

Verification of OpenJDK's LinkedList using KeY

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



Centrum Wiskunde & Informatica

Summary:

There is a bug in `LinkedList`. It can be fixed.

This is formally shown to be correct:
the bug no longer occurs.

Outline

Summary:

There is a bug in `LinkedList`. It can be fixed.
This is formally shown to be correct:
the bug no longer occurs.

Outline:

1. What is a doubly-linked list?
2. There is a bug in Java's `LinkedList`
3. Specification & Verification
4. **KeY (demo)**
5. *Extra: the bug is 20 years old*

Assumptions

In the following:

- ▶ Some realistic subset of Java 8 (OpenJDK)
- ▶ Without generics
⇒ treat as with type erasure
- ▶ Forget about multi-threading
⇒ treat as there is only one thread

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:

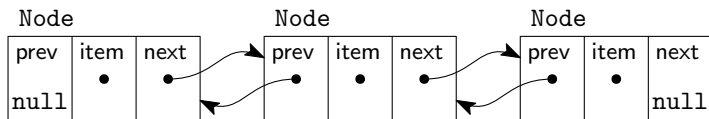
- ▶ a **prev** reference to a node,
- ▶ an **item** reference to an element,
- ▶ a **next** reference to a node.

References may be **null**.

Chains

Definition

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



Remark

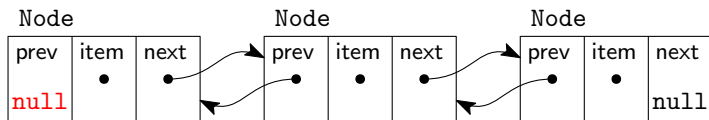
We treat sequences as 0-indexed: index 0 is the first element

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

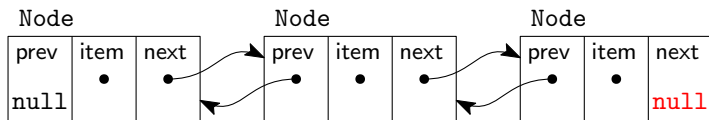
We treat sequences as 0-indexed: index 0 is the first element

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

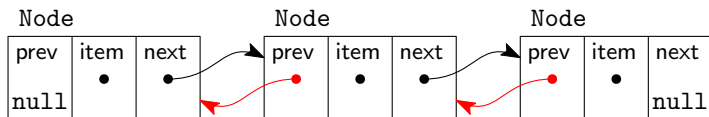
We treat sequences as 0-indexed: index 0 is the first element

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

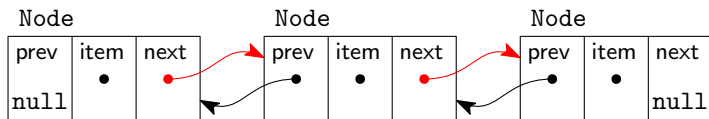
We treat sequences as 0-indexed: index 0 is the first element

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:

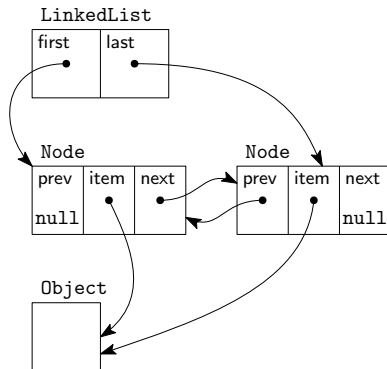
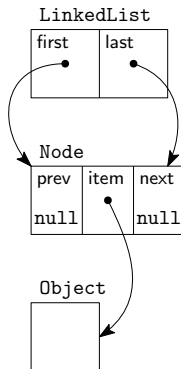
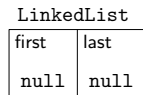
- ▶ **first** and **last** are both **null**, or
- ▶ there is a chain between the **first** node and the **last** node.

Remark

Standard literature (e.g. Knuth) uses a *list head*. We do not.

What is a doubly-linked list?

