Databases (Sheet #1)

Marius Gavrilescu

1. (a) We will use `[]` to signify an empty bag and `a :: A` to mean "the result of adding a to the bag A".

```
STATE: queue : bag int
INIT: queue = []

clear()
POST: queue = []

isEmpty() = r : Boolean
POST: r = true if queue = [], false otherwise

insert(i : Int)
POST: queue = int :: queue

deleteMax() : Int
PRE: queue <> []
POST: r = max(queue0) and queue0 = r :: queue
```

   (b)
```
class PriorityQueue(private val MAX : Int) {
  // Concrete state space:
  val arr = new Array[Int](MAX)
  var size = 0
  // Invariant: 0 <= size < MAX
  // Abs: queue = size[0..size)

  def clear() = size = 0
  def isEmpty() = size == 0

  def insert(i : Int) = {
    assume(size < MAX)
    arr(size) = i
    size = size + 1
  }

  def deleteMax() : Int = {
    require(size > 0)
    var ans = 0
    for(i <- 1 until size) {
      if(arr(i) > arr(ans))
        ans = i
    }
```

```
      val ret = arr(ans)
      arr(ans) = arr(size - 1)
      size = size - 1
      ret
    }
  }
```

(c)
```
class BetterPriorityQueue(private val MAX : Int) {
  // Concrete state space:
  val elems = new Array[Int](MAX)
  var size = 0
  // Invariant: 0 <= size <= MAX
  // and elems(i) >= elems(2*i+1) if 0 <= i, 2*i+1 < size
  // and elems(i) >= elems(2*i+2) if 0 <= i, 2*i+2 < size
  // Abs: elems = elems[0..size)

  def clear() = size = 0
  def isEmpty() = size == 0

  def insert(i : Int) = {
    assume(size < MAX)
    var pos = size
    elems(pos) = i
    size = size + 1
    while(pos > 0 && elems((pos - 1) / 2) < elems(pos)) {
      val t = elems(pos)
      elems(pos) = elems((pos - 1) / 2)
      elems((pos - 1) / 2) = t
      pos = (pos - 1) / 2
    }
  }

  def deleteMax() : Int = {
    require(size > 0)
    val t = elems(0)
    elems(0) = elems(size - 1)
    elems(size - 1) = t
    var pos = 0
    var keep_going = true
    while(2 * pos + 1 < size && keep_going) {
      var largest = pos
      if(elems(2 * pos + 1) > elems(largest))
        largest = 2 * pos + 1
      if(2 * pos + 2 < size && elems(2 * pos + 2) > elems(largest))
        largest = 2 * pos + 2
      val t = elems(largest)
      elems(largest) = elems(pos)
      elems(pos) = t
      if(pos == largest)
        keep_going = false
      else
```

```
        pos = largest
      }
      size = size - 1
      elems(size)
    }
  }
```

2. (a)
```
class BoolArrayIntSet(private val MAX : Int) {
  // Concrete state space:
  val arr = new Array[Boolean](MAX)
  // Invariant: true
  // Abs: members = {i | 0 <= i < MAX and arr(i)}

  def insert(x : Int) = {
    require(x >= 0 && x < MAX)
    arr(x) = true
  }

  def delete(x : Int) = {
    require(x >= 0 && x < MAX)
    arr(x) = false
  }

  def contains(x : Int) = {
    require(x >= 0 && x < MAX)
    arr(x)
  }

  def isEmpty : Boolean = {
    for (i <- 0 until MAX)
      if (arr(i))
        return false
    return true
  }
}
```

This implementation does constant-time insert, delete, contains and linear-time isEmpty. It could be made to do constant-time isEmpty as well with the addition of a size field. The other implementation is slower, doing linear-time insert, delete, contains and constant-time isEmpty.

This implementation uses $O(MAX)$ memory, while the other implementation uses just $O(size)$ memory.

(b)
```
class HybridIntSet(private val MAX1 : Int, private val MAX2 : Int) {
  // Concrete state space:
  val boolSet = new BoolArrayIntSet(MAX1)
  val intSet = new IntSet(MAX2)
  // Invariant: invariant_of_boolSet && invariant_of_intSet
  // Abs: members = members_of_boolSet \cup members_of_intSet

  def insert(x : Int) = {
    if(x < MAX1)
      boolSet.insert(x)
    else
      intSet.insert(x)
  }

  def delete(x : Int) = {
    if(x < MAX1)
      boolSet.delete(x)
    else
      intSet.delete(x)
  }

  def contains(x : Int) = {
    if(x < MAX1)
      boolSet.contains(x)
    else
      intSet.contains(x)
  }

  def isEmpty = (boolSet.isEmpty && intSet.isEmpty)
}
```

3.
```
abstract class Shape(protected var d1 : Int, protected var d2 : Int){
  def setD1(d : Int) = d1 = d
  def setD2(d : Int) = d2 = d
  def equalDs : Boolean = (d1 == d2)
  def isSquare : Boolean
  def isCircle : Boolean
}

class Rectangle(private var width : Int, private var height : Int)
 extends Shape(width, height) {
  def setWidth(w : Int) = setD1(w)
  def setHeight(h : Int) = setD2(h)

  def isSquare = equalDs
  def isCircle = false
}

class Ellipse(private var sMaj : Int, private var sMin : Int)
 extends Shape(sMaj, sMin) {
  def setSemiMajor(w : Int) = setD1(w)
  def setSemiMinor(h : Int) = setD2(h)

  def isSquare = false
  def isCircle = equalDs
}
```

   Here we could also require `sMaj` $\geq$ `sMin`, but that is not a requirement.

4. One invariant is that `area = width * height`. It can be fixed by making `area` a function, not a field. Another invariant that is not enforced is `width, height > 0`.

5. One invariant is that `_floorArea = _floorSize.width * _floorSize.height`. This can fail if someone changes the Rectangle passed to the House constructor – the obvious solution is making rectangles immutable. Alternatively we could calculate floorArea on-the-fly instead of caching it (therefore getting rid of the _floorArea field).

   Another way this can fail is overflow – if width * height is larger than the maximum storable Int, we have overflow.

6.
```
Accepted an object of type A.
Accepted an object of type B.
```

   In the first call to accept, the type of variable `c` is `C` (although the object inside is of type `D`), and the only function in `C` that can handle a `B` is `accept(a: A)`. As this function is not overriden in `D`, it is used.

   In the second case we look in class `D` for a function that can handle a `B` and we find `accept(b: B)`. We then call it.