CSCI 230 Data Structures and Algorithms
Problem Set 2 - Data Structures
Jonathan Limpus

Assignment

This assignment is based on material from the course primary textbook, “Data Structures and Algorithms in Java” by Michael Goodrich, chapters:

- Chapter 6 Stacks, Queues, and Dequeues
- Chapter 7 List and Iterator ADTs
- Chapter 8 Trees

Problem 1. Create class `LinkedQueueOperations` which contains the single definition `public static <E> void concatenate(LinkedQueue<E> Q1, LinkedQueue<E> Q2)` which appends all elements of `Q2` to the end of `Q1`. The operation should run in constant-time and should leave `Q2` empty.

Specify the function signature if `concatenate` was a member method of the generic class `LinkedQueue<E>`. Explain your answer for partial credit.

Solution.

Code 1: `LinkedQueueOperations`

```
/**
 * concatenate - append all elements of one queue to the end of another in constant time
 * See SinglyLinkedList.concatenate for details of implemetation
 */
public static <E> void concatenate(LinkedQueue<E> Q1, LinkedQueue<E> Q2) {
  // Append Q2 to Q1 in constant-time
  SinglyLinkedList.concatenate(Q1.list, Q2.list);
  assert Q2.isEmpty() : "Error: Q2 should be empty!";
  // See https://stackoverflow.com/questions/5509082/eclipse-enable-assertions to
  // enable assertions
  // Terminal Users: java -ea EntryClass
  }
```

Code 2: `SinglyLinkedList`

```
public static <E> void concatenate(SinglyLinkedList<E> list1, SinglyLinkedList<E> list2) {

    Node<E> pointNode = new Node<E>(list2.head.getElement(), list2.head.getNext()); //
    ↪  Pointer for list2's head
    list1.tail.setNext(pointNode);
    list1.tail = list2.tail;
    list1.size += list2.size;
    // Free the memory used by the second list, effectively deleting it
    list2.head = null;
    list2.size = 0;
  }
```

□

Problem 2. Modify the `LinkedPositionalList` implementation, as described in Section 3.6 from the course primary textbook, to support the `Cloneable` interface. The class declaraction should now read `public class LinkedPositionalList<E> implements PositionalList<E>, Cloneable`.

Solution.

Code 3: LinkedPositionalList

```
public class LinkedPositionalList<E> implements PositionalList<E>, Cloneable {
    ...
    // Line 433
      public LinkedPositionalList<E> clone() throws CloneNotSupportedException{
    LinkedPositionalList<E> attackOfTheClones = (LinkedPositionalList<E>)super.clone();
    if(size() > 0) {
      attackOfTheClones.header = new Node(null, null, null);
      Node<E> iter = header.getNext(); //This node is used for iteration throughout the
      ↪  original list
      Node<E> theCloneWars = attackOfTheClones.header;  // Head of the new list

      while(iter.getNext() != null) {
        Node<E> copy = new Node(iter.getElement(), theCloneWars, iter.getNext());
        theCloneWars.setNext(copy);
        theCloneWars = copy;
        iter = iter.getNext();
      }

    }

    return attackOfTheClones;
  }
}
```

□

Problem 3. Implement a *preorder traversasl* lazy iterator for the `AbstractTree<E>` class, that is, your iterator must step through the elements of the tree in the same order a preorder traversal would. **Hint**: The `AbstractTree<E>` generic class already implements a *snapshot iterator* which you may use as a reference for your solution.

Solution.

Code 4: AbstractTree

```
public abstract class AbstractTree<E> implements Tree<E> {
    ...
    private class PreorderIterator implements Iterator<E> {
        Iterator<Position<E>> posIterator = positions().iterator();
    Stack<Position<E>> stack = new Stack<>();
    boolean isValid =true;
    @Override
    public boolean hasNext() {
      // TODO Auto-generated method stub
      return posIterator.hasNext();
    }

    @Override
    public E next() {
      // TODO Auto-generated method stub
      return helper(posIterable).getElement();
    }


    @Override
    public void remove() {
      posIterator.remove();
    }
```

```
  private Position<E> helper (Position<E> currentPosition) {
   if(parent(currentPosition) == null) //LEAVE
    continue;
   Iterator<Position<E>> siblings = children(parent(currentPosition));

   while(siblings.hasNext()) {
    if(siblings.next() == currentPosition)
     break;
   }


  }
  public Iterator<E> lazyIterator() {
   return new PreorderIterator();
  }

  public Iterable<Position<E>> lazyPreorder() {
   Queue<Position<E>> nodes = new LinkedQueue<>();
   Position<E> test = root();
   Iterable<Position<E>> iterable;

  }
  }
  ...
}
```

□

Works Cited

I received assistance from John O'Leary and Daniel Bickle in this assignment.