

Verification of OpenJDK's LinkedList using KeY

Frank de Boer
Marko van Eekelen
Stijn de Gouw
Hans-Dieter Hiep
Olaf Maathuis

Outline

1. What is a doubly-linked list?
2. There is a bug in Java's `LinkedList`
3. Modular specification of Java's Collection Framework
4. On-going work on verification in KeY
5. The bug is 20 years old
6. Related work

Assumptions

In the following:

- ▶ Forget about generics
⇒ treat as with type erasure
- ▶ Forget about multi-threading
⇒ treat as there is only one thread
- ▶ Forget about user-defined equality
⇒ treat as there is only referential equality ==

What is a doubly-linked list?

What is a doubly-linked list?

A linked list is a list of elements kept in nodes chained together.

Definition

A *node* structure consists of:

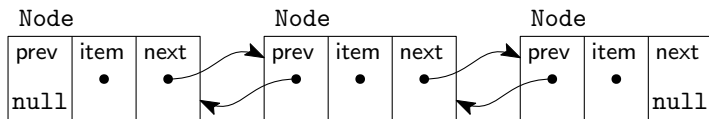
- ▶ a **prev** reference to a node,
- ▶ an **item** reference to an element,
- ▶ a **next** reference to a node.

References may be **null**.

Chains

Definition

A *chain* is a sequence of length $n > 0$ of nodes such that:



Remark

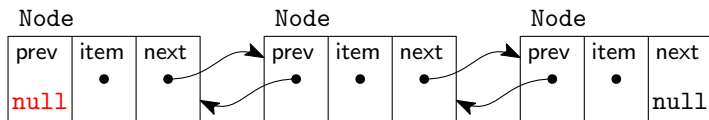
We treat sequences as 0-indexed: index 0 is the first element

Chains

Definition

A *chain* is a sequence of length $n > 0$ of nodes such that:

- ▶ the *prev* reference of the first node is **null**,



Remark

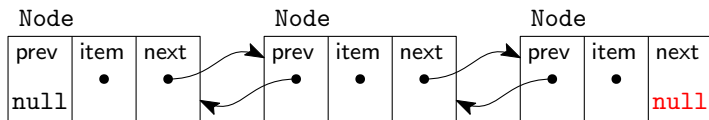
We treat sequences as 0-indexed: index 0 is the first element

Chains

Definition

A *chain* is a sequence of length $n > 0$ of nodes such that:

- ▶ the **prev** reference of the first node is **null**,
- ▶ the **next** reference of the last node is **null**,



Remark

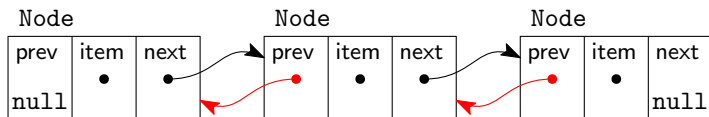
We treat sequences as 0-indexed: index 0 is the first element

Chains

Definition

A *chain* is a sequence of length $n > 0$ of nodes such that:

- ▶ the **prev** reference of the first node is **null**,
- ▶ the **next** reference of the last node is **null**,
- ▶ the **prev** of node i is node $i - 1$ for every index $0 < i < n$,



Remark

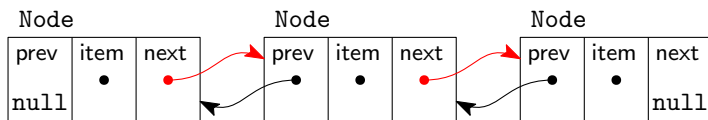
We treat sequences as 0-indexed: index 0 is the first element

Chains

Definition

A *chain* is a sequence of length $n > 0$ of nodes such that:

- ▶ the **prev** reference of the first node is **null**,
- ▶ the **next** reference of the last node is **null**,
- ▶ the **prev** of node i is node $i - 1$ for every index $0 < i < n$,
- ▶ the **next** of node i is node $i + 1$ for every index $0 \leq i < n - 1$.



Remark

We treat sequences as 0-indexed: index 0 is the first element

Doubly-linked list

Definition

A *doubly-linked list* structure consists of:

- ▶ a **first** reference to a node,
- ▶ a **last** reference to a node.

such that either:

- ▶ **first** and **last** are both **null**, or
- ▶ there is a chain between the **first** node and the **last** node.

Doubly-linked list

Definition

A *doubly-linked list* structure consists of:

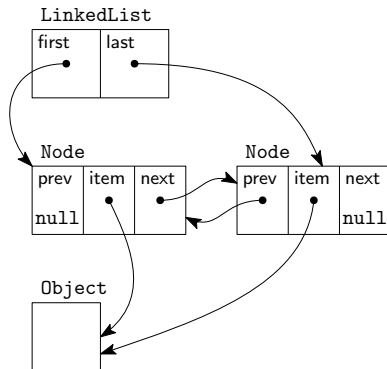
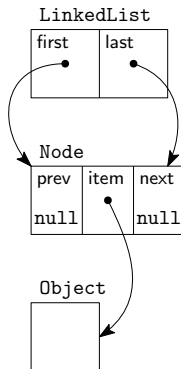
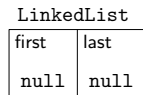
- ▶ a **first** reference to a node,
- ▶ a **last** reference to a node,

such that either:

- ▶ **first** and **last** are both **null**, or
- ▶ there is a chain between the **first** node and the **last** node.

What is a doubly-linked list?

Example



Chains

Lemma

Every chain is acyclic, that is, does not have the same node at two different indexes.

Chains

Lemma

Every chain is acyclic, that is, does not have the same node at two different indexes.

Proof by contradiction.

1. Node i and node j are the same for some $0 \leq i < j < n$.

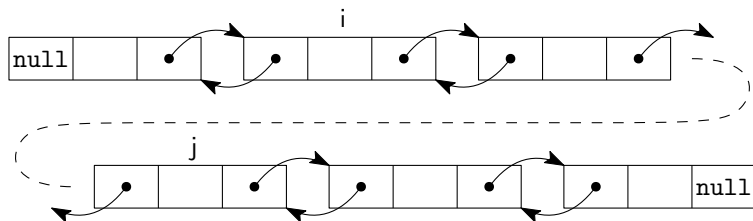
Chains

Lemma

Every chain is acyclic, that is, does not have the same node at two different indexes.