Example



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 indexOf(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(y = new Object())`
- ▶ `x.contains(y)` returns **false**

How lists work

You would expect:

- ▶ Given **any** `LinkedList` instance `x`
- ▶ Given **any** `Object` instance `y`
- ▶ Given that **int** `i = x.indexOf(y)`
- ▶ Either `i = -1` or `x.get(i)` returns `y`

How lists work

You would expect:

- ▶ Given **any** `LinkedList` instance `x`
- ▶ Given **any** `Object` instance `y`
- ▶ Given that **int** `i = x.indexOf(y)`
- ▶ Either `i = -1` or `x.get(i)` returns `y`

In reality:

- ▶ `x = new LinkedList()`
- ▶ `⋮`
- ▶ **int** `i = x.indexOf(y)`
- ▶ **assert** `i != -1` succeeds
- ▶ `x.get(i)` throws `IndexOutOfBoundsException` exception

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?

After a particular size:

- ▶ incorrect `indexOf(o)`
 - ▶ thus incorrect `contains(o)`
 - ▶ thus incorrect `containsAll(c)`

What is wrong?

After a particular size:

- ▶ incorrect `indexOf(o)`
 - ▶ thus incorrect `contains(o)`
 - ▶ thus incorrect `containsAll(c)`
- ▶ ...
- ▶ at **least 20 methods** are broken

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

Specification & Verification



Use the **KeY** System

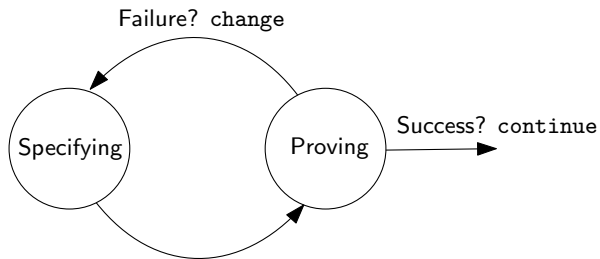
Goals:

- ▶ Repair the `LinkedList` implementation
- ▶ **Prove** that structural integrity is maintained
- ▶ **Prove** that integer overflow no longer occurs

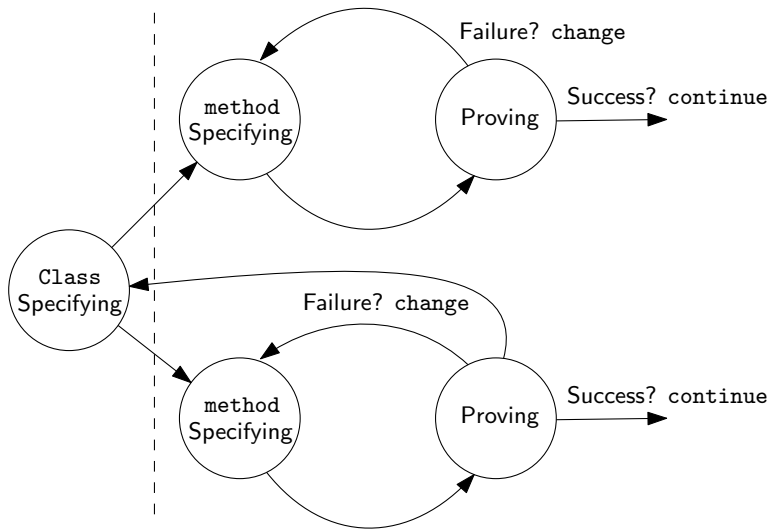
Non-goals:

- ▶ Specification of Java Collections framework
- ▶ Prove properties of clients of `LinkedList`

Approach



Approach



Challenges

Challenges:

- ▶ Find right specification
 1. Class invariants
 2. Method contracts
 3. Loop invariants

Challenges

Challenges:

- ▶ Find right specification
 1. Class invariants
 2. Method contracts
 3. Loop invariants
- ▶ Find right modelling technique
 - ▶ Model fields and model methods
 - ▶ Ghost fields, ghost parameters

Challenges

Challenges:

- ▶ Find right specification
 1. Class invariants
 2. Method contracts
 3. Loop invariants
- ▶ Find right modelling technique
 - ▶ Model fields and model methods
 - ▶ Ghost fields, ghost parameters
- ▶ Tackle technical issues
 - ▶ Reference aliasing and separation
 - ▶ Equality of `Object`

Challenges

Challenges:

- ▶ Find right specification
 1. Class invariants
 2. Method contracts
 3. Loop invariants
- ▶ Find right modelling technique
 - ▶ Model fields and model methods
 - ▶ Ghost fields, ghost parameters
- ▶ Tackle technical issues
 - ▶ Reference aliasing and separation
 - ▶ Equality of `Object`
- ▶ Understanding proofs on intuitive level

KeY (demo)

LinkedList.indexOf

We specify `LinkedList.indexOf(Object)`.

