int xCoord;
    private int yCoord;

    public Point(int x, int y) {
        xCoord = x;
        yCoord = y;
    }

    public void move(int dx, int dy) {
        xCoord += dx;
        yCoord += dy;
    }
}
```
Example of a Statement

(the class also has two accessor methods \texttt{getX()} and \texttt{getY()})

One possible statement we might make about \texttt{Point} instances is:

\[ 0 \leq \text{Coord} \quad \text{and} \quad 0 \leq \text{yCoord} \]

If we have an instance \( p \) of type \texttt{Point}, this statement will be true \textit{for} \( p \) if

\[ 0 \leq p.\text{Coord} \quad \text{and} \quad 0 \leq p.\text{yCoord} \]

Note that our notion of truth is relative to instances
Example of a Statement

(the class also has two accessor methods \texttt{getX()} and \texttt{getY()});

One possible statement we might make about \texttt{Point} instances is:

\[ 0 \leq x\text{Coord} \quad \text{and} \quad 0 \leq y\text{Coord} \]

If we have an instance \( p \) of type \texttt{Point}, this statement will be true \textit{for} \( p \) if

\[ 0 \leq p.x\text{Coord} \quad \text{and} \quad 0 \leq p.y\text{Coord} \]

Note that our notion of truth is relative to instances.
Example of a Statement

(the class also has two accessor methods `getX()` and `getY()`)

One possible statement we might make about `Point` instances is:

\[ 0 \leq x\text{Coord} \quad \text{and} \quad 0 \leq y\text{Coord} \]

If we have an instance `p` of type `Point`, this statement will be true for `p` if

\[ 0 \leq p.\text{xCoord} \quad \text{and} \quad 0 \leq p.\text{yCoord} \]

Note that our notion of truth is relative to instances
Example of a Statement

(the class also has two accessor methods `getX()` and `getY()`)

One possible statement we might make about `Point` instances is:

\[ 0 \leq x\text{Coord} \quad \text{and} \quad 0 \leq y\text{Coord} \]

If we have an instance `p` of type `Point`, this statement will be true for `p` if

\[ 0 \leq p.x\text{Coord} \quad \text{and} \quad 0 \leq p.y\text{Coord} \]

Note that our notion of truth is relative to instances
For Example

Some instances of `Point`:

```java
Point p;
p = new Point(12, 348);
p = new Point(1, -5);
p = new Point(2, 56);
p.move(-3, 22);
p.move(1, 0);
```

The statement is true for `p`.

The statement is false for `p`.

Clearly, this statement is not true for all instances of class `Point`, and so is not a class invariant.
For Example

For Example

some instances of Point

Point p;
p = new Point(12, 348);
p = new Point(1,-5);
p = new Point(2, 56)
p.move(-3, 22);
p.move(1, 0);

the statement is true for p

the statement is false for p

Clearly, this statement is not true for all instances of class Point, and so is not a class invariant.
For Example

some instances of Point

```java
Point p;
p = new Point(12, 348);
p = new Point(1,-5);
p = new Point(2, 56)
p.move(-3, 22);
p.move(1, 0);
```

the statement is true for p

the statement is false for p

Clearly, this statement is not true for all instances of class Point, and so is not a class invariant.
For Example

```java
Point p;
p = new Point(12, 348);
p = new Point(1,-5);
p = new Point(2, 56)
p.move(-3, 22);
p.move(1, 0);
```

the statement is true for p

the statement is false for p

Clearly, this statement is not true for all instances of class Point, and so is not a class invariant.
For Example

```java
Point p;
p = new Point(12, 348);
p = new Point(1, -5);
p = new Point(2, 56)
p.move(-3, 22);
p.move(1, 0);
```

the statement is true for p

the statement is false for p

Clearly, this statement is not true for all instances of class Point, and so is not a class invariant.
For Example

```java
Point p;
p = new Point(12, 348);
p = new Point(1, -5);
p = new Point(2, 56)
p.move(-3, 22);
p.move(1, 0);
```

the statement is **true** for `p`

the statement is **false** for `p`

Clearly, this statement is not true for all instances of class `Point`, and so is not a class invariant.
For Example