Proof by contradiction.

1. Node i and node j are the same for some $0 \leq i < j < n$.
2. Node $i + k$ and node $j + k$ are the same for all $0 \leq k < n - j$.



Chains

Lemma

Every chain is acyclic, that is, does not have the same node at two different indexes.

Proof by contradiction.

1. Node i and node j are the same for some $0 \leq i < j < n$.
2. Node $i + k$ and node $j + k$ are the same for all $0 \leq k < n - j$.
3. Node $k - (j - i)$ and node k are the same for all $j \leq k < n$.

Chains

Lemma

Every chain is acyclic, that is, does not have the same node at two different indexes.

Proof by contradiction.

1. Node i and node j are the same for some $0 \leq i < j < n$.
2. Node $i + k$ and node $j + k$ are the same for all $0 \leq k < n - j$.
3. Node $k - (j - i)$ and node k are the same for all $j \leq k < n$.
4. Node $(n - 1) - (j - i)$ has **next** node $n - (j - i)$.

Chains

Lemma

Every chain is acyclic, that is, does not have the same node at two different indexes.

Proof by contradiction.

1. Node i and node j are the same for some $0 \leq i < j < n$.
2. Node $i + k$ and node $j + k$ are the same for all $0 \leq k < n - j$.
3. Node $k - (j - i)$ and node k are the same for all $j \leq k < n$.
4. Node $(n - 1) - (j - i)$ has **next** node $n - (j - i)$.
5. Node $n - 1$ has **next** **null**.

Chains

Lemma

Every chain is acyclic, that is, does not have the same node at two different indexes.

Proof by contradiction.

1. Node i and node j are the same for some $0 \leq i < j < n$.
2. Node $i + k$ and node $j + k$ are the same for all $0 \leq k < n - j$.
3. Node $k - (j - i)$ and node k are the same for all $j \leq k < n$.
4. Node $(n - 1) - (j - i)$ has **next** node $n - (j - i)$.
5. Node $n - 1$ has **next** **null**.
6. Contradiction of 4. and 5. by 3. with $k = n - 1$.



Chains

Lemma

Every chain is acyclic, that is, does not have the same node at two different indexes.

Proof by contradiction.

1. Node i and node j are the same for some $0 \leq i < j < n$.
2. Node $i + k$ and node $j + k$ are the same for all $0 \leq k < n - j$.
3. Node $k - (j - i)$ and node k are the same for all $j \leq k < n$.
4. Node $(n - 1) - (j - i)$ has **next** node $n - (j - i)$.
5. Node $n - 1$ has **next** **null**.
6. Contradiction of 4. and 5. by 3. with $k = n - 1$.



There is a bug in Java's `LinkedList`

Class `java.util.LinkedList`

`/** Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null). ... */`

```
public class LinkedList ... {  
    public boolean contains(Object o) { ... }  
    public int size() { ... }  
    public void add(Object e) { ... }  
    ...  
}
```

How containers should contain

You would expect:

- ▶ Given any `LinkedList` instance `x`
- ▶ Directly after calling `x.add(y = new Object())`
- ▶ `x.contains(y)` returns **true**

How containers should contain

You would expect:

- ▶ Given any `LinkedList` instance `x`
- ▶ Directly after calling `x.add(y = new Object())`
- ▶ `x.contains(y)` returns **true**

In reality:

- ▶ `x = new LinkedList();`
- ▶ `x.add(null);`
- ▶ `:`
- ▶ `x.add(null);`
- ▶ `x.add(y = new Object());`
- ▶ `x.contains(y)` returns **false**

How large containers are

You would expect:

- ▶ Given any `LinkedList` instance `x`
- ▶ `x.size()` is never negative

How large containers are

You would expect:

- ▶ Given any `LinkedList` instance `x`
- ▶ `x.size()` is never negative

In reality:

- ▶ `x = new LinkedList();`
 $\Rightarrow x.size() == 0$
- ▶ `x.add(null);`
 $\Rightarrow x.size() == 1$
- ▶ $\vdots$
- ▶ `x.add(null);`
 $\Rightarrow x.size() == \text{Integer.MAX_VALUE} == 2^{31} - 1$
- ▶ `x.add(y = new Object());`
 $\Rightarrow x.size() == \text{Integer.MIN_VALUE} == -2^{31}$

Source code dive

```
public boolean contains(Object o) {  
    return indexOf(o) >= 0;  
}  
  
public int indexOf(Object o) {  
    int index = 0;  
    if (o == null) { ... }  
    else {  
        for (Node x=first; x != null; x=x.next) {  
            if (o.equals(x.item)) return index;  
            index++;  
        }  
    }  
    return -1;  
}
```

What is wrong?

Consider this example:

$$e_1, \dots, e_{2^{31}-1}, e_{2^{31}}, \dots, e_{2^{32}-1}, e_{2^{32}}, e_{2^{32}+1}, \dots$$

1. Initially: size is zero
2. First elements: size is positive
3. Add more elements: size is negative
4. Add an element: size is zero
5. Add elements: size is positive
6. ...

What is wrong?

Consider this example:

$$e_1, \dots, e_{2^{31}-1}, e_{2^{31}}, \dots, e_{2^{32}-1}, e_{2^{32}}, e_{2^{32}+1}, \dots$$

1. Initially: size is zero
2. First elements: size is positive
3. Add more elements: size is negative
4. Add an element: size is zero
5. Add elements: size is positive
6. ...

What is wrong?

Consider this example:

$$e_1, \dots, e_{2^{31}-1}, e_{2^{31}}, \dots, e_{2^{32}-1}, e_{2^{32}}, e_{2^{32}+1}, \dots$$

1. Initially: size is zero
2. First elements: size is positive
3. Add more elements: size is negative
4. Add an element: size is zero
5. Add elements: size is positive
6. ...

What is wrong?

