Heaps

Michael P. Fourman

February 2, 2010

Introduction

We have introduced several abstract datatypes, dictionaries, sets, queues, stacks, and priority queues. The Project demonstrates the usefulness of these ADTs, which we have implemented using lists. We have also introduced other concrete datatypes, trees, arrays, and references. In the next few lectures, we use these concrete datastructures to provide more efficient implementations of priority queues, sets, and dictionaries. Trees play a special role in many of these structures.

Our first example uses a tree to record and organise the results of comparisons we have made between entries in a priority queue, so that we can exploit these results to avoid making unnecessary comparisons while implementing the priority queue operations. Throughout this note we consider a type `Item`, equipped with a priority ordering, `<=`, which is transitive (if $a <= b$, and $b <= c$, then $a <= c$). The data-structure is a special kind of labelled tree, called a *heap.*

Definition 1 *A* **heap** *is a tree with items stored at the nodes, with the property that*

> *the priority of the item labelling a node is at least as great as the priorities of each of the items labelling its children.*

Thus the label at the root of a heap is an item of maximal priority, any subtree of a heap is a heap, and the labels encountered along any path down the tree will be in non-increasing order.

An ordered list is a degenerate form of heap. Earlier, we used an list in non-decreasing order to implement a priority queue. This made `deq` an $O(1)$ operation, and `enq` an $O(n)$ operation where n is the number of entries in the queue. We shall see that we can reduce the amortised cost of adding

and removing an element from a priority queue to $O(\lg n)$, by using suitably balanced trees for our heaps

We will use heaps to implement priority queues. Finding an item of maximal priority is easy, we use the label at the root. We must address two problems. First, if we remove the root we are left with a forest of subtrees (each of which is a heap). To complete an implementation of `deq` we must combine these to form a new heap. Second, we need to add an entry to a heap to implement `enq`. We also want to keep our heaps balanced while we perform these operations. We consider two varieties of heap: Vuillemin heaps, based on bushy trees, and binary heaps, based on binary trees.

Vuillemin heaps

We use the following datatype of trees

```
datatype Tree = T of Item * Tree list
```

One easy way to make new heaps from old is to make one a child of the root of the other. By doing this the right way round, we can maintain the heap property.

```
fun join (a as T(av, aa)) (b as T(bv, bb)) =
    if bv <= (av:int) then T(av, b :: aa)
                      else T(bv, a :: bb)
```

When we join two heaps using this operation, the result is a heap. This is the operation we will use to build Vuillemin heaps. A heap with a single entry, x, is given by the leaf, `T(`x`, [])`. To ensure that our trees are reasonably balanced, we will only ever join two trees of the same size.

Definition 2 *A* **Vuillemin heap** *(V-heap) of degree* 0 *is a leaf; a Vuillemin heap of degree* $n+1$ *is obtained by joining two Vuillemin heaps of degree* n*.*

It is easy to check, by induction, that a V-heap of degree n has height n, and has 2^n nodes. The children of the root node form a list of n V-heaps, whose degrees are $n-1, \ldots, 0$.

Since the size of a V-heap is a power of 2, a priority queue can only be represented by a single V-heap if its size, s, is a power of 2. However, the binary expansion of a number s represents s as a sum of powers of two, so we can represent any queue by a list of V-heaps. The list will include a heap of size 2^i if the binary expansion of s has a 1 in the i^{th} place; in all, it will include at most $\lg s$ heaps, as there are $\lg s$ digits in the binary expansion of s.

We take this idea a few steps further. The binary representation of a number is a sequence of zeroes and ones. The V-representation of a priority

queue is similar; we represent a queue as a sequence of `Zero`es, and `One`s. A `One` in the i^{th} place carries with it a V-heap of size 2^i.

```
datatype Digit = Zero | One of Tree

type PQueue = Digit list
```

The "least significant" digit comes at the head of the list.

We add two binary numbers by adding together corresponding digits, and where necessary "rippling" a carry to the next place. Just as we can add two binary numbers, we can add together two queues. When we add two digits, adding a `Zero` and anything is trivial; to add two `One`s, we join the corresponding trees to create a carry `One` of the appropriate size.

```
fun carry Zero     x            = x
  | carry x        []           = [x]
  | carry x       (Zero :: aa)  = x :: aa
  | carry (One c) (One a :: aa) = Zero :: carry (One(join c a)) aa

fun add (One a :: aa) (One b :: bb) =
                               Zero :: carry (One(join a b)) (add aa bb)
  | add (Zero :: aa) (b :: bb) = b :: add aa bb
  | add (a :: aa) (Zero :: bb) = a :: add aa bb
  | add aa [] = aa | add [] bb = bb
```

Joining two trees is an $O(1)$ operation. When we add an element to a queue of size $2^n - 1$, we need to perform n join operations; so, in the worst case, adding an element is an $O(\lg N)$ operation, where n is the size of the queue. If we build a queue by repeatedly adding a new queue of size 1 to an existing queue, then we create a carry whenever the size of the existing queue is odd—exactly half the time. Half of these carries ripple to the next place, and so on. The average number of join operations for each item we add is bounded by the sum $1/2 + 1/4 + \ldots + 1/2^n \ldots$, which has a limiting value of 1. So the total cost of adding a sequence of k elements to a queue is $O(k)$; the amortised cost of adding each element is $O(1)$. This analysis is valid when we add $\lg N$ elements to the queue in succession.