```java
Point p;
p = new Point(12, 348);
p = new Point(1,-5);
p = new Point(2, 56)
p.move(-3, 22);
p.move(1, 0);
```

the statement is true for `p`
the statement is false for `p`

Clearly, this statement is not true for all instances of class `Point`, and so is not a class invariant.
Consider another class:

class NonNegPoint {
    private int xCoord;
    private int yCoord;

    public NonNegPoint() {
        xCoord = 0;
        yCoord = 0;
    }

    public void move(int dx, int dy) {
        xCoord += Math.max(dx, -dx);
        yCoord += Math.max(dy, -dy);
    }
}

For this class (NonNegPointP)

\[ 0 \leq x\text{Coord} \quad \text{and} \quad 0 \leq y\text{Coord} \]

is a class invariant.

The property is true when a NonNegPoint instance is created, and it remains true after any method is called (there is only one method in this example).
For Example

```java
NonNegPoint p;
p = new NonNegPoint();
p.move(-1, -1);
p.move(2, -22);
```

Clearly, there is nothing we can do to get a negative value for `xCoord` or `yCoord`. 
For Example

```java
NonNegPoint p;
p = new NonNegPoint();
p.move(-1, -1);
p.move(2, -22);
```

Clearly, there is nothing we can do to get a negative value for `xCoord` or `yCoord`.
For Example

```java
NonNegPoint p;
p = new NonNegPoint();
p.move(-1, -1);
p.move(2, -22);
p.xCoord = 0;
p.yCoord = 0
p.xCoord = 1;
p.yCoord = 1
p.xCoord = 3;
p.yCoord = 23
```

Clearly, there is nothing we can do to get a negative value for xCoord or yCoord.
For Example

```java
NonNegPoint p;
p = new NonNegPoint();
p.move(-1, -1);
p.move(2, -22);
```

\[p.xCoord = 0; p.yCoord = 0\]
\[p.xCoord = 1; p.yCoord = 1\]
\[p.xCoord = 3; p.yCoord = 23\]

Clearly, there is nothing we can do to get a negative value for `xCoord` or `yCoord`. 
For Example

```java
NonNegPoint p;
p = new NonNegPoint();
p.move(-1, -1);
p.move(2, -22);
```

Clearly, there is nothing we can do to get a negative value for `xCoord` or `yCoord`. 
We can check that a property is a class invariant by checking:

1. each constructor makes the property true; and
2. each public method ‘preserves’ the property
   i.e., if it is true when the method is called, then it is true when the method has finished.
Checking Invariance

We can check that a property is a class invariant by checking:

1. each constructor makes the property true; and
2. each public method ‘preserves’ the property
   i.e., if it is true when the method is called,
   then it is true when the method has finished.
Checking Invariance

The constructor makes the property true:

```java
public NonNegPoint() {
    xCoord = 0;
    yCoord = 0;
    // 0 <= xCoord and 0 <= yCoord
}
```
Checking Invariance

The constructor makes the property true:

```java
public NonNegPoint() {
    xCoord = 0;
    yCoord = 0;
    // 0 <= xCoord and 0 <= yCoord
}
```
Checking Invariance

...and is preserved by the one public method:

```java
public void move(int dx, int dy) {
    // if 0 <= xCoord and 0 <= yCoord here
    xCoord += Math.max(dx, -dx);
    yCoord += Math.max(dy, -dy);
    // then 0 <= xCoord and 0 <= yCoord here
}
```
Checking Invariance

...and is preserved by the one public method:

```java
public void move(int dx, int dy) {
    // if 0 <= xCoord and 0 <= yCoord here
    xCoord += Math.max(dx, -dx);
    yCoord += Math.max(dy, -dy);
    // then 0 <= xCoord and 0 <= yCoord here
}
```
Checking Invariance

...and is preserved by the one public method:

