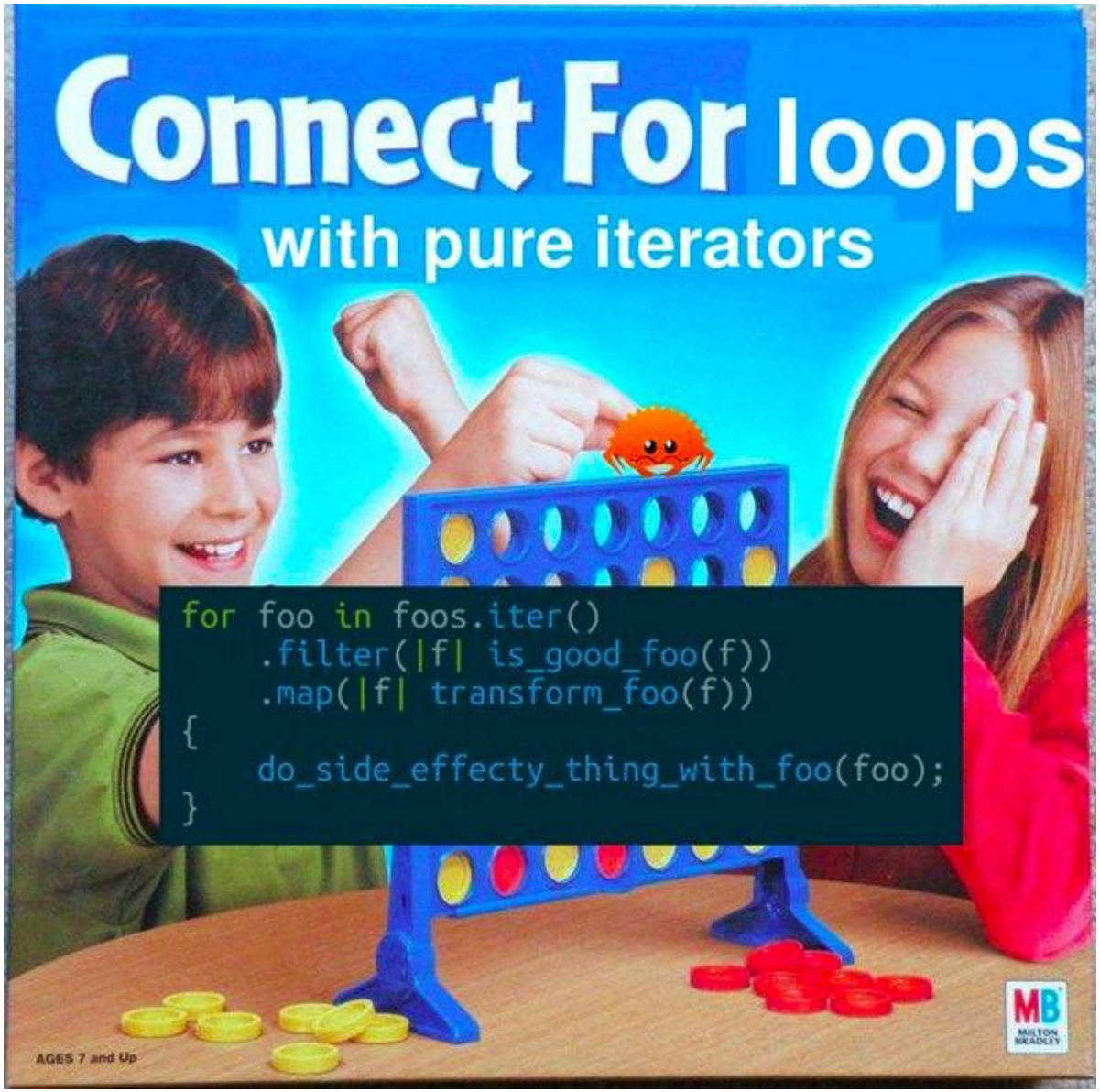


Credit: Reddit user u/mkawia

Lab 8: Iterators

CS 2112 Fall 2022

October 28 / 30, 2024

Motivation

Often we want to loop over all the elements in a collection:

```
1 for(int i = 0; i < a.length; i++) {
2     Object element = a[i];
3     // Do something with element
4 }
```

But this is cumbersome, especially if the collection does not allow for easy random access. Imagine running the same on a linked list.

```
1 for(int i = 0; i < a.size(); i++) {
2     // With linked lists, get() is an O(n) operation!
3     Object element = a.get(i);
4     // Do something with element
5 }
```

This simple loop becomes $O(n^2)$

Iterator Pattern

What we want is an efficient way to go through all the elements in a collection, independent of the collection's implementation. Of course, as with all problems in computer science, the solution is to introduce another abstraction.