To implement `deq` we must examine the root of each tree, to find one with highest priority. A call from `deq` to `best` recurses down the list of digits. If the heap is empty it raises the exception `Deq`. When a tree, a, is found, a recursive call looks for a better tree further down the list; if no other tree is found, the exception is handled by returning a. We call the tree we find the "best" tree.

We must remove the best tree from the data-structure (replacing it with a `Zero`), and then re-integrate its children with the rest of the data-structure.

The function `addchildren` performs both these tasks.

The height of the best tree, and hence its position in the digit list, is given by the number of children of its root. We find the right digit to remove by scanning down the list of children and the list of digits, in step. When the list of children is exhausted, we have arrived at the parent; we remove the corresponding `One` from the queue, (and replace it with a `Zero`).

The children are then added back into the queue as carries, as we unwind the recursion. The binary expansion of one less than a power of 2 is a list of ones. The children of a V-tree are V-trees of the appropriate sizes to add in to the data-structure, but the list of children must be reversed so that they occur in the the right order.

Since the number of digits is $\lg s$, these operations are all $O(\lg n)$, where n is the size of the queue.

```
functor VHEAP(type Item val < : Item * Item -> bool):QueueSig =
struct exception Deq

    datatype Tree  = T of Item * (Tree list)
    datatype Digit = Zero | One of Tree
    type Item = Item and Queue = Digit list
    (* if One(t) occurs in the nth place, t is a V-heap of size 2^n *)

    val empty = []
    fun isEmpty [] = true | isEmpty (Zero :: t) = isEmpty t | isEmpty _ = false

    fun join (a as T(av, aa)) (b as T(bv, bb)) =
                      if bv < av then T(av, b :: aa) else T(bv, a :: bb)

    fun carry c  []           = [One c]
      | carry c (Zero  :: aa) = One c :: aa
      | carry c (One a :: aa) = Zero :: carry (join c a) aa

    fun add (One a :: aa) (One b :: bb) =
                       Zero :: carry (join a b) (add aa bb)
      | add (Zero :: aa) (b :: bb) = b :: add aa bb
      | add (a :: aa) (Zero :: bb) = a :: add aa bb
      | add aa [] = aa | add [] bb = bb

    fun enq(q, a) = carry (T(a,[])) q

    fun better (a as T(av,_)) (b as T(bv,_)) = if av < bv then b else a

    fun best (Zero  :: aa) =  best aa
      | best (One a :: aa) = (better a (best aa) handle Deq => a)
      | best [] = raise Deq

    fun addchildren [] (_ :: aa) = (case aa of [] => []
                                             |  _ => Zero :: aa)
      | addchildren (h :: t) (a :: aa) =
           carry h (a :: addchildren t aa)

    fun deq q = let val T(av,aa) = best q (* will raise Deq if appropriate *)
                in (addchildren (rev aa) q, av) end
end;
```

Binary heaps

Another way of using a heap to implement a priority queue is to use a binary tree. Again, we have to do some work to maintain the heap property, and ensure that our trees don't become unbalanced, when we add and remove items from the tree.

To insert an element in a binary heap is easy. For a non-empty tree, if the new element should come below the root, we recursively insert the element in one or other of the subtrees, otherwise we place the new element at the root, and recursively insert the old root in one of the subtrees. (Inserting in an empty tree (a leaf) is *very* easy.)

To maintain a *balanced* heap, we should insert elements alternately in to one or other subtree. To achieve this effect, we always insert into the same subtree, either left or right, but swap the subtrees around each time we make an insertion. (This trick is due to Arthur Norman.)

```
fun insert x Lf = Nd(Lf, x, Lf)
  | insert x (Nd(lt, v, rt)) =
       if v < x then Nd(rt, x, insert v lt)
       else          Nd(rt, v, insert x lt)
```

In this way we can use a sequence of insertions to produce a balanced heap. The function insert can be used to enqueue items in a priority queue.