```java
public void move(int dx, int dy) {
    // if 0 <= xCoord and 0 <= yCoord here
    xCoord += Math.max(dx, -dx);
    yCoord += Math.max(dy, -dy);
    // then 0 <= xCoord and 0 <= yCoord here
}
```

in Class NonNegPoint
Invariance for Doubly-linked Lists

A class invariant for LList (doubly-linked lists) is that the list of BiNodes is well-formed:

i.e., going along a tail pointer, then along a prev pointer takes you back to where you started (and vice-versa).
Well-formed Lists

In more detail,

- **null** is well-formed
- a non-null **BiNode** \( b \) is well-formed if
  - \( b.\text{prev} \) is null, or
  - \( b.\text{prev} \) is not null, and
  - \( b.\text{prev} \) is well-formed and
  - \( b.\text{prev.\text{tail}} \) is \( b \), and
  - \( b.\text{tail} \) is null, or
  - \( b.\text{tail} \) is not null, and
  - \( b.\text{tail} \) is well-formed and
  - \( b.\text{tail.\text{prev}} \) is \( b \)
A Non-well-formed List
Another invariant for LList is that listStart points to the start of the list, and listEnd points to the end of the list.

In more detail, either:

- listStart = listEnd = null, or
- both listStart and listEnd are not null and
  - listStart.prev = null and
  - listEnd.tail = null
LList Class Invariant

In its full glory:

- listStart is well-formed and
  - listStart = listEnd = null, or
  - both listStart and listEnd are not null and
    - listStart.prev = null and
    - listEnd.tail = null
Checking Invariance

Recall that \texttt{LList\langle A\rangle} had only the ‘default’ constructor:

```java
public LList() {
}
```

Both \texttt{listStart} and \texttt{listEnd} are \texttt{null} — and \texttt{null} is well-formed. So the constructor makes the class invariant true.
Checking Invariance

Recall that `LList<A>` had only the ‘default’ constructor:

```java
public LList() {
}
```

Both `listStart` and `listEnd` are `null` — and `null` is well-formed.

So the constructor makes the class invariant true.
Checking Invariance

Let’s look at the public method `add(A)`:

```java
public void add(A b) {
    // assuming listStart is well-formed
    // and ... here
    listStart = new BiNode(b, listStart);
    if (listEnd == null) {
        listEnd = listStart;
    }
    // we need to know listStart is well-formed
    // and ... here
}
```
Let's start at the beginning:

```java
listStart = new BiNode(b, listStart);
```

**BiNode constructor**

```java
BiNode(A b, BiNode t) {
    value = b;
    tail = t;
    if (tail != null) {
        tail.prev = this;
    }
    // prev = null
}
```

By assumption, listStart was well-formed, so the tail of the new BiNode is well-formed; so is the prev BiNode . . .
Let’s start at the beginning:

```java
listStart = new BiNode(b, listStart);
```

BiNode constructor

```java
BiNode(A b, BiNode t) {
    value = b;
    tail = t;
    if (tail != null) {
        tail.prev = this;
    }
    // prev = null
}
```

By assumption, `listStart` was well-formed, so the tail of the new `BiNode` is well-formed; so is the `prev` `BiNode` . . .
Let’s start at the beginning:

```java
listStart = new BiNode(b, listStart);
```

BiNode constructor

```java
BiNode(A b, BiNode t) {
    value = b;
    tail = t;
    if (tail != null) {
        tail.prev = this;
    }
    // prev = null
}
```

By assumption, `listStart` was well-formed, so the tail of the new `BiNode` is well-formed; so is the `prev` `BiNode` . . .
Let's start at the beginning:

```java
listStart = new BiNode(b, listStart);
```

**BiNode constructor**

```java
BiNode(A b, BiNode t) {
    value = b;
    tail = t;
    if (tail != null) {
        tail.prev = this;
    }
    // prev = null
}
```

By assumption, `listStart` was well-formed, so the tail of the new `BiNode` is well-formed; so is the `prev` `BiNode` ...
Checking Invariance

...and the if-statement:

```java
BiNode(A b, BiNode t) {
  value = b;
  tail = t;
  if (tail != null) {
    tail.prev = this;
  }
}
```

makes sure that the new BiNode is well-formed.
Checking Invariance

...and the if-statement:

```java
BiNode(A b, BiNode t) {
    value = b;
    tail = t;
    if (tail != null) {
        tail.prev = this;
    }
}
```

makes sure that the new BiNode is well-formed.
Checking Invariance

```java
public void add(A b) {
    // if listStart is well-formed here
    listStart = new BiNode(b, listStart);
    // then listStart is well-formed here

    if (listEnd == null) {
        listEnd = listStart;
    }
    // so listStart is well-formed here
}
```

because the if-statement doesn’t change listStart
Checking Invariance