1. Original code (Java)

```
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;  
}
```

LinkedList.indexOf

We specify `LinkedList.indexOf(Object)`.

2. Class invariant (JML)

```
/*@ invariant
@   nodeList.length == size &&
@   nodeList.length <= Integer.MAX_VALUE &&
@   (\forallall \bigint i; 0 <= i < nodeList.length;
@       nodeList[i] instanceof Node) &&
@   ((nodeList==\seq_empty && first==null && last==null)
@   || (nodeList!=\seq_empty && first != null &&
@       first.prev == null && last != null &&
@       last.next==null && first == (Node)nodeList[0] &&
@       last == (Node)nodeList[nodeList.length-1])) &&
@   (\forallall \bigint i; 0 < i < nodeList.length;
@       ((Node)nodeList[i]).prev == (Node)nodeList[i-1]) &&
@   (\forallall \bigint i; 0 <= i < nodeList.length-1;
@       ((Node)nodeList[i]).next == (Node)nodeList[i+1]);
@*/
```

LinkedList.indexOf

We specify `LinkedList.indexOf(Object)`.

3. Method contract (JML)

```
/*@ ...
  @ public normal_behavior
  @   requires
  @     o != null;
  @   ensures
  @     \result >= -1 && \result < nodeList.length;
  @   ensures
  @     \result == -1 ==>
  @       (\forallall \bigint i; 0 <= i < nodeList.length;
  @         !o.equals(((Node)nodeList[i]).item));
  @   ensures
  @     \result >= 0 && \result < nodeList.length ==>
  @       (\forallall \bigint i; 0 <= i < \result;
  @         !o.equals(((Node)nodeList[i]).item)) &&
  @         o.equals(((Node)nodeList[\result]).item);
  @*/
public /*@ strictly_pure @*/ int
  indexOf(/*@ nullable @*/ Object o)
```

LinkedList.indexOf

We specify `LinkedList.indexOf(Object)`.

4. Loop invariant (JML)

```
/*@
  @ maintaining
  @   (\forallall \bigint i; 0 <= i < index;
  @     !o.equals((Node)nodeList[i]).item));
  @ maintaining
  @   0 <= index && index <= nodeList.length;
  @ maintaining
  @   0 <= index && index < nodeList.length ==>
  @     x == (Node)nodeList[index];
  @ maintaining
  @   index == nodeList.length <==> x == null;
  @ decreasing
  @   nodeList.length - index;
  @ assignable
  @   \strictly_nothing;
  @*/
for (Node x=first; x != null; x=x.next)
```

LinkedList.indexOf

We verify `LinkedList.indexOf(Object)`:



using the **KeY System**

(Live demo)

Chain properties

Lemma

Only the last node has `next` set to `null`.

Chain properties

Lemma

Only the last node has `next` set to `null`.

Proof.

- ▶ Goal: for every node $0 \leq i < n$ in a chain, if its `next` is `null` then $i = n - 1$.

Chain properties

Lemma

*Only the last node has **next** set to **null**.*

Proof.

- ▶ Goal: for every node $0 \leq i < n$ in a chain, if its **next** is **null** then $i = n - 1$.
1. Suppose **next** of node i is **null**.

Chain properties

Lemma

Only the last node has `next` set to `null`.

Proof.

- ▶ Goal: for every node $0 \leq i < n$ in a chain,
if its `next` is `null` then $i = n - 1$.
- 1. Suppose `next` of node i is `null`.
- 2. Either: $i = n - 1$ (trivial) or $i < n - 1$. Suppose $i < n - 1$.

Chain properties

Lemma

Only the last node has `next` set to `null`.

Proof.