We can design an iterator object that handles the details of actually getting the individual elements out of the collection. All it needs to provide to the user is the ability to check whether elements are left, and if so, to get the next one.

Interfaces

This leads to the following interface for the iterator:

```
1 interface Iterator<T> {
2     boolean hasNext();
3     T next();
4 }
```

And for the collection we can iterate over:

```
1 interface Iterable<T> {
2     Iterator<T> iterator();
3 }
```

Usage

You can use iterators directly:

```
1 Iterator<T> i = a.iterator();
2 while(i.hasNext()) {
3     T element = i.next();
4     // Process element
5 }
```

But Java provides convenient syntactic sugar for `Iterable` collections:

```
1 for(T element : a) {
2     // Process element
3 }
```

An Unhelpful Analogy

Most data structures have their values ready to access at any time repeatedly.

An iterator can only bring up its values in the order they are assigned.

That, and only once per each instance.

Basically, an iterator has a series of things that it would like to return in some order.

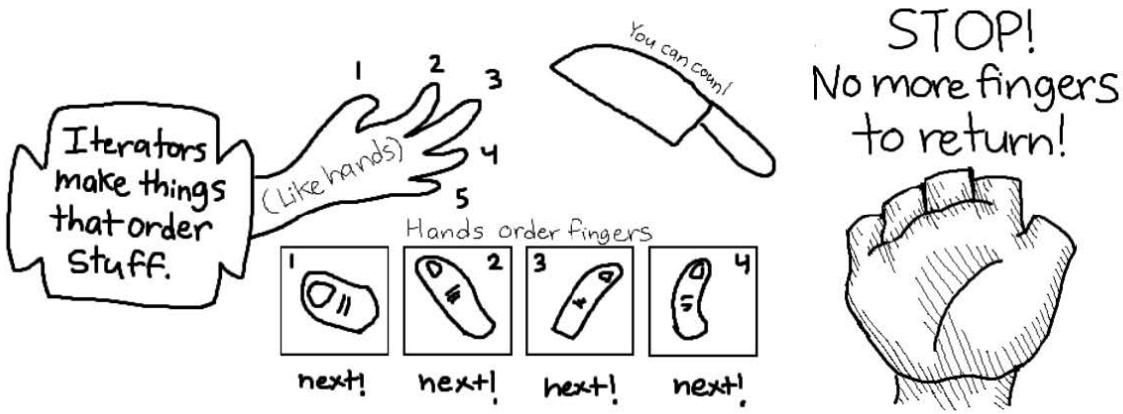


Figure: http://www-inst.eecs.berkeley.edu/~cs61a/su14/

Pitfalls to Avoid

ConcurrentModificationException

```
1 for(int x : lst) {
2     lst.remove(x); // bad bad bad
3 }
```

Linked List Iterator

Say we have a singly linked list we'd like to build an iterator for:

```
1  public class LinkedList<T> implements Iterable<T> {
2      Node head;
3      // ... other methods here
4      private class Node {
5          Node next;
6          T value;
7      }
8
9      @Override
10     public Iterator<T> iterator() {
11         // TODO make this
12         return new LinkedListIterator();
13     }
14 }
```

Implementation

Create an inner class called `LinkedListIterator` and keep track of the next unvisited node.

```
1  class LinkedListIterator implements Iterator<T> {
2      /** Next node to visit (unvisited) */
3      Node next;
4      public LinkedListIterator() {
5          next = head;
6      }
7      public boolean hasNext() {
8          return next != null;
9      }
10     public T next() {
11         if (!hasNext()) { throw new NSEE(); }
12         Node curr = next;
13         next = next.next;
14         return curr.value;
15     }
16 }
```

Binary Search Tree

Assume we have a binary search tree where each node also has a backpointer to its parent. How could we build an iterator for this tree?

