

Functional Binomial Queues

David J. King
University of Glasgow*

Abstract

Efficient implementations of priority queues can often be clumsy beasts. We express a functional implementation of binomial queues which is both elegant and efficient. We also quantify some of the differences with other functional implementations. The operations `decreaseKey` and `delete` always pose a problem without destructive update, we show how our implementation may be extended to express these.

1 Functional priority queues

A crucial part of many algorithms is the data structure that is used. Frequently, an algorithm needs an abstract data type providing a number of primitive operations on a data structure. A priority queue is one such data structure that is used by a number of algorithms. Applications include, Dijkstra's [4] algorithm for single-source shortest paths, and the minimum cost spanning tree problem (see Tarjan [12] for a discussion of minimum spanning tree algorithms). See Knuth [8] and Aho *et al* [1] for many other applications of priority queues.

A priority queue is a set where each element has a key indicating its priority. The most common primitive operations on priority queues are:

<code>emptyQ</code>	Return the empty queue.
<code>isEmpty q</code>	Return <code>True</code> if the queue <code>q</code> is empty, otherwise return <code>False</code> .
<code>insertQ i q</code>	Insert a new item <code>i</code> into queue <code>q</code> .
<code>findMin q</code>	Return the item with minimum key in queue <code>q</code> .
<code>deleteMin q</code>	Delete the item with minimum key in queue <code>q</code> .
<code>meld p q</code>	Return the queue formed by taking the union of queues <code>p</code> and <code>q</code> .

In addition, the following two operations are occasionally useful:

<code>delete i q</code>	Delete item <code>i</code> from queue <code>q</code> .
<code>decreaseKey i q</code>	Decrease the key of item <code>i</code> in queue <code>q</code> .

There are numerous ways of implementing the abstract data type for priority queues. Using *heap-ordered* trees is one of the most common implementations. A tree is heap-ordered if the item at every node has a smaller key than its descendents. Thus the entry at the root of a heap has the earliest priority. A

*Author's address: Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: gnik@dcs.gla.ac.uk. URL: <http://www.dcs.gla.ac.uk/~gnik>.

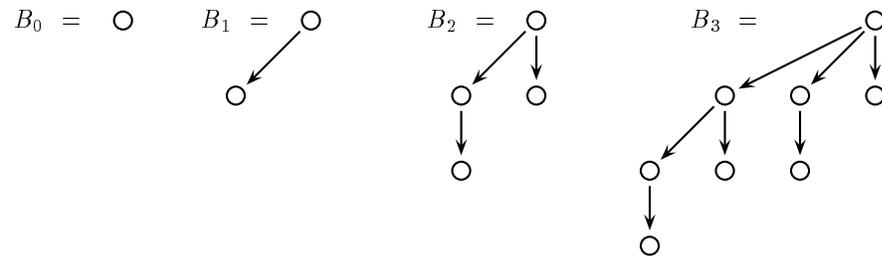
variety of different trees have been used including: heaps [8], splay trees [11], skew heaps [11], 2-3 trees [1]. In addition, lists (sorted or unsorted) are another possible implementation of queues, but will be less efficient on large data sets. For a comparative study of these implementations and others in an imperative paradigm see Jones [7].

There is very little in the literature on priority queues in a functional paradigm. Heaps are the most common functional implementation, see Paulson [9], for example. The disadvantage of using heaps, or balanced trees is that more bookkeeping is required for balancing. This extra bookkeeping adds to the amount of storage space needed by the queue, as well as making the implementation of the primitives more complex. We present a functional implementation of priority queues that is based on trees, but does not require as much storage space or balancing code as other implementations. The implementation is far more elegant than a typical imperative implementation, lending itself well to a formal proof of correctness.

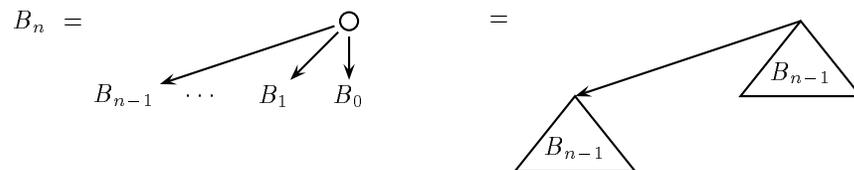
Vuillemin [13] describes *binomial queues* which support the full complement of priority queue operations in $O(\log n)$ worst-case time. They are based on heap-ordered trees in that a priority queue is represented by a list of heap-ordered trees (that is, a forest), where each tree in the forest is a binomial tree. We present a purely functional implementation of binomial queues expressing the full complement of priority queue operations in Haskell (Hudak *et al* [6]).