```java
public void add(A b) {
    // if listStart is well-formed here
    listStart = new BiNode(b, listStart);
    // then listStart is well-formed here
    if (listEnd == null) {
        listEnd = listStart;
    }
    // so listStart is well-formed here
}
```

because the if-statement doesn’t change listStart
public void add(A b) {
    // if listStart is well-formed here
    listStart = new BiNode(b, listStart);
    // then listStart is well-formed here
    if (listEnd == null) {
        listEnd = listStart;
    }
    // so listStart is well-formed here
}

because the if-statement doesn’t change listStart
Checking Invariance

```java
public void add(A b) {
    ... 
    if (listEnd == null) {
        // then listStart was null
        listEnd = listStart;
        // listEnd.tail = null
    }
}
```

because `listStart`'s previous value is the value of the new BiNode's tail.
Moreover, the new BiNode's prev is null.
So the invariant is preserved!
Checking Invariance

```java
public void add(A b) {
    ...
    if (listEnd == null) {
        // then listStart was null
        listEnd = listStart;
        // listEnd.tail = null
    }
}
```

because `listStart`'s previous value is the value of the new `BiNode`'s tail.
Moreover, the new `BiNode`'s `prev` is null.
So the invariant is preserved!
Checking Invariance

public void add(A b) {
    ...

    if (listEnd == null) {
        // then listStart was null
        listEnd = listStart;
        // listEnd.tail = null
    }
}

because listStart’s previous value is the value of the new BiNode’s tail.
Moreover, the new BiNode’s prev is null.
So the invariant is preserved!
Checking Invariance

```java
public void add(A b) {
    ...
    if (listEnd == null) {
        // then listStart was null
        listEnd = listStart;
        // listEnd.tail = null
    }
}
```

because `listStart`'s previous value is the value of the new `BiNode`'s tail.
Moreover, the new `BiNode`'s `prev` is null.

So the invariant is preserved!
Checking Invariance

```java
public void add(A b) {
    ...
    if (listEnd == null) {
        // then listStart was null
        listEnd = listStart;
        // listEnd.tail = null
    }
}
```

because `listStart`’s previous value is the value of the new `BiNode`’s tail.
Moreover, the new `BiNode`’s prev is null.
So the invariant is preserved!
Checking Invariance

... and so we continue through all the other public methods ...

So far, the notion of class invariant just seems like hard work:
- formulating the invariant, then
- checking it.

But can class invariants help the programmer?
... and so we continue through all the other public methods ... 

So far, the notion of class invariant just seems like hard work:
- formulating the invariant, then
- checking it.

But can class invariants help the programmer?
Checking Invariance

... and so we continue through all the other public methods ...

So far, the notion of class invariant just seems like hard work:

- formulating the invariant, then
- checking it.

But can class invariants help the programmer?
Checking Invariance

... and so we continue through all the other public methods ...

So far, the notion of class invariant just seems like hard work:

- formulating the invariant, then
- checking it.

But can class invariants help the programmer?
 Checking Invariance

... and so we continue through all the other public methods ...

So far, the notion of class invariant just seems like hard work:
  - Formulating the invariant, then
  - Checking it.

But can class invariants help the programmer?
Removing a BiNode

Let’s write a method that will remove the i-th BiNode from a list.

```java
/**
 * Remove the i-th element from the list (counting from 0). If i is greater than
 * the length of the list, no element is removed.
 * @param i
 * @param i
 * the index of the element to remove
 */
public void remove(int i) {
}
```
Removing a BiNode

Let’s write a method that will remove the i-th BiNode from a list.

```java
/**
 * Remove the i-th element from the list
 * (counting from 0). If i is greater than
 * the length of the list, no element is
 * removed.
 * @param i
 * @param the index of the element to remove
 */
public void remove(int i) {
}
```
public void remove(int i) {
    // find the i-th element
    BiNode n = listStart;
    int count = 0;
    while (n != null) {
        if (count == i) {
            // remove the current node
            ...
            return;
        }
        count++;
        n = n.tail();
    }
    count++;  
n = n.tail();
}
Removing a BiNode

public void remove(int i) {
    // find the i-th element
    BiNode n = listStart;
    int count = 0;
    while (n != null) {
        if (count == i) {
            // remove the current node
            ...
            return;
        }
        count++;
        n = n.tail();
    }
}
Removing a BiNode

in class LList

