Lecture 20: Playing on Trees: Iterators

CS 62
Spring 2019
William Devanny & Alexandra Papoutsaki
Pre-order Iterator

if (!isEmpty()){
    doSomething to this.value()
    left.preOrder()
    right.preOrder()
}
class BTPreorderIterator\<E\> extends AbstractIterator\<E\> {
  protected BinaryTree\<E\> root;  // root of tree to be traversed
  protected Stack\<BinaryTree\<E\>\> todo;  // stack of unvisited nodes
  public BTPreorderIterator(BinaryTree\<E\> root) {
    todo = new StackList\<BinaryTree\<E\>\>();
    this.root = root;
    reset();
  }
  public void reset() {
    todo.clear();  // stack is empty; push on root
    if (root != null)
      todo.push(root);
  }
  public boolean hasNext() {
    return !todo.isEmpty();
  }
  public E next() {
    BinaryTree\<E\> old = todo.pop();
    E result = old.value();
    if (!old.right().isEmpty())
      todo.push(old.right());
    if (!old.left().isEmpty())
      todo.push(old.left());
    return result;
  }
}
Iterators for Lists

• Method iterator easy to implement
  
  for (E elt: myList) { doSomething(elt)}

• Alternative:
  
  myList.forEach(x -> doSomething)

• Different strategies:
  
  • In first, elt from list is parameter to operation (active)
  • In second, operation is parameter to list (passive)
Anonymous classes vs lambdas

```java
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // do something
    }
});

button.addActionListener(e -> do something);
```
Functional Interfaces

@FunctionalInterface

Can define as many default and static methods as it requires.

It must declare exactly one abstract method, or the compiler will complain that it isn't a functional interface.

Can omit its name and use a lambda expression when implementing it
Lambda expressions

( formal-parameter-list ) -> { expression-or-statements }

Formal-parameter-list: list of parameters that match the parameters of the functional interface’s single abstract method

Examples:

(int x, int y) -> x+y
(x, y) -> { return x+y; }
(int x, int y) -> { System.out.println(x+y); return x+y; }
## Functional Interfaces in `java.util.Function`

<table>
<thead>
<tr>
<th>Functional Interface</th>
<th>Function descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate&lt;T&gt;</td>
<td>T -&gt; boolean</td>
</tr>
<tr>
<td>Consumer&lt;T&gt;</td>
<td>T -&gt; void</td>
</tr>
<tr>
<td>Function&lt;T, R&gt;</td>
<td>T -&gt; R</td>
</tr>
<tr>
<td>Supplier&lt;T&gt;</td>
<td>() -&gt; T</td>
</tr>
<tr>
<td>UnaryOperator&lt;T&gt;</td>
<td>T -&gt; T</td>
</tr>
<tr>
<td>BinaryOperator&lt;T&gt;</td>
<td>(T, T) -&gt; T</td>
</tr>
<tr>
<td>BiPredicate&lt;L, R&gt;</td>
<td>(L, R) -&gt; boolean</td>
</tr>
<tr>
<td>BiConsumer&lt;T, U&gt;</td>
<td>(T, U) -&gt; void</td>
</tr>
<tr>
<td>BiFunction&lt;T, U, R&gt;</td>
<td>(T, U) -&gt; R</td>
</tr>
</tbody>
</table>
## Lambdas with functional interfaces

<table>
<thead>
<tr>
<th>Use case</th>
<th>Example of lambda</th>
<th>Matching functional Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>A boolean expression</td>
<td>(List&lt;String&gt; list) -&gt; list.isEmpty()</td>
<td>Predicate&lt;List&lt;String&gt;&gt;</td>
</tr>
<tr>
<td>Creating objects</td>
<td>() -&gt; new Car(“Toyota”)</td>
<td>Supplier&lt;Car&gt;</td>
</tr>
<tr>
<td>Consuming from an object</td>
<td>(Car c) -&gt; System.out.println(c.getLicensePlates())</td>
<td>Consumer&lt;Car&gt;</td>
</tr>
<tr>
<td>Select/extract from an object</td>
<td>(String s) -&gt; s.length</td>
<td>Function&lt;String, Integer&gt;</td>
</tr>
<tr>
<td>Combining two values</td>
<td>(int a, int b) -&gt; a*b</td>
<td>BinaryOperator&lt;Integer&gt;</td>
</tr>
<tr>
<td>Compare two objects</td>
<td>(Car c1, Car c2) -&gt; c1.getLicensePlates().compareTo(c2.getLicensePlates())</td>
<td>BiFunction&lt;Car, Car, Integer&gt;</td>
</tr>
</tbody>
</table>
Java Streams

- java.util.stream
- Library of data processing methods that take functional interfaces
  - filter, map, reduce, forEach
- Examples:
  - stream.filter(x -> !x.isEmpty())
    - returns a stream that only contains non-empty elements
  - stream.map(x -> x*x)
    - returns a stream where every element is squared
  - stream.forEach(x -> System.out.println(x))
    - prints out every element in the stream
Implementing iterators with lambdas

```java
public void doPostorder(Consumer<E> action) {
    if (!isEmpty()) {
        left.doPostorder(action);
        right.doPostorder(action);
        action.accept(val);
    }
}

full.doPostorder(String s -> {System.out.println(s);});
```

Consumer is a functional interface with an abstract method action that takes an element of type E and returns type void
Calculating using lambdas

```java
public interface TrinaryFunction<E>{
    E apply(E a, E b, E c);
}

public E calcPostorder(TrinaryFunction<E> operation, E id) {
    if(!isEmpty()) {
        return operation.apply(
            left.calcPostorder( operation, id),
            val,
            right.calcPostorder(operation, id)
        );
    }
    return id;
}

System.out.println("The sum is "+ full.calcPostorder((left, root, right) -> left + root + right, 0));
```
Can’t do this

int sum = 0;
myTree.doPostorder(s -> sum = sum + s);

Local variable sum defined in an enclosing scope must be final or effectively final
Practice Time

4 classic interview problems on linked lists, queues, and stacks

Work in groups

Pick 2 problems (one from linked lists and one from queues & stacks)

Write unit tests!

Continue with a new one if done before the allotted time

Assume you can use data structures offered in structure5 package
Write a Java program that:

1) Removes duplicate nodes in an unsorted singly linked list
   • Hint: Remember that you can use two pointers to traverse a list

2) Returns the kth to last element of a singly linked list
   • Hint: Think recursion

3) Represents a queue using two stacks. Should support enqueue, dequeue, peek, size

4) Reverses a queue using a stack