Object Oriented Programming (Sheet #4)

Marius Gavrilescu

1. Two `TreeNode`s are equal if they have the same key, same number of children, and their children are all equal. It is unclear whether two nodes that have the same children but in a different order should be considered equal. The following implementations considers order significant:

```
override def equals(other : Any) : Boolean = {
  if (other.isInstanceOf[TreeNode[Any]]) {
    val o : TreeNode[Any] = other.asInstanceOf[TreeNode[Any]]
    if(key != o.key) return false
    if(numberOfChildren != o.numberOfChildren) return false
    for(i <- 0 until numberOfChildren) {
      if(getChild(i) != o.getChild(i)) return false
    }
    return true
  }
  return false;
}

override def clone() : Object = {
  val ret = new TreeNode[T](numberOfChildren)
  ret.key = key
  for(i <- 0 until numberOfChildren)
    ret.setChild(i, getChild(i))
  ret
}
```

2. Returns true, as the two nodes are structurally equal because 5 is equal to 5.0 and the type parameter is erased.

4. HashSet is not covariant in its generic paramter. Mutable types are not covariant in Scala to avoid problematic code such as:

```
val setOfB : HashSet[B] = new HashSet[B]
val setOfA : HashSet[A] = setOfB
setOfA.add(new A)
setOfB.head
```

   This would be valid code if HashSet was covariant, but it is nonsensical – the head method has to return an object of type B, but the first element of the set has type A.

3. In the first case, clone will loop forever.

   In the second case, clone will clone the structure duplicating n5.

   We can avoid this by introducing a flag `isUsed` in every TreeNode, setting that flag for roots and all nodes that have parents, clearing the flag for nodes that are detached and making `setChild` require the new child to not be used.

   We add/change the following symbols in `TreeNode`:

```
private var isUsed_ : Boolean = false

def setUsed() = isUsed_ = true
def clearUsed() = isUsed_ = false
def isUsed : Boolean = isUsed_

def setChild(index : Int, child : TreeNode[T]) = {
  require(!child.isUsed)
  if(children(index) != null)
    children(index).clearUsed()
  child.setUsed()
  children(index) = child
}
```

   This also requires users of the class to call `setUsed` on every node that is the root of a tree.

5. Just keep a timestamp of changes to the LinkedList[T], and save that timestamp in the iterator.

```
class LinkedList[T] {
  private var head: LinkedList.Node[T] = null
  private var timestamp = 0

  def getTimestamp() : Int = timestamp

  def addAtHead(obj: T) = {
    timestamp = timestamp + 1
    head = new LinkedList.Node[T](obj, head)
  }

  def iterator() = new LinkedList.ListIterator[T](this)
}

protected class ListIterator[T](list: LinkedList[T]) extends Iterator[T] {
  private var current = list.head
  private val savedTS = list.getTimestamp()

  def _checkTS = {
    if (list.getTimestamp() != savedTS)
      throw new ConcurrentModificationException
  }

  override def hasNext = _checkTS; current != null
```

```
  override def next(): T = {
    _checkTS;
    if (current == null) throw new NoSuchElementException
    val value = current.obj
    current = current.next
    return value

  }

  def remove(): Unit = {
    _checkTS;
    if(current.next == null) throw new NoSuchElementException
    current.next = current.next.next
  }
}
```

6. We expand S and push $(2, a)$ and $(3, b)$ into the queue.
   Then we pop a from the queue and push $(9, c)$.
   Then we pop b from the queue and push $(8, c)$, and $(9, d)$. This can be done as a DECREASE-KEY operation to not insert c twice.
   Then we pop c from the queue and push $(12, f)$.
   Then we pop d from the queue and push $(14, f)$. This is a no-op if we're using DECREASE-KEY, as f is already in the queue with a lower cost.
   Finally we pop f from the queue and return the answer 12.

   The PriorityQueue in Scala does not offer any methods for modifying an element already in the queue, or even for removing an element that is not the highest priority one. Hence both solutions (decreasing the key of an element, removing an element and reinserting them) are infeasible with the Scala PriorityQueue.

   If we add u to the queue again with a different priority, this means the length of our priority queue is no longer bounded by N but by $N + M$ (the number of edges), which increases the complexity from $(N + M) \log N$ to $(N + M) \log(N + M)$.

7. If we keep a HashMap from coordinates to towns then we can just query the map for every pixel in an 11 by 11 square centered on the pointer.

8 9.
```
import scala.collection.mutable.HashMap
import scala.collection.mutable.HashSet
import scala.collection.mutable.MultiMap
import scala.collection.mutable.Set
import scala.ref.WeakReference
```

```
class Node(val value: Integer, val next: Node) {
  override def equals(other : Any) : Boolean = {
    if(other.isInstanceOf[Node]) {
      val o : Node = other.asInstanceOf[Node]
      value == o.value && (next eq o.next)
    } else {
      false
    }
  }

  override def hashCode() : Int = {
    var hc = value
    if(next != null) {
      hc = hc * 101 + next.hashCode()
    }
    hc
  }
}

object Node {
  var nodesCreatedCounter = 0
  private val mem : MultiMap[Integer, WeakReference[Node]] =
      new HashMap[Integer, Set[WeakReference[Node]]]
      with MultiMap[Integer, WeakReference[Node]];

  object Tester {
    def getMem : MultiMap[Integer, WeakReference[Node]] = mem
  }

  def extend(i: Integer, next: Node) : Node = {
    val nodes = mem.getOrElseUpdate(i, new HashSet[WeakReference[Node]]())
    val foundNode = nodes.find((wr) => wr.get.exists((n) => n.next eq next))
    foundNode match {
      case None =>
        nodesCreatedCounter = nodesCreatedCounter + 1
        val newNode = new Node(i, next)
        nodes.add(new WeakReference(newNode))
        newNode
      case Some(wr) => wr.get.get
    }
  }
}
```

```
object NodeTest {
  def main(args : Array[String]) {
    var x = Node.extend(10, null)
    var l1 = Node.extend(5, x)
    var l2 = Node.extend(5, x)

    assert (l1.next eq x)        // Smoke test #1
    assert (l1.next.value == 10) // Smoke test #2
    assert (l1 eq l2)            // Test that we share nodes

    var savedCreatedNodes = Node.nodesCreatedCounter
    var y = Node.extend(10, null)
    assert (Node.nodesCreatedCounter == savedCreatedNodes)
      // Another sharing test

    x = null
    l1 = null
    l2 = null
    y = null
    System.gc()

    val memValues = Node.Tester.getMem.values
    assert (!memValues.exists (
      (xs : Set[WeakReference[Node]]) =>
      xs.exists((x : WeakReference[Node]) => !x.get.isEmpty)))
      // Test gc removed all cached nodes
    x = Node.extend(10, null)
    assert (Node.nodesCreatedCounter > savedCreatedNodes)
      // Test memory leak
  }
}
```

We would strip the *Node$Tester* class and the *NodeTest* class
(because it depends on *Node$Tester*) when we distribute the library. Alternatively, we can split *NodeTest* into two classes: one that does black-box testing (which can remain in the published library) and one that does white-box testing (which would be stripped).