A sequence of insertions produces a sequence of trees of different shapes; for each size of queue there is an appropriate shape. When we dequeue the item of highest priority from a tree of size $n+1$, we want to produce a tree of the right shape for size n.

```
fun del (Nd(Lf, v, Lf)) = (Lf, v)
  | del (Nd(lt, v, rt)) =
     let val (rt', x) = del rt
     in (Nd(rt', v, lt), x) end
```

The function `del` provides an "inverse" for `insert`, in the sense that if $(t', e) = \texttt{del}(\texttt{insert}(x, t))$ then t and t' have the same shape. Furthermore, $\texttt{leaves}(t) \cup \{x\} = \texttt{leaves}(t') \cup \{e\}$, and applying `del` obviously preserves the heap property. So we have removed an entry and made a new heap. But the entry removed will not normally have highest priority; we find *that* entry at the top of the heap, and we've just removed an item from the bottom. To implement `deq`, we must replace the root node of the new tree by the entry we have just removed.

```
fun deq Lf = raise Deq
  | deq (Nd(Lf, v, Lf)) = (v, Lf)
  | deq t = let val (t' as Nd(_, v, _), x) = del t
```

```
                in (v, replaceRoot t' x) end
```

The call `replaceRoot` t' x takes a heap, t', and produces a heap of the same shape, and *except* that the root entry is replaced by x, the same entries. If x dominates the rest of the tree, we can replace the root directly. Otherwise, we copy the root of the dominant sub-tree into top position, and recursively replace this entry in the subtree. This recursion continues until we find an appropriate level for x.

```
fun replaceRoot (Nd(lt, _, rt)) x =
    case rt of Lf => Nd(Lf, x, Lf) (* lt is also a leaf *)
    | Nd(_, rv, _) =>
      (case lt of Lf  => if x < rv then Nd(Lf, rv, Nd(Lf, x, Lf))
                                   else Nd(Lf, x, rt)
       | Nd(_, lv, _) => if rv < lv then
                         if x < lv then Nd(replaceRoot lt x, lv, rt)
                                   else Nd(lt, x, rt)
                         else (* lv <= rv *)
                         if x < rv then Nd(lt, rv, replaceRoot rt x)
                                   else Nd(lt, x, rt))
```

The idea is simple, but the code is complicated by the need to consider a number of cases. By construction, our trees always lean to the right (if at all), so if a right subtree is a leaf, so is its sibling. If a left subtree is a leaf, its sibling's height is at most 1.

Binary heaps don't support an efficient implementation of heap merge and, in this functional implementation, they are slower than Vuillemin heaps. They are important because a mutable implementation of binary heaps can carry out the necessary tree manipulations "in place".

Mutable Binary Heaps

Consider a full, infinite binary tree whose nodes are numbered, from 1, in breadth-first order. Each level of the tree has a certain number of nodes, at the next level there are twice as many nodes. The first level consists of just the root, the i^{itth} level has 2^{i-1} nodes. The two children of the node labelled n are labelled $2n$ and $2n+1$; the parent of a non-root node labelled n is labelled $2/n$.

We will implement a priority queue of size s as a binary heap using a finite tree corresponding to the nodes labelled $1, \ldots, s$. As with our functional implementation, we fix on a shape of tree for each size of heap, but the shapes are not quite the same.

To add an element to the queue, we place the item in the next node, and then move the element up the path to the root node, using a function `upheap`, by repeatedly swapping it with its parent, until it reaches the right level to satisfy the heap property. When we dequeue an element, we replace the root node with the element from the last node; we then use a function `downheap` (analogous to `replaceRoot`) to move this element down the heap until the heap property is satisfied.

Since these operations require us to find both the parent and the children of a node, a straightforward tree datatype is inappropriate (as it only lets us find the children of a node). Instead, we use an array, a, to represent the tree; the label of the node numbered i is stored in the entry $a[i]$. We implement a mutable priority queue. The functions `upheap` and `downheap` do the work of maintaining the heap property.

```
infix sub;
functor BHEAP(type Item val <= : Item * Item -> bool):
    sig type Heap
type Item
val empty : int -> Heap
val enq : Heap * Item -> unit
val deq : Heap -> Item
val isEmpty : Heap -> bool
    end
    =
struct
    open Array
    fun swap a i j = let val t = a sub i
                     in update(a,i,a sub j); update(a,j,t) end

    exception Deq and FullHeap
    datatype Entry = Void | Some of Item
    type Item = Item
    type Heap = int ref * Entry Array.array
    infix <<=
    fun (Some x) <<= (Some y) = x <= y
      | _ <<= _               = true

    fun empty n = (ref 0,Array.array(n+1, Void))
    fun isEmpty (ref n,_) = n = 0

    fun parent i = i div 2 and left i = 2 * i and right i = 2 * i + 1

    fun upheap a 1 = ()
      | upheap a i = let val p = parent i in
                         if (a sub i) <<= (a sub p) then ()
                         else (swap a i p; upheap a p) end

    fun downheap (n,a) i =
        let val l = left i and r = right i
            val b = if l < n andalso (a sub i) <<= (a sub l) then l else i
            val c = if r < n andalso (a sub b) <<= (a sub r) then r else b
        in if c = i then () else (swap a i c; downheap (n,a) c) end

    fun enq((n,a), x) = if !n + 1 = length a then raise FullHeap
                        else (n := !n +1;
   update(a,!n,Some x);
   upheap a (!n))
```



```
    fun deq(n,a)= if !n < 1 then raise Deq else
                  let val Some v = a sub 1
                  in  update(a, 1, a sub (!n));
                      update(a, !n, Void);
                      downheap (!n,a) 1;
```