- How to impose order?
- How to avoid duplicating work and wasting memory?
- How to keep track of state?

Approach

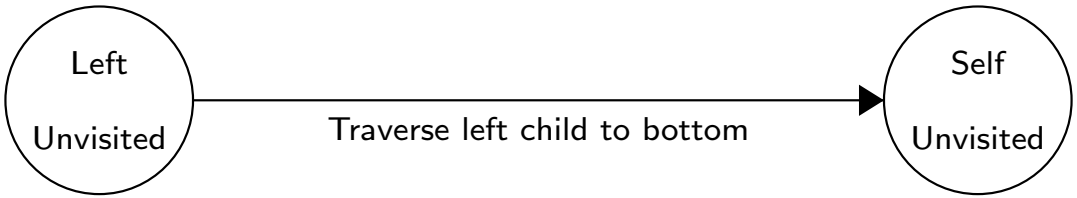
For a binary search tree, an in-order traversal returns elements in order, so that makes the most sense.
Since `next()` has to call `hasNext()` anyways, it makes sense to do all the traversal work inside `hasNext()`.
An in-order traversal has four states:

1. The left subtree is unvisited (as is everything else)
2. The left subtree is visited (but the current node is unvisited)
3. The right subtree is unvisited (but everything else is)
4. Everything rooted at this node is visited (at which point we'd want to move up to the previous parent node)

An obvious design pattern to use would be to create a **finite state machine** with these four states.

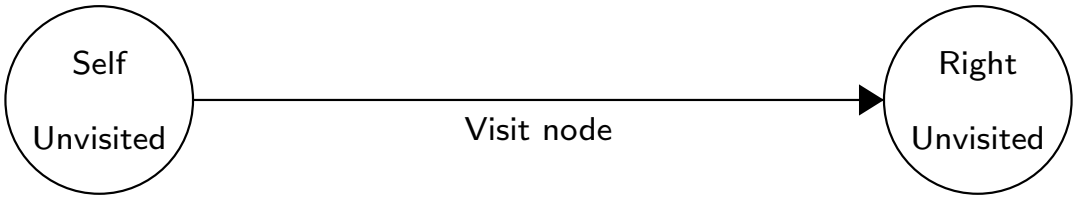
Left Unvisited State

```
1 while (!node.left.isEmpty()) {
2     node = node.left;
3 }
4 state = SELF_UNVISITED;
```



Self Unvisited State

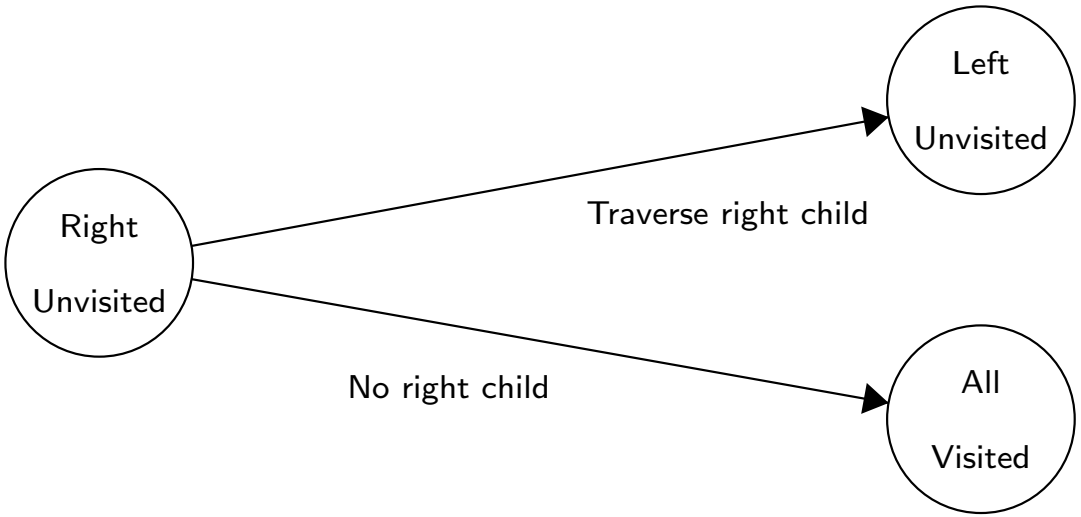
```
1 state = RIGHT_UNVISITED;
2 return node.value
```



