IP Lecture 19: Priority queues

Joe Pitt-Francis

Motivation for priority queues

- List sorting (heapsort);
- Scheduling;
- Prim's algorithm (minimum spanning tree);
- Dijkstra's algorithm (shortest distance);
- A*, E.g:
  - Artificial intelligence,
  - Dijkstra,
  - word-chains,
  - Sudoku;
- Huffman encoding for data compression;
- Operating systems: load balancing.

Priority queue: specifying as sequence

```
/* state:  q : seq T
 * init:  q = [] */

trait PriorityQueue[T] { //or PriorityQueue[T <% Ordered[T]]
  /** Empty the queue.          post:  q = [] */
  def clear

  /** Insert a new item x.      post:  q = q0 ++ [x] */
  def insert(x: T)

  /** Delete and return the largest item
    * pre:  q ≠ []
    * post:  returns  x s.t. x = max(q0)
    *         ∧ q = qinit ++ qtail  where  q0 = qinit ++ [x] ++ qtail */
  def delMax: T

  /** Is the queue empty?       post:  q = q0 ∧ returns  q = [] */
  def isEmpty: Boolean
}
```

Priority queue: specifying as set

```
/* state: s : ℙ T
 * init: s = {} */

trait PriorityQueue[T] {
  /** Empty the queue.          post: s = {} */
  def clear

  /** Insert a new item x.      post: s = s0 ∪ {x} */
  def insert(x: T)

  /** Delete and return the largest item
    * pre: s ≠ {}
    * post: returns x s.t. x = max(s0) ∧ s0 = s ∪ {x} */
  def delMax: T

  /** Is the queue empty?       post: s = s0 ∧ returns s = {} */
  def isEmpty: Boolean
}
```

Priority queue: specifying as bag

```
/* state: b : PT
 * init: b = {| |} */

trait PriorityQueue[T] {
  /** Empty the queue.          post: b = {| |} */
  def clear

  /** Insert a new item x.      post: b = b0 ⊎ {| x |} */
  def insert(x: T)

  /** Delete and return the largest item
    * pre: b ≠ {| |}
    * post: returns x s.t. x = max(b0) ∧ b0 = b ⊎ {| x |} */
  def delMax: T

  /** Is the queue empty?       post: b = b0 ∧ returns b = {| |} */
  def isEmpty: Boolean
}
```

Criticisms

- Specification with Sequence q is an over-specification.

  It assumes that the internal structure preserves the items in the order in which they were initially added. This ordering is not necessary (it won't be seen from the outside of the abstraction) and it may steer the developer towards an inefficient concrete class.

- Specification with Set s is flawed because elements are unique.

  This will give incorrect results in applications where duplicate elements are put into the queue.

- Specification with Bag b is the correct one.

  The new notation of $\mathcal{P}$ (power multiset) and $\uplus$ (bag union) may look a little alien, but it's just a simple change from set notation.

Unordered array (concrete datatype 1)

Staying simple, just like our first attempt at a concrete implementation of the `PhoneBook` trait, we can use an unordered array.

```
class ArrayPriorityQueue extends PriorityQueue[Int] {
  private val MAX = 100
  private val elems = new Array[Int](MAX)
  private var size = 0
```

The abstraction function is

Abs: $b = \biguplus\{\!|\ elems(i) \mid 0 \leq i < size\ |\!\}$.

with

DTI: $0 \leq size \leq MAX$.

Or, since this implementation is inspired by sequences, we could say that the abstraction function (abstracting to the first ADT) is

Abs: $q = elems[0..size)$

Unordered array: `clear`, `isEmpty`, `insert`

These operations are all straightforward.

The only complication is the addition of the precondition on `insert`, to stop the array from overflowing.

```
def clear: Unit = size = 0

def isEmpty: Boolean = (size == 0)

/** pre: size < MAX */
def insert(x: Int) {
  require (size < MAX)
  elems(size) = x
  size = size + 1
}
```

Unordered array: `delMax`

Find the maximum value (or one of them if there are duplicates).

```
def delMax():Int = {
  require(size > 0)
  var maxind = 0
  var max = elems(0)
  // Find maximum and its index
  var i = 1
  while (i < size) {
    if (elems(i) > max){max = elems(i); maxind = i}
    i += 1
  }
  // Fill the gap  --- (this version breaks seq specification)
  elems(maxind) = elems(size-1)
  size -= 1
  // Return
  max
}
```