Consider this example:

$$e_1, \dots, e_{2^{31}-1}, e_{2^{31}}, \dots, e_{2^{32}-1}, e_{2^{32}}, e_{2^{32}+1}, \dots$$

1. Initially: size is zero
2. First elements: size is positive
3. Add more elements: size is negative
4. Add an element: size is zero
5. Add elements: size is positive
6. ...

What is wrong?

Consider this example:

$$e_1, \dots, e_{2^{31}-1}, e_{2^{31}}, \dots, e_{2^{32}-1}, e_{2^{32}}, e_{2^{32}+1}, \dots$$

1. Initially: size is zero
2. First elements: size is positive
3. Add more elements: size is negative
4. Add an element: size is zero
5. Add elements: size is positive
6. ...

What is wrong?

Consider this example:

$$e_1, \dots, e_{2^{31}-1}, e_{2^{31}}, \dots, e_{2^{32}-1}, e_{2^{32}}, e_{2^{32}+1}, \dots$$

1. Initially: size is zero
2. First elements: size is positive
3. Add more elements: size is negative
4. Add an element: size is zero
5. Add elements: size is positive
6. ...

Implementation

<code>linkFirst(o)</code>	<code>linkLast(o)</code>	<code>linkBefore(o,n)</code>
<code>unlinkFirst(n)</code>	<code>unlinkLast(n)</code>	<code>unlink(n)</code>

<code>add(o)</code>	<code>poll()</code>	<code>element()</code>
<code>addLast(o)</code>	<code>pollFirst()</code>	<code>getFirst()</code>
<code>push(o)</code>	<code>pollLast()</code>	<code>getLast()</code>
<code>addFirst(o)</code>	<code>pop()</code>	<code>peek()</code>
<code>offer(o)</code>	<code>remove()</code>	<code>peekFirst()</code>
<code>offerLast(o)</code>	<code>removeFirst()</code>	<code>peekLast()</code>
<code>offerFirst(o)</code>	<code>removeLast()</code>	

<code>remove(o)</code>	<code>removeFirstOccurrence(o)</code>
	<code>removeLastOccurrence(o)</code>

Implementation

<code>linkFirst(o)</code>	<code>linkLast(o)</code>	<code>linkBefore(o,n)</code>
<code>unlinkFirst(n)</code>	<code>unlinkLast(n)</code>	<code>unlink(n)</code>

<code>add(o)</code>	<code>poll()</code>	<code>element()</code>
<code>addLast(o)</code>	<code>pollFirst()</code>	<code>getFirst()</code>
<code>push(o)</code>	<code>pollLast()</code>	<code>getLast()</code>
<code>addFirst(o)</code>	<code>pop()</code>	<code>peek()</code>
<code>offer(o)</code>	<code>remove()</code>	<code>peekFirst()</code>
<code>offerLast(o)</code>	<code>removeFirst()</code>	<code>peekLast()</code>
<code>offerFirst(o)</code>	<code>removeLast()</code>	

<code>remove(o)</code>	<code>removeFirstOccurrence(o)</code>
	<code>removeLastOccurrence(o)</code>

Source code dive

```
public void offer(Object e) { return add(e); }
```

Source code dive

```
public void offer(Object e) { return add(e); }  
public void add(Object e) {  
    linkLast(e);  
    return true;  
}
```

Source code dive

```
public void offer(Object e) { return add(e); }

public void add(Object e) {
    linkLast(e);
    return true;
}

private void linkLast(Object e) {
    Node l = last;
    Node newNode = new Node(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

Implementation

<code>linkFirst(o)</code>	<code>linkLast(o)</code>	<code>linkBefore(o,n)</code>
<code>unlinkFirst(n)</code>	<code>unlinkLast(n)</code>	<code>unlink(n)</code>

<code>node(i)</code>	<code>checkElementIndex(i)</code>	<code>checkPositionIndex(i)</code>
----------------------	-----------------------------------	------------------------------------

<code>add(i,o)</code>	<code>set(i,o)</code>	<code>get(i)</code>
<code>addAll(c)</code>	<code>isEmpty()</code>	<code>size()</code>
<code>addAll(i,c)</code>	<code>indexOf(o)</code>	<code>toArray()</code>
<code>remove(i)</code>	<code>lastIndexOf(o)</code>	<code>iterator()</code>
<code>removeAll(c)</code>	<code>contains(o)</code>	<code>listIterator()</code>
<code>retainAll(c)</code>	<code>containsAll(c)</code>	<code>listIterator(i)</code>

Implementation

<code>linkFirst(o)</code>	<code>linkLast(o)</code>	<code>linkBefore(o,n)</code>
<code>unlinkFirst(n)</code>	<code>unlinkLast(n)</code>	<code>unlink(n)</code>

<code>node(i)</code>	<code>checkElementIndex(i)</code>	<code>checkPositionIndex(i)</code>
----------------------	-----------------------------------	------------------------------------

<code>add(i,o)</code>	<code>set(i,o)</code>	<code>get(i)</code>
<code>addAll(c)</code>	<code>isEmpty()</code>	<code>size()</code>
<code>addAll(i,c)</code>	<code>indexOf(o)</code>	<code>toArray()</code>
<code>remove(i)</code>	<code>lastIndexOf(o)</code>	<code>iterator()</code>
<code>removeAll(c)</code>	<code>contains(o)</code>	<code>listIterator()</code>
<code>retainAll(c)</code>	<code>containsAll(c)</code>	<code>listIterator(i)</code>

Source code dive

```
public Object get(int index) {  
    checkElementIndex(index);  
    return node(index).item;  
}
```

Source code dive

```
public Object get(int index) {  
    checkElementIndex(index);  
    return node(index).item;  
}  
  
private void checkElementIndex(int index) {  
    if (!(index >= 0 && index < size))  
        throw new IndexOutOfBoundsException(...);  
}
```

Source code dive

```
public Object get(int index) {  
    checkElementIndex(index);  
    return node(index).item;  
}
```