Note this is the only state transition that happens inside the `next()` method, not inside `hasNext()`

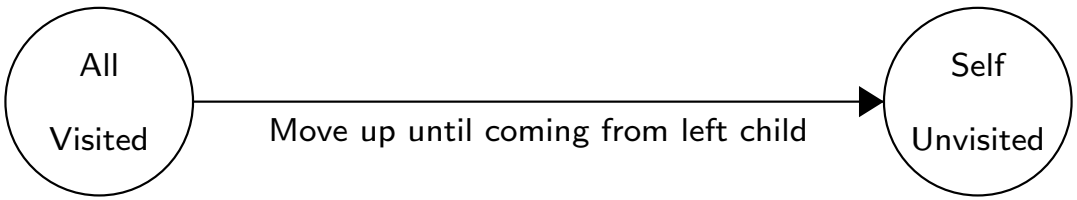
Right Unvisited State

```
1 if (node.right.isEmpty()) {
2     state = ALL_VISITED;
3 } else {
4     node = node.right;
5     state = LEFT_UNVISITED;
6 }
```

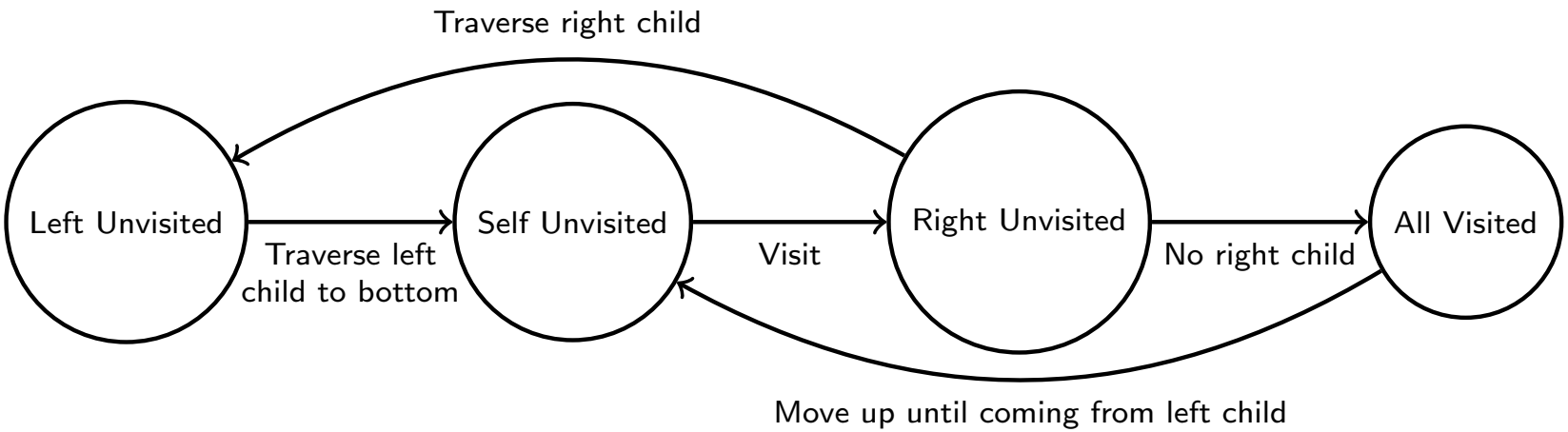


All Visited State

```
1  Node child = node;
2  while (node.left != child) {
3      child = node;
4      node = node.parent;
5      if (node.isEmpty()) {
6          return false;
7      }
8  }
9  state = SELF_UNVISITED;
```



State Transition Diagram



Code

```
1  boolean hasNext() {
2      // Do all that state transition stuff
3      return true;
4  }
5  T next() {
6      if (!hasNext()) {
7          throw new NoSuchElementException();
8      }
9      state = RIGHT_UNVISITED;
10     return node.value;
11 }
```

See the exercise files for today's lab for the full solution.

Exercise

Download the exercise from the course website under lab 8.

You will be implementing an iterator over an InputStream.

You may not use the `available()`, `mark()`, and `reset()` methods. In fact, the solution code only uses the `read()` method on the underlying stream.