Unordered array: factory constructor

For later testing it will become convenient to add a factory constructor from an array of integers. Works with all concrete types.

```
object ArrayPriorityQueue {
  // Factory method (trait)
  def apply(el: Array[Int]): ArrayPriorityQueue = {
    val pq = new ArrayPriorityQueue; require(el.size<=pq.MAX)
    for (x <- el) pq.insert(x);      pq
  }
}
```

For this concrete type, we could break through the abstraction:

```
  // Factory method (direct)
  def apply(el: Array[Int]): ArrayPriorityQueue = {
    val pq = new ArrayPriorityQueue; require(el.size<=pq.MAX)
    el.copyToArray(pq.elems); pq.size=el.size
    pq
  }
```

Ordered array (concrete datatype 2)

How about we use an ordered array?

If we use increasing order then the largest element is in the rightmost position and we don't need to shuffle elements on `delMax`.

```
class OrderedArrayPriorityQueue extends PriorityQueue[Int] {
  private val MAX = 100
  private val elems = new Array[Int](MAX)
  private var size = 0
```

The unchanged abstraction function is

Abs: $b = \biguplus\{\!|\ elems(i) \mid 0 \leq i < size\ |\!\}$.

with a slightly stricter invariant:

DTI: $0 \leq size \leq MAX \wedge$

$\forall i, j \bullet 0 \leq i < j < size \bullet elem(i) \leq elem(j)$.

Ordered array: `clear`, `isEmpty`, `delMax`

`clear` and `isEmpty` are unchanged.

`delMax` is now simpler.

```
  def clear: Unit = size = 0

  def isEmpty: Boolean = (size == 0)

  def delMax():Int = {
    require(size > 0)
    // Return the last elem in the array
    size -= 1
    elems(size)
  }
```

Ordered array: `insert`

`insert` has become more complicated because we have to ensure that each new element is inserted into the correct position in the array.

```
def insert(x: Int) {
  require(size < MAX)
  // Find where to put x
  // Inv: elems[0..i) < x <= elems[j..size) && 0<=i<=j<=size
  var i = 0; var j = size
  while(i<j){
    var m = (i+j)/2; // i <= m < j
    if( elems(m)<x ) i = m+1 else j = m
  }
  // Shuffle elements up
  j = size
  while (j>i){ elems(j)=elems(j-1); j -= 1}
  elems(i) = x
  size += 1
}
```

Ordered array: factory constructor

It is not possible to copy the array into the concrete `elems` and obey the ordering part of the DTI, so we just use the `insert` method. (This is the same as the first factory constructor.)

```
object OrderedArrayPriorityQueue {
  // Factory method (trait)
  def apply(el: Array[Int]): OrderedArrayPriorityQueue = {
    val pq = new OrderedArrayPriorityQueue
    require(el.size<=pq.MAX)
    for (x <- el) pq.insert(x)
    pq
  }
}
```

Heap (concrete datatype 3)

Implementing a MaxHeap (inside an array) is fairly straightforward.

```
class HeapPriorityQueue extends PriorityQueue[Int] {
  private val MAX = 100
  private val elems = new Array[Int](MAX)
  private var size = 0
```

Recall parent/child relationship:

$$\text{parent}(i) = \lfloor (i-1)/2 \rfloor, \quad \text{left}(i) = 2i+1, \quad \text{right}(i) = 2i+2.$$

The unchanged abstraction function is

Abs: $b = \biguplus \{\!|\ elems(i) \mid 0 \leq i < size\ |\!\}.$

with an even stricter invariant:

DTI: $0 \leq size \leq MAX \wedge$

$$\forall i \bullet 1 \leq i < size \bullet elem(\lfloor \tfrac{(i-1)}{2} \rfloor) \geq elem(i).$$

Heap array: clear, isEmpty, insert

```
def clear: Unit = size = 0

def isEmpty: Boolean = (size == 0)

def insert(x: Int) {
  require(size < MAX)
  // Place at final leaf. Bubble towards the root
  var i = size
  elems(i) = x
  var parent = (i-1)/2
  while (i > 0 && x > elems(parent)) {
      elems(i) = elems(parent)
      i = parent; parent = (i-1)/2
  }
  elems(i) = x
  size += 1
}
```

Heap array: `delMax`

The `heapify` "bubble-down" operation has been extracted out, to allow it to be reused.

```
def delMax():Int = {
  require(size > 0)

  val result = elems(0) // Save for later
  // Remove final element from heap and place at head
  size -= 1
  elems(0) = elems(size)

  heapify(0)
  // Return
  result
}
```

