Lecture 18: Playing on Trees: Iterators

CS 62

Fall 2018

Alexandra Papoutsaki & William Devanny

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;
	}
}
```

```
class BTInorderIterator<E> extends AbstractIterator<E>{
        protected BinaryTree<E> root; // root of subtree to be traversed
        protected Stack<BinaryTree<E>> todo; // stack of unvisited ancestors
        public BTPreorderIterator(BinaryTree<E> root){
                todo = new StackList<BinaryTree<E>>();
                this.root = root;
                reset();
        }
        public void reset() {
                todo.clear(); // stack is empty; push on nodes from root to leftmost descendant
                BinaryTree<E> current = root;
                while (!current.isEmpty())
                        current=current.left();
        }
        public boolean hasNext() {
                return !todo.isEmpty();
        }
        public E next(){
                BinaryTree<E> old = todo.pop();
                E result = old.value();
                if (!old.right().isEmpty())
                        BinaryTree<E> current = old.right();
                do {
                        todo.push(current);
                        current = current.left();
                }
                while(!current.isEmpty());
                return result;
        }
}
```

Iterators for Lists

- Method iterator easy to implement

```
for (E elt: myList) { doSomething(elt)}
```

- Alternative :
  `myList.forEach(x -> doSomething)`
- Implements `Comparable interface`
- Different strategies:
  - In first, elt from list is parameter to operation (active)
  - In second, operation is parameter to list (passive)

Anonymous classes vs lambdas

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

//defines an anonymous class that implements the ActionListener interface

```
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

Examples from Java 8 in Action

Lambda expressions

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

`Formal-paramater-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; }
() -> { System.out.println("Hello World!");}
```

Functional Interfaces in `java.util.function`

Functional Interface	Function descriptor
`Predicate<T>`	`T -> boolean`
`Consumer<T>`	`T -> void`
`Function<T, R>`	`T -> R`
`Supplier<T>`	`() -> T`
`UnaryOperator<T>`	`T -> T`
`BinaryOperator<T>`	`(T, T) -> T`
`BiPredicate<L, R>`	`(L, R) -> boolean`
`BiConsumer<T, U>`	`(T, U) -> void`
`BiFunction<T, U, R>`	`(T, U) -> R`

Lambdas with functional interfaces

Use case	Example of lambda	Matching functional Interface
A boolean expression	`(List<String> list) -> list.isEmpty()`	`Predicate<List<String>>`
Creating objects	`() -> new Car("Toyota")`	`Supplier<Car>`
Consuming from an object	`(Car c) -> System.out.println(c.getLicensePlates())`	`Consumer<Car>`
Select/extract from an object	`(String s) -> s.length`	`Function<String, Integer>`
Combining two values	`(int a, int b) -> a*b`	`BinaryOPerator<Integer>`
Compare two objects	`(Car c1, Car c2) -> c1.getLicensePlates().compareTo(c2.getLicensePlates())`	`BiFunction<Car, Car, Integer>`

What about those?

1. `T -> R`
2. `(int, int) -> int`
3. `T -> void`
4. `() -> T`
5. `(T, U) -> R`

Implementing iterators with lambdas

```
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

```
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

Work on both for the next ~30'
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