```java
public void remove(int i) {
    // find the i-th element
    BiNode n = listStart;
    int count = 0;
    while (n != null) {
        if (count == i) {
            // remove the current node
            ...
            return;
        }
        count++;
        n = n.tail();
    }
}
```
public void remove(int i) {
    // find the i-th element
    BiNode n = listStart;
    int count = 0;
    while (n != null) {
        if (count == i) {
            // remove the current node
            ...
            return;
        }
        count++;
        n = n.tail();
    }
}
Removing a BiNode

We can ‘remove’ the middle BiNode:

...by rearranging the pointers:
Removing a BiNode

We can ‘remove’ the middle BiNode:

...by rearranging the pointers:
The Class Invariant

The class invariant tells us we need to ensure:

- \( b.\text{prev} \) is null, or
  - \( b.\text{prev} \) is not null, and
    - \( b.\text{prev}.\text{tail} \) is \( b \), and
- \( b.\text{tail} \) is null, or
  - \( b.\text{tail} \) is not null, and
    - \( b.\text{tail}.\text{prev} \) is \( b \)

for each node \( b \)

So we should check whether the nodes on either side of \( n \) are null.
If they’re both not null, we’ll rearrange the pointers as required (\( b.\text{tail}.\text{prev} \) is \( b \), etc.)
The Class Invariant

The class invariant tells us we need to ensure:

from the definition of well-formed

- b.prev is null, or
  - b.prev is not null, and
    - b.prev.tail is b, and
- b.tail is null, or
  - b.tail is not null, and
    - b.tail.prev is b

for each node b

So we should check whether the nodes on either side of n are null.

If they’re both not null, we’ll rearrange the pointers as required (b.tail.prev is b, etc.)
The Class Invariant

The class invariant tells us we need to ensure:

- \( b.\text{prev} \) is null, or
  - \( b.\text{prev} \) is not null, and
  - \( b.\text{prev}.\text{tail} \) is \( b \), and
- \( b.\text{tail} \) is null, or
  - \( b.\text{tail} \) is not null, and
  - \( b.\text{tail}.\text{prev} \) is \( b \)

for each node \( b \)

So we should check whether the nodes on either side of \( n \) are null.
If they’re both not null, we’ll rearrange the pointers as required (\( b.\text{tail}.\text{prev} \) is \( b \), etc.)
Considering Cases

Suppose the node to the left is null. Then \( n \) is the first node in the list.

Well-formedness isn’t an issue (remember we assume we started in a state where the list was well-formed, so we’re assuming that all the list to the right — which is the list that will remain after we remove \( n \) — is well-formed).

But the other part of the class invariant tells us:

<table>
<thead>
<tr>
<th>class invariant for LList</th>
</tr>
</thead>
<tbody>
<tr>
<td>- listStart = listEnd = null, or</td>
</tr>
<tr>
<td>- both listStart and listEnd are not null and</td>
</tr>
<tr>
<td>- listStart.prev = null and</td>
</tr>
<tr>
<td>- listEnd.tail = null</td>
</tr>
</tbody>
</table>
Considering Cases

Suppose the node to the left is \textit{null}.
Then \textit{n} is the first node in the list.

Well-formedness isn’t an issue (remember we assume we started in a state where the list was well-formed, so we’re assuming that all the list to the right — which is the list that will remain after we remove \textit{n} — is well-formed).

But the other part of the class invariant tells us:

\begin{itemize}
  \item listStart = listEnd = null, \textbf{or}
  \item both listStart and listEnd are not null \textbf{and}
    \begin{itemize}
      \item listStart.prev = null and
      \item listEnd.tail = null
    \end{itemize}
\end{itemize}
Considering Cases

Suppose the node to the left is `null`. Then `n` is the first node in the list.

Well-formedness isn’t an issue (remember we assume we started in a state where the list was well-formed, so we’re assuming that all the list to the right — which is the list that will remain after we remove `n` — is well-formed).

But the other part of the class invariant tells us:

<table>
<thead>
<tr>
<th>class invariant for LList</th>
</tr>
</thead>
<tbody>
<tr>
<td>• listStart = listEnd = null, or</td>
</tr>
<tr>
<td>• both listStart and listEnd are not null and</td>
</tr>
<tr>
<td>• listStart.prev = null and</td>
</tr>
<tr>
<td>• listEnd.tail = null</td>
</tr>
</tbody>
</table>
At the Start of the List

If \( n \) is also the \textit{end} of the list (i.e., there is just the one node in the list), this part of the invariant tells us we need to set both \texttt{listStart} and \texttt{listEnd} to \texttt{null}.

Otherwise, we need to ensure

\[
\text{listStart.prev} = \text{null}
\]

Note that, in the ‘otherwise’ case (there are nodes to the right),

\[
\text{listEnd.tail} = \text{null}
\]

holds \textit{by assumption}.
At the Start of the List

If \( n \) is also the end of the list (i.e., there is just the one node in the list), this part of the invariant tells us we need to set both listStart and listEnd to null.

Otherwise, we need to ensure