Heap array: private helper operation heapify

```
private def heapify(ind: Int) {
  // Heapify from ind down
  var i = ind
  var ch = 2*i + 1
  while (ch < size) {
    // Check if right is larger child than left
    if (ch+1 < size && elems(ch) < elems(ch+1)) ch += 1

    if (elems(i) >= elems(ch)) return // Correct place

    // else carry on bubbling down
    val temp = elems(i)
    elems(i) = elems(ch)
    elems(ch) = temp
    i = ch; ch = 2*i+1
  }
}
```

Heap array: factory constructors

Using the `insert` method of the ADT still works;

```
object HeapPriorityQueue {
  // Factory method (trait)
  def apply(el:Array[Int]) : HeapPriorityQueue = {
    val pq = new HeapPriorityQueue; require(el.size<=pq.MAX)
    for (x <- el) pq.insert(x);      pq
  }
}
```

but we can do better by copying the array and enforcing the DTI.

```
  def apply(el: Array[Int]): HeapPriorityQueue = {
    val pq = new HeapPriorityQueue; require(el.size<=pq.MAX)
    el.copyToArray(pq.elems); pq.size=el.size     /* copy data */
    for (i <- pq.size/2 to 0 by -1) pq.heapify(i) /* MakeHeap  */
    pq
  }
```

Efficiency

Here's a summary of the efficiency of all the operations we have seen.

	Unordered array	Ordered array	Heap
`clear`	$O(1)$	$O(1)$	$O(1)$
`isEmpty`	$O(1)$	$O(1)$	$O(1)$
`insert`	$O(1)$	$O(n)$	$O(\log n)$
`delMax`	$O(n)$	$O(1)$	$O(\log n)$
Factory (naive)	$O(n)$	$O(n^2)$	$O(n \log n)$
Factory (bespoke)	$O(n)$?	—	$O(n)$

Under what circumstances will an unordered array always outperform an ordered array?

Assertion testing with ScalaTest

```
class PQTest extends FunSuite{
  for(pq <- List(new ArrayPriorityQueue,
                 new OrderedArrayPriorityQueue,
                 new HeapPriorityQueue)){
    test(pq+": is empty"){
      assert(pq.isEmpty === true)
    }
    test(pq+": capacity is fixed size")
    {
      assert(pq.isEmpty === true)
      for (i<-1 to 100) pq.insert(1)
      assert(pq.isEmpty === false)
      intercept[IllegalArgumentException]{ pq.insert(1) }
      pq.delMax
      pq.insert(1)
    }
    test(pq+": clear"){ pq.clear; assert(pq.isEmpty)}
    ...
```

Assertion testing with ScalaTest

```
    test(pq+": elements come out reverse sorted")
    {
      for (i<-1 to 100) pq.insert(i)
      for (i<-100 to 1 by -1) assert(pq.delMax === i)
    }
  } // for (pq <- List(...
  test("Factory array initialisers")
  {
    val a = Array.fill(10){scala.util.Random.nextInt(5)}
    val pqa = ArrayPriorityQueue(a); val pqh = HeapPriorityQueue(a)
    val pqo = OrderedArrayPriorityQueue(a)
    for (i <- 1 to 10){
      val maxa=pqa.delMax; val maxo=pqo.delMax; val maxh=pqh.delMax
      assert(maxa === maxo); assert(maxo === maxh)
    }
    assert(pqa.isEmpty); assert(pqo.isEmpty); assert(pqh.isEmpty)
  }
}
```

Property testing with ScalaCheck

```
object PQSpecification extends Properties("PQ of Int") {
  val smallInt = Gen.choose(-10,10) //Range of elements
  property("Ints in [-10,10]")=forAll(smallInt){n =>  n>= -10 && n<=10}

  val listInts = Gen.resize(100, Gen.listOf(smallInt))
  property("List not too long")=forAll(listInts){l => l.size<=100}

  for (pq <-List(new ArrayPriorityQueue,new OrderedArrayPriorityQueue,
                 new HeapPriorityQueue)){
    property(pq+": sorted") = forAll(listInts){
      alist =>
        for (w <- alist) pq.insert(w) // Fill queue from array
        var out:List[Int] = List()    // Place to store output
        for (w <- alist) out = pq.delMax() :: out //Add to head
        ( alist.sorted == out && pq.isEmpty )     // Result
    }
  }
}
```

Summary

- Priority queue specification;
- Implementation as array;
- Implementation as ordered array;
- Implementation as heap;
- Testing.
- Next time: Sudoku.