```
Node node(int index) {  
    if (index < (size >> 1)) {  
        Node x = first;  
        for (int i = 0; i < index; i++)  
            x = x.next;  
        return x;  
    } else {  
        Node x = last;  
        for (int i = size - 1; i > index; i--)  
            x = x.prev;  
        return x;  
    }  
}
```

What is wrong?

Consider this example again:

$$e_1, \dots, e_{2^{31}-1}, e_{2^{31}}, \dots, e_{2^{32}-1}, e_{2^{32}}, e_{2^{32}+1}, e_{2^{32}+2}$$

1. Initially: size is zero
2. First elements: size is positive
3. Add more elements: size is negative
4. Add an element: size is zero
5. Add two elements: size is 2
6. Now `get (0)` returns e_1 and `get (1)` returns $e_{2^{32}+2}$

What is wrong?

Consider this example again:

$$e_1, \dots, e_{2^{31}-1}, e_{2^{31}}, \dots, e_{2^{32}-1}, e_{2^{32}}, e_{2^{32}+1}, e_{2^{32}+2}$$

1. Initially: size is zero
2. First elements: size is positive
3. Add more elements: size is negative
4. Add an element: size is zero
5. Add two elements: size is 2
6. Now `get (0)` returns e_1 and `get (1)` returns $e_{2^{32}+2}$

What is wrong?

After a particular size:

- ▶ incorrect `size()`
 - ▶ thus incorrect `isEmpty()`
 - ▶ thus `subList(i, i)` **broken**

What is wrong?

After a particular size:

- ▶ incorrect `size()`
 - ▶ thus incorrect `isEmpty()`
 - ▶ thus `subList(i, i)` **broken**
- ▶ incorrect `indexOf(o)`
 - ▶ thus incorrect `contains(o)`
 - ▶ thus incorrect `containsAll(c)`

What is wrong?

After a particular size:

- ▶ incorrect `size()`
 - ▶ thus incorrect `isEmpty()`
 - ▶ thus `subList(i, i)` **broken**
- ▶ incorrect `indexOf(o)`
 - ▶ thus incorrect `contains(o)`
 - ▶ thus incorrect `containsAll(c)`
- ▶ positional access `node(i)` **broken**:
 - ▶ thus also `get(i)`, `set(i, o)`, `add(i, o)`, `remove(i)`

What is wrong?

After a particular size:

- ▶ `incorrect size()`
 - ▶ `thus incorrect isEmpty()`
 - ▶ `thus subList(i, i) broken`
- ▶ `incorrect indexOf(o)`
 - ▶ `thus incorrect contains(o)`
 - ▶ `thus incorrect containsAll(c)`
- ▶ `positional access node(i) broken:`
 - ▶ `thus also get(i), set(i, o), add(i, o), remove(i)`
- ▶ `iterator() and listIterator() are broken`
 - ▶ `thus equals(o) and hashCode() are broken`
- ▶ ...

What is *not* wrong?

- ▶ new elements can still be added `add(o)`
- ▶ elements can still be removed: `remove(o)`

What is *not* wrong?

- ▶ new elements can still be added `add(o)`
- ▶ elements can still be removed: `remove(o)`
- ▶ `clear()` and `clone()` still work

What is *not* wrong?

- ▶ new elements can still be added `add(o)`
- ▶ elements can still be removed: `remove(o)`
- ▶ `clear()` and `clone()` still work
- ▶ double ended queue operations still work
 - ▶ first and last element are always accessible
`getFirst()`, `getLast()`
 - ▶ first and last element can be removed
`removeFirst()`, `removeLast()`
- ▶ ...

What is *not* wrong?

- ▶ new elements can still be added `add(o)`
- ▶ elements can still be removed: `remove(o)`
- ▶ `clear()` and `clone()` still work
- ▶ double ended queue operations still work
 - ▶ first and last element are always accessible
`getFirst()`, `getLast()`
 - ▶ first and last element can be removed
`removeFirst()`, `removeLast()`
- ▶ ...

Thus, data can still be recovered:

- ▶ clone the original linked list
- ▶ destructively read from the clone

Requires twice the amount of memory!

A **mismatch** between the
cached size and the **actual** size
of the linked list.

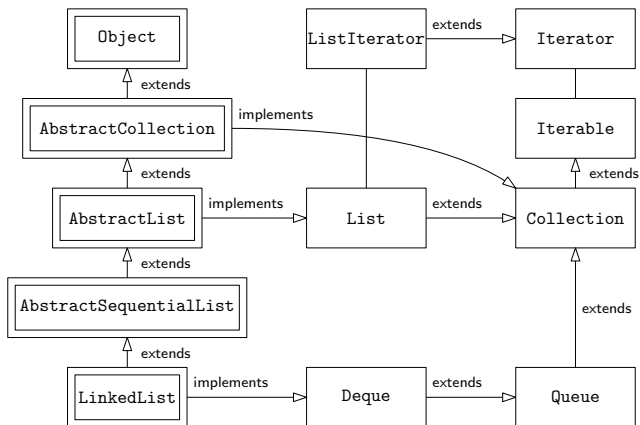
Modular specification of Java's Collection Framework

Related work

*Based on our experience, informal specifications are often to a great extent **incomplete** and **erroneous**. Defects even remain **incognito** for years or decades, which consequently means that a direct translation from an informal specification is often impractical and has to be taken with **caution**.*

Alexander Knüppel, Thomas Thüm, Carsten Pardylla, and Ina Schaefer.
[Experience Report on Formally Verifying Parts of OpenJDK's API with KeY.](#)
In *4th Workshop on Formal Integrated Development Environment*, July 2018.

Class Hierarchy



List

Every element in a list has a definite **index**:

- ▶ `indexOf(o)` returns a primitive **int**

such that:

- ▶ if contains(o) then `get(indexOf(o)) == o`

List

Every element in a list has a definite **index**:

- ▶ `indexOf(o)` returns a primitive **int**

such that:

- ▶ if `contains(o)` then `get(indexOf(o)) == o`

Every new element in the list has a definite **position**:

- ▶ `add(i, o)` takes a primitive **int**

such that:

- ▶ `add(size(), o)` adds to the end

thus:

- ▶ `size()` returns a position

List

`size()` returns *the number of elements in this collection*.

However (cf. Javadoc):

If this collection contains more than `MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

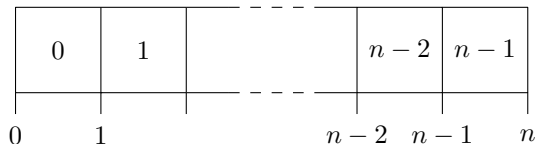
A size of `MAX_VALUE` indicates an unknown size.

A size indicates a maximum position.

Thus `MAX_VALUE` indicates an unknown position.

ListIterator

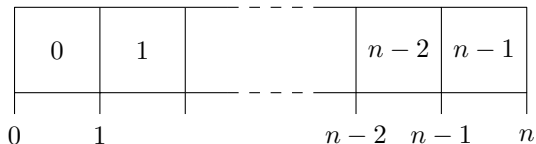
A list iterator is a specialized `Iterator` for lists.



A list of size n has $n+1$ definite **cursor positions**.

ListIterator

A list iterator is a specialized `Iterator` for lists.



A list of size n has $n+1$ definite **cursor positions**.

Suppose `MAX_VALUE` were the maximum size of a list.

Then the position *after* the last element **overflows**.

Suppose `MAX_VALUE - 1` were the maximum size of a list.

Then `MAX_VALUE` would be the position *after* the last element.

But then `MAX_VALUE` indicates an unknown position.

ListIterator

A list iterator is a specialized `Iterator` for lists.



A list of size n has $n+1$ definite **cursor positions**.

Suppose `MAX_VALUE` were the maximum size of a list.
Then the position *after* the last element **overflows**.

Suppose `MAX_VALUE - 1` were the maximum size of a list.
Then `MAX_VALUE` would be the position *after* the last element.
But then `MAX_VALUE` indicates an unknown position.

The maximum list size is `MAX_VALUE - 2`.

Queue

An implementation may allow **unbounded** queues:

- ▶ `add(o)` can be called arbitrary many times
- ▶ `remove()` returns all elements in some order

However (cf. Javadoc):

*`toArray()` returns an array containing **all** of the elements in this collection.*

If size is `MAX_VALUE` typically results in `OutOfMemoryError`

In a **capacity-restricted** queue:

- ▶ `add(o)` may throw `IllegalStateException`

A class that implements both `Queue` and `List` is necessarily capacity-restricted

Main problem

A `List` has a maximum size.

A `Queue` may be unbounded.

A `Collection` could be infinite.

Main problem

A `List` has a maximum size.

A `Queue` may be unbounded.

A `Collection` could be infinite.

`UnboundedLinkedDeque`

Main problem

A `List` has a maximum size.

A `Queue` may be unbounded.

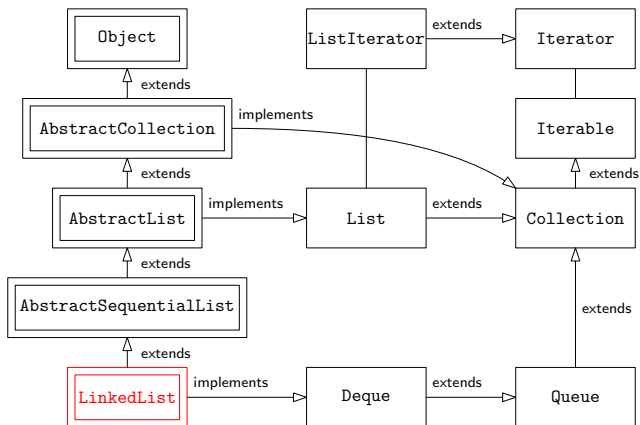
A `Collection` could be infinite.

`UnboundedLinkedListDeque`

(Bounded)`LinkedList`

On-going work on verification in KeY

Class Hierarchy



LinkedList implementation

Approach:

1. Employ **ghost variable** to store sequence of nodes
2. Encode linked list structure by invariant
3. Specify each method using the ghost variable

LinkedList implementation

Approach:

1. Employ **ghost variable** to store sequence of nodes
2. Encode linked list structure by invariant
3. Specify each method using the ghost variable

Pros:

- ▶ Allows verification of concrete implementation

Contras:

- ▶ Exposes implementation details
- ▶ Not related to supertypes

LinkedList implementation