```python
listStart.prev = null
```

Note that, in the ‘otherwise’ case (there are nodes to the right),

```python
listEnd.tail = null
```

holds by assumption.
At the Start of the List

If $n$ is also the *end* of the list (i.e., there is just the one node in the list), this part of the invariant tells us we need to set both `listStart` and `listEnd` to `null`.

Otherwise, we need to ensure

```java
listStart.prev = null
```

Note that, in the ‘otherwise’ case (there are nodes to the right),

```java
listEnd.tail = null
```

holds *by assumption*.
Removing a BiNode

// remove the current node
if (n.prev() == null) { // start of list
    listStart = n.tail();
    if (n.tail() == null) {
        listEnd = null;
    } else {
        listStart.prev = null;
    }
}

By assumption, n.tail() is well-formed, so listStart is.
Also at the start of the list; listStart and listEnd are both null.
To ensure listStart.prev is null
Removing a BiNode

```java
// remove the current node
if (n.prev() == null) { // start of list
    listStart = n.tail();
    if (n.tail() == null) {
        listEnd = null;
    } else {
        listStart.prev = null;
    }
}
```

By assumption, `n.tail()` is well-formed, so `listStart` is. Also at the start of the list; `listStart` and `listEnd` are both `null`. To ensure `listStart.prev` is null
Removing a BiNode

```
// remove the current node
if (n.prev() == null) { // start of list
    listStart = n.tail();
    if (n.tail() == null) {
        listEnd = null;
    } else {
        listStart.prev = null;
    }
}
```

By assumption, `n.tail()` is well-formed, so `listStart` is.

Also at the start of the list; `listStart` and `listEnd` are both null. To ensure `listStart.prev` is null.
Removing a BiNode

```java
// remove the current node
if (n.prev() == null) {  // start of list
    listStart = n.tail();
    if (n.tail() == null) {
        listEnd = null;
    } else {
        listStart.prev = null;
    }
}
```

By assumption, `n.tail()` is well-formed, so `listStart` is.

Also at the start of the list; `listStart` and `listEnd` are both `null`.

To ensure `listStart.prev` is `null`
Removing a BiNode

```java
// remove the current node
if (n.prev() == null) { // start of list
    listStart = n.tail();
    if (n.tail() == null) {
        listEnd = null;
    } else {
        listStart.prev = null;
    }
} }
```

By assumption, `n.tail()` is well-formed, so `listStart` is. Also at the start of the list; `listStart` and `listEnd` are both `null`. To ensure `listStart.prev` is `null`
Case: End of List

Similarly, if we’re at the end of the list (but not the start: this is an ‘else’ case), then *by assumption of the class invariant*

- n.prev is well-formed
- listStart.prev (off to the right) is null

we only need to ensure that listEnd.tail is null.
Case: End of List

Similarly, if we’re at the end of the list (but not the start: this is an ‘else’ case), then by assumption of the class invariant

- n.prev is well-formed
- listStart.prev (off to the right) is null

we only need to ensure that listEnd.tail is null.
Case: End of List

Similarly, if we’re at the end of the list (but not the start: this is an ‘else’ case), then by *assumption of the class invariant*

- \( n.\text{prev} \) is well-formed
- \( \text{listStart.prev} \) (off to the right) is \text{null}

we only need to ensure that \( \text{listEnd.tail} \) is \text{null}. 
Case: End of List

Similarly, if we’re at the end of the list (but not the start: this is an ‘else’ case), then by assumption of the class invariant

- \( n.\text{prev} \) is well-formed
- \( \text{listStart.prev} \) (off to the right) is null

we only need to ensure that \( \text{listEnd.tail} \) is null.
‘Else’ Case: End of the List