2 Binomial trees

Binomial trees are general trees that have a regular shape. They are best presented diagrammatically:



There are two equally good ways of expressing the general case, for $n \geq 1$.



In Haskell we define a general tree with the datatype:

```
data Tree a = Node a [Tree a]
```

Then using this datatype we can define binomial trees inductively:

$$\begin{aligned} B_0 &= \text{Node } x \ [] \\ B_n &= \text{Node } x \ [B_{n-1}, \dots, B_1, B_0], \text{ for } n > 0 \end{aligned}$$

We may verify that a general tree is the binomial tree B_k with the following function:

```
isBinTree :: Int -> Tree a -> Bool
isBinTree k (Node x []) = k==0
isBinTree k (Node x ts) = and (zipWith isBinTree [0..]
                                (reverse ts))
```

This works by checking that the last subtree is a B_0 tree, and then checking that the penultimate subtree is a B_1 tree, and so on for all subtrees.

Binomial trees have some very nice combinatorial properties. For instance, the binomial tree B_k has 2^k nodes, and $\binom{k}{d}$ nodes of depth d , hence their name. See Vuillemin [13] and Brown [3] for more properties.

3 Implementing binomial queues functionally

Vuillemin [13] represents a priority queue with a forest of binomial trees. It is important that a list of trees is used to represent the forest because the ordering is important (a set of trees would not do). The first tree in the binomial queue must either be a B_0 tree or just *Zero* meaning no tree, and the second a B_1 tree or just *Zero* so we have the following structure for a binomial queue:

$$[T_0, T_1, \dots, T_n] \text{ where } T_k = \text{Zero} \mid B_k \text{ for } 0 \leq k \leq n.$$

Vuillemin [13] and others use an array to represent the forest, moreover, for simplicity binary trees are used to represent the binomial trees. Imperative implementations of linked structures of this kind usually turn out to be clumsy. Instead we express the primitives as recursive functions on a list of general trees, giving a very natural encoding.

So the following datatypes are used:

```
type BinQueue = [Maybe (Tree Item)]
data Maybe a = Zero | One a
```

We use the standard **Maybe** datatype with constructors **Zero** and **One**, these names were chosen because the queue primitives are analogous with binary arithmetic.

Each item is a pair of entry and key:

```
type Item = (Entry, Key)
```

So the projection functions on items are:

```
entry :: Item -> Entry
entry = fst
```

```
key :: Item -> Key
key = snd
```

The following functions may be used to verify that a list of trees has the right structure to be a binomial queue.

```
isBinQ :: BinQueue -> Bool
isBinQ q = isBinQTail 0 q

isBinQTail :: Int -> BinQueue -> Bool
isBinQTail k q = and (zipWith isTree [k..] q)
  where isTree m Zero    = True
        isTree m (One t) = isBinTree m t
```

The correctness of the queue primitives that follow may be shown by using the above functions. For example, the correctness of the `meld` function is shown by proving:

$$\forall p, q. \text{isBinQ } p \wedge \text{isBinQ } q \Rightarrow \text{isBinQ } (\text{meld } p \ q)$$

Now we can start to express the priority queue operations. Creating a new empty queue, and testing for the empty queue follow immediately:

```
emptyQ :: BinQueue
emptyQ = []

isEmpty :: BinQueue -> Bool
isEmpty q = null q
```