```
public class LinkedList extends ... {  
    public ghost \seq nodeList;
```

LinkedList implementation

```
public class LinkedList extends ... {  
    public ghost \seq nodeList;  
    public invariant nodeList.length = size;  
}
```

LinkedList implementation

```
public class LinkedList extends ... {  
    public ghost \seq nodeList;  
    public invariant nodeList.length = size;  
    public invariant size  $\leq$  Integer.MAX_VALUE - 2;  
}
```

LinkedList implementation

```
public class LinkedList extends ... {  
    public ghost \seq nodeList;  
    public invariant nodeList.length = size;  
    public invariant size  $\leq$  Integer.MAX_VALUE - 2;  
    public invariant  $\forall 0 \leq i^{\text{bigint}} < \text{nodeList.length} :$   
        nodeList[i] instanceof Node;
```

LinkedList implementation

```
public class LinkedList extends ... {  
    public ghost \seq nodeList;  
    public invariant nodeList.length = size;  
    public invariant size  $\leq$  Integer.MAX_VALUE - 2;  
    public invariant  $\forall 0 \leq i^{\text{bigint}} < \text{nodeList.length} :$   
        nodeList[i] instanceof Node;  
    public invariant (nodeList =  $\varepsilon$   $\wedge$  first = null  $\wedge$  last = null)  $\vee$   
        (nodeList  $\neq \varepsilon$   $\wedge$  first = nodeList[0]  $\wedge$  first.prev = null  $\wedge$   
            last.next = null  $\wedge$  last = nodeList[nodeList.length - 1]);  
}
```

LinkedList implementation