- ▶ Goal: for every node $0 \leq i < n$ in a chain,
if its `next` is `null` then $i = n - 1$.
- 1. Suppose `next` of node i is `null`.
- 2. Either: $i = n - 1$ (trivial) or $i < n - 1$. Suppose $i < n - 1$.
- 3. `next` of node i is node $i + 1$ (def)

Chain properties

Lemma

Only the last node has `next` set to `null`.

Proof.

- ▶ Goal: for every node $0 \leq i < n$ in a chain,
if its `next` is `null` then $i = n - 1$.
- 1. Suppose `next` of node i is `null`.
- 2. Either: $i = n - 1$ (trivial) or $i < n - 1$. Suppose $i < n - 1$.
- 3. `next` of node i is node $i + 1$ (def)
- 4. node $i + 1$ is a node and thus not `null` (def)

Chain properties

Lemma

Only the last node has `next` set to `null`.

Proof.

- ▶ Goal: for every node $0 \leq i < n$ in a chain,
if its `next` is `null` then $i = n - 1$.
- 1. Suppose `next` of node i is `null`.
- 2. Either: $i = n - 1$ (trivial) or $i < n - 1$. Suppose $i < n - 1$.
- 3. `next` of node i is node $i + 1$ (def)
- 4. node $i + 1$ is a node and thus not `null` (def)
- 5. Contradiction of 1. and 3. by 4.



Chain properties

Lemma

Only the last node has `next` set to `null`.

Proof.

- ▶ Goal: for every node $0 \leq i < n$ in a chain,
if its `next` is `null` then $i = n - 1$.
- 1. Suppose `next` of node i is `null`.
- 2. Either: $i = n - 1$ (trivial) or $i < n - 1$. Suppose $i < n - 1$.
- 3. `next` of node i is node $i + 1$ (def)
- 4. node $i + 1$ is a node and thus not `null` (def)
- 5. Contradiction of 1. and 3. by 4.



Chain properties

Lemma

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

Chain properties

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 $0 \leq i < j < n$.

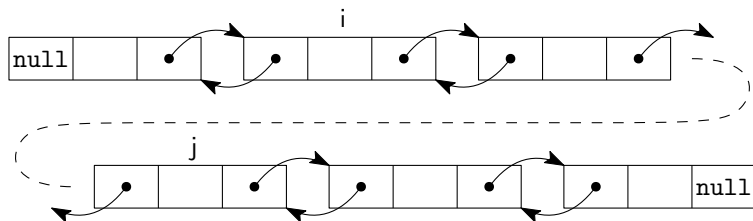
Chain properties

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 $0 \leq k < n - j$.



Chain properties

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 $j \leq k < n$.

Chain properties

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 $n - (j - i)$. (def)

Chain properties

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 $n - (j - i)$. (def)
5. Node $n - 1$ has **next** **null**. (def)

Chain properties

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 $n - (j - i)$. (def)
5. Node $n - 1$ has **next** **null**. (def)
6. Contradiction of 4. and 5. by 3. with $k = n - 1$.



Chain properties

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 $n - (j - i)$. (def)
5. Node $n - 1$ has **next** **null**. (def)
6. Contradiction of 4. and 5. by 3. with $k = n - 1$.



Back-up slides

What is equality?

We **assume** it is:

- ▶ *Terminating*
- ▶ *Deterministic*
- ▶ *Side-effect free*

Thus,

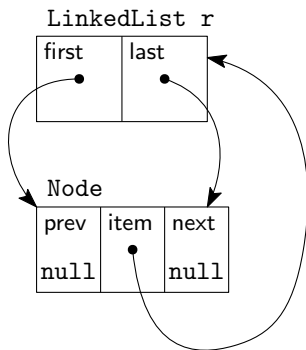
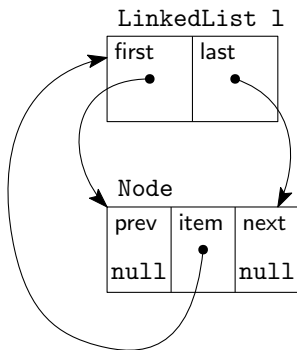
```
/*@ public normal_behavior
   @ requires true;
   @ ensures \result == self.equals(param0);
   @*/
public /*@ helper strictly_pure @*/ boolean
    equals(/*@ nullable */ Object param0);
```

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);
```