```java
else if (n.tail == null) {
    // end of list; nodes to left
    listEnd = n.prev();
    listEnd.tail = null;
}
```

We know this isn’t `null`…

…so `listEnd.tail` won’t throw a `NullPointerException`. 
‘Else’ Case: End of the List

```java
else if (n.tail == null) {
   // end of list; nodes to left
   listEnd = n.prev();
   listEnd.tail = null;
}
```

We know this isn’t `null`...

...so `listEnd.tail` won’t throw a `NullPointerException`. 
‘Else’ Case: End of the List

```java
else if (n.tail == null) {
    // end of list; nodes to left
    listEnd = n.prev();
    listEnd.tail = null;
}
```

We know this isn’t `null`...

...so `listEnd.tail` won’t throw a `NullPointerException`. 
Final Case: in media res

Otherwise: there are nodes to left and right.

By assumption:
- off to the left:
  - n.prev is well-formed; and
  - listStart.prev is null
- off to the right:
  - n.tail is well-formed; and
  - listEnd.tail is null

We therefore need only concern ourselves with well-formedness. The requirement is:

- from the definition of well-formed
  - b.prev.tail is b, and
  - b.tail.prev is b

for each node b on either side of n.
Final Case: in media res

Otherwise: there are nodes to left and right.

By assumption:

- off to the left:
  - \( n.\text{prev} \) is well-formed; and
  - \( \text{listStart}.\text{prev} \) is null

- off to the right:
  - \( n.\text{tail} \) is well-formed; and
  - \( \text{listEnd}.\text{tail} \) is null

We therefore need only concern ourselves with well-formedness. The requirement is:

from the definition of well-formed

- \( b.\text{prev}.\text{tail} \) is \( b \), and
- \( b.\text{tail}.\text{prev} \) is \( b \)

for each node \( b \) on either side of \( n \).
Final Case: in media res

Otherwise: there are nodes to left and right.

By assumption:

- off to the left:
  - \( n\.prev \) is well-formed; and
  - \( \text{listStart}.prev \) is null

- off to the right:
  - \( n\.tail \) is well-formed; and
  - \( \text{listEnd}.tail \) is null

We therefore need only concern ourselves with well-formededness. The requirement is:

from the definition of well-formed

- \( b\.prev\.tail \) is \( b \), and
- \( b\.tail\.prev \) is \( b \)

for each node \( b \) on either side of \( n \).
Final Case: in media res

Otherwise: there are nodes to left and right.

By assumption:

- off to the left:
  - \( n.\text{prev} \) is well-formed; and
  - \( \text{listStart}.\text{prev} \) is null

- off to the right:
  - \( n.\text{tail} \) is well-formed; and
  - \( \text{listEnd}.\text{tail} \) is null

We therefore need only concern ourselves with well-formedness. The requirement is:

from the definition of well-formed

- \( b.\text{prev}.\text{tail} \) is \( b \), and
- \( b.\text{tail}.\text{prev} \) is \( b \)

for each node \( b \) on either side of \( n \).
else {
    // nodes to left and right
    // cut out n from the left
    n.prev().tail = n.tail();
    // cut out n from the right
    n.tail().prev = n.prev();
}

// done!
else {
  // nodes to left and right
  // cut out n from the left
  n.prev().tail = n.tail();

  // cut out n from the right
  n.tail().prev = n.prev();
}

// done!
else {
  // nodes to left and right
  // cut out n from the left
  n.prev().tail = n.tail();
  // cut out n from the right
  n.tail().prev = n.prev();
}

// done!
else {
    // nodes to left and right
    // cut out n from the left
    n.prev().tail = n.tail();
    // cut out n from the right
    n.tail().prev = n.prev();
}
// done!
Done!
That’s All, Folks!

Summary

- Class Invariants . . .
- . . . crystallise design

Next:

More Invariants