```
public class LinkedList extends ... {  
    public ghost \seq nodeList;  
  
    public invariant nodeList.length = size;  
  
    public invariant size  $\leq$  Integer.MAX_VALUE - 2;  
  
    public invariant  $\forall 0 \leq i^{\text{bigint}} < \text{nodeList.length} :$   
        nodeList[i] instanceof Node;  
  
    public invariant (nodeList =  $\varepsilon$   $\wedge$  first = null  $\wedge$  last = null)  $\vee$   
        (nodeList  $\neq \varepsilon$   $\wedge$  first = nodeList[0]  $\wedge$  first.prev = null  $\wedge$   
            last.next = null  $\wedge$  last = nodeList[nodeList.length - 1]);  
  
    public invariant  $\forall 0 < i^{\text{bigint}} < \text{nodeList.length} :$   
        ((Node) nodeList[i]).prev = (Node) nodeList[i-1];  
    public invariant  $\forall 0 \leq i^{\text{bigint}} < \text{nodeList.length} - 1 :$   
        ((Node) nodeList[i]).next = (Node) nodeList[i+1];
```

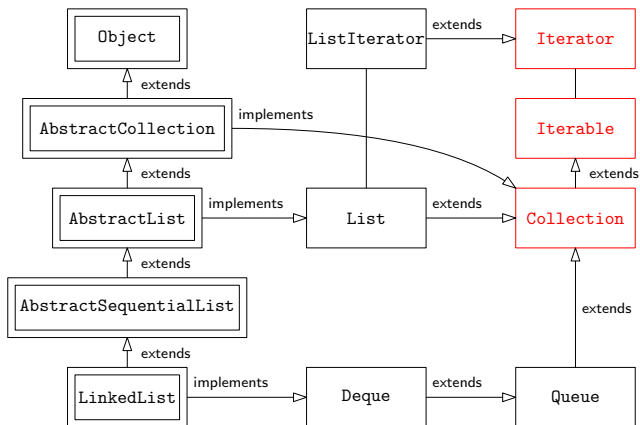
LinkedList implementation

impure method	number of rules
linkFirst(o)	8,136
addFirst(o)	8,170
linkLast(o)	17,865
addLast(o)	17,596
add(o)	18,997
offer(o)	16,986
offerFirst(o)	8,235
offerLast(o)	18,616
push(o)	8,387

pure method	number of rules
peekFirst()	1,382
peekLast()	1,519
getFirst()	916
getLast()	1,023
peek()	1,383
element()	938
<ctor>()	721
size()	1,102
isElementIndex()	1,941
isPositionIndex()	1,963



Class Hierarchy



Collection framework

Approach:

1. Employ **model methods** to abstract implementation
2. Apply instance invariants to constrain model methods
3. Specify each method using model methods

Collection framework

Approach:

1. Employ **model methods** to abstract implementation
2. Apply instance invariants to constrain model methods
3. Specify each method using model methods

Pros:

- ▶ Allows abstract usage of interfaces

Contras:

- ▶ Difficult to come up with right definition
- ▶ Managing and working with theorems

Iterator

Definition

An *iterator* structure consists of:

- ▶ a map $\text{dom} : \mathbb{N} \rightarrow \text{boolean}$,
- ▶ a map $\text{elem} : \mathbb{N} \rightarrow \text{Object}$,

such that:

$$\forall n^{\mathbb{N}} : \text{dom}(n) = \text{false} \implies \text{dom}(n+1) = \text{false} \wedge \text{elem}(n) = \text{null}.$$

Iterator

Definition

An *iterator* structure consists of:

- ▶ a map $\text{dom} : \mathbb{N} \rightarrow \text{boolean}$,
- ▶ a map $\text{elem} : \mathbb{N} \rightarrow \text{Object}$,

such that:

$$\forall n^{\mathbb{N}} : \text{dom}(n) = \text{false} \implies \text{dom}(n+1) = \text{false} \wedge \text{elem}(n) = \text{null}.$$

Intuitively, think of either finite or countably infinite sequences.

Iterator

An iterator is either **empty** or **non-empty**, that is,

$$(\forall n^{\mathbb{N}} : \text{dom}(n) = \text{false}) \vee (\exists n^{\mathbb{N}} : \text{dom}(n) = \text{true})$$

Iterator

An iterator is either **empty** or **non-empty**, that is,

$$(\forall n^{\mathbb{N}} : \text{dom}(n) = \text{false}) \vee (\exists n^{\mathbb{N}} : \text{dom}(n) = \text{true})$$

An iterator is either **finite** or **infinite**, that is,

$$(\exists n^{\mathbb{N}} : \text{dom}(n) = \text{false}) \vee (\forall n^{\mathbb{N}} : \text{dom}(n) = \text{true})$$

Iterator

An iterator is either **empty** or **non-empty**, that is,

$$(\forall n^{\mathbb{N}} : \text{dom}(n) = \text{false}) \vee (\exists n^{\mathbb{N}} : \text{dom}(n) = \text{true})$$

An iterator is either **finite** or **infinite**, that is,

$$(\exists n^{\mathbb{N}} : \text{dom}(n) = \text{false}) \vee (\forall n^{\mathbb{N}} : \text{dom}(n) = \text{true})$$

Every finite iterator has a **length** ℓ , that is,

$$(\exists n^{\mathbb{N}} : \text{dom}(n) = \text{false}) \implies (\exists \ell^{\mathbb{N}} \forall m^{\mathbb{N}} : \text{dom}(m) = (m < \ell))$$

Iterator

An iterator is either **empty** or **non-empty**, that is,

$$(\forall n^{\mathbb{N}} : \text{dom}(n) = \text{false}) \vee (\exists n^{\mathbb{N}} : \text{dom}(n) = \text{true})$$

An iterator is either **finite** or **infinite**, that is,

$$(\exists n^{\mathbb{N}} : \text{dom}(n) = \text{false}) \vee (\forall n^{\mathbb{N}} : \text{dom}(n) = \text{true})$$

Every finite iterator has a **length** ℓ , that is,

$$(\exists n^{\mathbb{N}} : \text{dom}(n) = \text{false}) \implies (\exists \ell^{\mathbb{N}} \forall m^{\mathbb{N}} : \text{dom}(m) = (m < \ell))$$

Lemma

Every empty iterator is finite and has length 0.

Every non-empty finite iterator has length > 0 .



Iterator

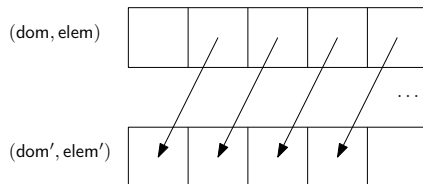
Interface `java.util.Iterator`:

- ▶ pure **boolean** `hasNext()` returns **false** iff **empty**

Iterator

Interface `java.util.Iterator`:

- ▶ pure **boolean** `hasNext()` returns **false** iff **empty**
- ▶ impure `Object next()` requires a **non-empty** iterator, returns **`elem(0)`** and relates the old and new iterator:



$$\forall n^{\mathbb{N}} : \text{dom}'(n) = \text{dom}(n+1) \wedge \text{elem}'(n) = \text{elem}(n+1)$$

Iterator

Interface `java.util.Iterator`:

- ▶ pure **boolean** `hasNext()` returns **false** iff **empty**
- ▶ impure `Object next()` requires a **non-empty** iterator, returns **elem(0)** and relates the old and new iterator.

Lemma

Every finite iterator remains finite under `next()`.

A finite iterator's length decreases by one after `next()`.



Iterator

Interface `java.util.Iterator`:

- ▶ pure **boolean** `hasNext()` returns **false** iff **empty**
- ▶ impure `Object next()` requires a **non-empty** iterator, returns **`elem(0)`** and relates the old and new iterator.

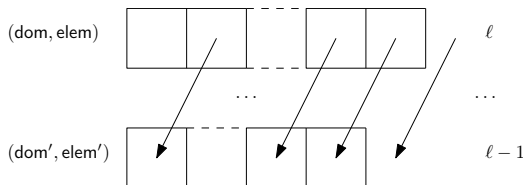
Lemma

Every finite iterator remains finite under `next()`.

A finite iterator's length decreases by one after `next()`.



$$\ell' = \ell - 1$$



Iterator

Interface `java.util.Iterator`:

- ▶ pure **boolean** `hasNext()` returns **false** iff **empty**
- ▶ impure `Object next()` requires a **non-empty** iterator, returns **`elem(0)`** and relates the old and new iterator.

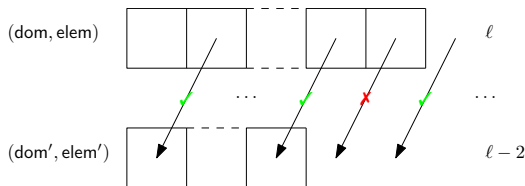
Lemma

Every finite iterator remains finite under `next()`.

A finite iterator's length decreases by one after `next()`.



$$\ell' < \ell - 1$$



Iterator

Interface `java.util.Iterator`:

- ▶ pure **boolean** `hasNext()` returns **false** iff **empty**
- ▶ impure `Object next()` requires a **non-empty** iterator, returns **elem(0)** and relates the old and new iterator.

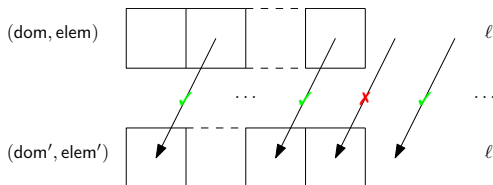
Lemma

Every finite iterator remains finite under `next()`.

A finite iterator's length decreases by one after `next()`.



$$\ell' > \ell - 1$$



Iterator

Interface `java.util.Iterator`:

- ▶ pure **boolean** `hasNext()` returns **false** iff **empty**
- ▶ impure `Object next()` requires a **non-empty** iterator, returns **elem(0)** and relates the old and new iterator.

public behavior

requires `\invariant_for(x) & x.isFinite()`;

ensures `x.isEmpty()` & `\result = true`;

```
public static boolean dec(Iterator x) {  
    maintaining \invariant_for(x) & x.isFinite();  
    decreasing x.length();  
    while (x.hasNext())  
        x.next();  
    return true;  
}
```

Iterator

Interface `java.util.Iterator`:

- ▶ pure **boolean** `hasNext()` returns **false** iff **empty**
- ▶ impure `Object next()` requires a **non-empty** iterator, returns **elem(0)** and relates the old and new iterator.

public behavior

requires `\invariant_for(x)`;

ensures `\result == \old(x).isFinite()`;

```
public static boolean dec(Iterator x) {  
    ?  
}
```


Collection

Interface `java.util.Collection`:

- ▶ `add(o)`, `addAll(c)`
- ▶ `remove(o)`, `removeAll(c)`, `retainAll(c)`, `clear()`
- ▶ `size()`, `isEmpty()`, `contains(o)`
- ▶ `toArray()`, `iterator()`

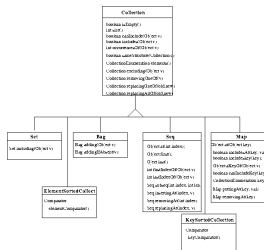
- ▶ A collection is **not** a multiset:
 - ▶ elements may be (de)duplicated and reordered freely
- ▶ A collection is **not** a set:
 - ▶ elements can occur more than once in an iteration

On-going work:

- ▶ Show how old and new iterator are related.
- ▶ Using **freeze variables** for element removal.

The bug is 20 years old

1995 Release of Doug Lea's Collection Framework



"Collections are objects that hold other objects that are accessed, placed, and maintained under some set of rules. [...] This package uses one of the simplest, splitting them into four main categories: sets, bags, seqs, maps [...]. Top-level functionality is described entirely via interfaces, not classes."

Time line

1995 Release of Doug Lea's [Collection Framework](#)

- ▶ Seems to contain the *archetype* of the bug

```
public abstract class UpdatableImpl ... {  
    protected int version_; ⇒ became modCount  
    protected int count_;   ⇒ became size  
    public syn.  int size() { return count_; }  
    ...  
}
```

1998 Java 1.2 introduces Collections Framework

- ▶ Influenced by Doug Lea's collections
- ▶ Its main contributor is Joshua Bloch
- ▶ Includes `java.util.LinkedList`
- ▶ Binary class distribution: source was proprietary

"This new unified architecture represents and manipulates collections for development efficiency and allows interoperability among unrelated APIs."

2002 Java 1.4 supports 64-bit architectures

- ▶ Address space goes from 32-bit to 64-bit.
- ▶ Thus, more than 2^{32} objects can be created!

"With compelling performance results and key new features such as new I/O and 64-bit support, J2SE is the ideal platform for large-scale data mining, business intelligence, engineering and scientific applications."

- ▶ 1.4.0 targets Solaris on SPARC v9
- ▶ 1.4.1 targets Windows and Linux on Intel Itanium
- ▶ 1.4.2_22 targets Solaris on x86-AMD64

2007 Open source release of OpenJDK

- ▶ Source for `java.util.*` became open

*From: rich.green at sun.com
Date: Tue, 08 May 2007 09:29:14 -0700
Subject: Open JDK is here!*

*Today begins the next phase for Java.
I am pleased to announce that we have
completed the open source release of
Open JDK.*

Reverse Engineering

- ▶ Comparing JDK 1.2.2, 1.4.1 and OpenJDK 1.6
 - ▶ Using Java Decompiler by Emmanuel Dupuy and others
- ▶ Decompiled and source files are very similar.
 - ▶ No changes between 1.2.2 and 1.4.1
 - ▶ Field names match
 - ▶ Line number distances mostly match
- ▶ Some implementation changes between 1.4.1 and 1.6:
 - ▶ Since 1.5, `Queue` is implemented
 - ▶ Since 1.6, `Deque` is implemented
 - ▶ 1.4.1–1.6, input collections are converted `toArray`
 - ▶ 1.4.1–1.6, `clear` helps GC by `nulling`
 - ▶ 1.4.1–1.6, change signature of private `remove`

Reverse Engineering

- ▶ Comparing JDK 1.2.2, 1.4.1 and OpenJDK 1.6
 - ▶ Using Java Decompiler by Emmanuel Dupuy and others
- ▶ Decompiled and source files are very similar.
 - ▶ No changes between 1.2.2 and 1.4.1
 - ▶ Field names match
 - ▶ Line number distances mostly match
- ▶ Some implementation changes between 1.4.1 and 1.6:
 - ▶ Since 1.5, `Queue` is implemented
 - ▶ Since 1.6, `Deque` is implemented
 - ▶ 1.4.1–1.6, input collections are converted `toArray`
 - ▶ 1.4.1–1.6, `clear` helps GC by `nulling`
 - ▶ 1.4.1–1.6, change signature of private `remove`

Conclusion: **bug present since Java 1.2, activated in 1.4**

(so more than 20 years old)

Version History

- 2007 Initial version
- 2009 `LinkedList` performance improvements
- 2011 Resync `java.util.concurrent` classes with Doug Lea's CVS
- 2011 Update usages of `InternalError` to use exception chaining
- 2013 `Splititerator` implementations
- 2013 `Iterator.remove` and `forEachRemaining` relationship not specified
- 2014 Remove redundant null initialization
- 2014 Pico-optimize `contains` method
- 2015 `LinkedList` has incorrect implementation comment

Version History

2009 LinkedList performance improvements

Reimplementation of LinkedList. Bug remains present.

2014 Pico-optimize contains method

```
public boolean contains(Object o) {  
-   return indexOf(o) != -1;  
+   return indexOf(o) >= 0;  
}
```

Conclusion

Back-up slides

Abstraction map

- ▶ An *abstraction map* hides structure but keeps elements
- ▶ Each LinkedList maps to the *sequence* of its elements

Abstraction map

- ▶ An *abstraction map* hides structure but keeps elements
- ▶ Each LinkedList maps to the *sequence* of its elements

Abstraction map for LinkedList:

- ▶ Empty $\rightarrow$ empty sequence
- ▶ Otherwise $\rightarrow$ sequence of items in the chain

Abstraction map

- ▶ An *abstraction map* hides structure but keeps elements
- ▶ Each LinkedList maps to the *sequence* of its elements

Abstraction map for LinkedList:

- ▶ Empty $\rightarrow$ empty sequence
- ▶ Otherwise $\rightarrow$ sequence of items in the chain

Remark

Although nodes are unique, items need not!

Abstraction map

- ▶ An *abstraction map* hides structure but keeps elements
- ▶ Each LinkedList maps to the *sequence* of its elements

Abstraction map for LinkedList:

- ▶ Empty $\rightarrow$ empty sequence
- ▶ Otherwise $\rightarrow$ sequence of items in the chain

Remark

Although nodes are unique, items need not!

Remark

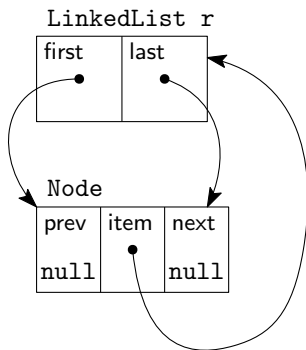
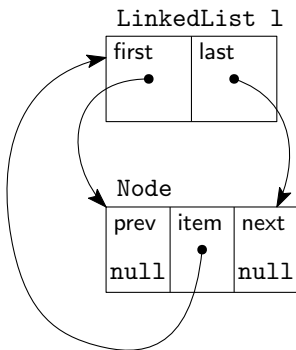
Ideally, for every sequence of elements, there is a LinkedList that maps to that sequence.

What is equality?

```
LinkedList l = new LinkedList();  
LinkedList r = new LinkedList();  
l.add(l);  
r.add(r);  
assert l.equals(r);
```

What is equality?

```
LinkedList l = new LinkedList();  
LinkedList r = new LinkedList();  
l.add(1);  
r.add(r);  
assert l.equals(r);
```



What is equality?

```
LinkedList l = new LinkedList();  
LinkedList r = new LinkedList();  
l.add(l);  
l.add(r);  
r.add(l);  
r.add(r);  
assert l.equals(r);
```

What is equality?

```
LinkedList l = new LinkedList();  
LinkedList r = new LinkedList();  
l.add(l);  
l.add(r);  
r.add(l);  
r.add(r);  
assert l.equals(r);
```