Taking the union of (or melding) two queues together is the most useful of all the primitive operations (because other primitives are defined with it) so we describe it next. There is a very strong analogy between `meld` and binary addition. Given two binomial queues $[P_0, P_1, \dots, P_n]$ and $[Q_0, Q_1, \dots, Q_m]$ melding is carried out positionally from left to right, using the property that two binomial trees B_k can be linked into a B_{k+1} binomial tree. First P_0 is melded with Q_0 , giving one of four possible results. If both P_0 and Q_0 contain trees (that is, they are not **Zero**) they are linked to form a B_1 tree so that the heap-order property is maintained. With just one tree and one **Zero** the result is the tree, and given two **Zero**'s the result is **Zero**. This process of linking is carried out on successive trees. If the result of melding P_k with Q_k results in a B_{k+1} tree then this is carried on (it is analogous to carry in binary arithmetic) and melded with P_{k+1} and Q_{k+1} .

```

meld :: BinQueue -> BinQueue -> BinQueue
meld p q = meldC p q Zero

meldC :: BinQueue -> BinQueue -> Maybe (Tree Item) -> BinQueue
meldC [] q Zero = q
meldC [] q c = meld [c] q
meldC p [] c = meldC [] p c
meldC (Zero:ps) (Zero:qs) c = c:meld ps qs
meldC (One (Node x xs): ps) (One (Node y ys): qs) c
= let t =
    if key x < key y
    then Node x (Node y ys: xs)
    else Node y (Node x xs: ys)
  in c:meldC ps qs (One t)
meldC (p:ps) (q:qs) c = meldC (q:ps) (c:qs) p

```

The asymptotic complexity of `meld` is $O(\log n)$ (where n is the number of items in the largest queue). We arrive at this by observing that two B_k trees can be linked in constant time, and the number of these linking operations will be equal to the size of the longest queue, that is $O(\log n)$. For a more detailed analysis of the complexity of `meld` and the other queue operations see Brown [3].

Inserting an item into the queue is most simply expressed by melding a B_0 tree holding the item, into the binomial queue.

```

insertQ :: Item -> BinQueue -> BinQueue
insertQ i q = meld [One (Node i [])] q

insertMany :: [Item] -> BinQueue
insertMany is = foldr insertQ [] is

```

As each binomial tree is heap-ordered the item with the minimum key will be a root of one of the trees. This is found by scanning the list of trees. To delete the item with minimum key we extract the tree that it is the root of, and meld the subtrees back into the binomial queue. This melding is easy as the subtrees themselves form a binomial queue, in reverse order. The subtrees could be stored in reverse order, but this would make it more difficult to define an efficient version of `meld`.

The extraction of the tree with minimum key is done in two steps. We first traverse the forest returning the tree:

```

minTree :: BinQueue -> Maybe (Tree Item)
minTree q = foldl1 least q
  where
    Zero          'least' t      = t
    t             'least' Zero  = t
    One (Node x xs) 'least' One (Node y ys)
                                     = if key x < key y
                                       then One (Node x xs)
                                       else One (Node y ys)

```

Then we delete the tree with the minimum key item from the binomial queue:

```

removeTree :: Item -> BinQueue -> BinQueue
removeTree i q = map match q
  where
    match Zero          = Zero
    match (One (Node x xs)) | x==i = Zero
                             | x/=i = One (Node x xs)

```

These functions are combined and the subtrees of the extracted tree are melded back into the queue:

```

deleteMin :: BinQueue -> BinQueue
deleteMin q = meld (map One (reverse ts)) (removeTree i q)
  where
    One (Node i ts) = minTree q

```

The total running time of `minTree` and `removeTree` is $O(\log n)$. As both operations traverse a list of length $\log n$ carrying out constant time operations. The `deleteMin` operation carries out a meld, as well as `minTree` and `removeTree`. Since the subtrees being melding back into the queue are smaller, the melding will take $O(\log n)$ time. Hence `deleteMin` will run in $O(\log n)$ time.

We may also use `minTree` to express `findMin` which again runs in $O(\log n)$ time.

```

findMin :: BinQueue -> Item
findMin q = i
  where
    One (Node i ts) = minTree q

```

The two pass algorithm for `deleteMin` can be performed in one pass over the binomial queue (giving a constant time speed-up) by using the standard cyclic programming technique, see Bird [2]. A function is used that both takes the item to be removed as an argument and returns the item with minimum key, as well as the binomial queue without the item. As is usually the case with more efficient algorithms, the implementation becomes more cumbersome, so we omit it here.

4 Comparison with other priority queues

Imperatively binomial queues perform better than most other priority queue implementations, see Jones [7] for an empirical comparison. More recently Fredman and Tarjan [5] have developed Fibonacci heaps which are based on binomial queues. Fibonacci heaps have a better amortised complexity for many of the operations. Unfortunately, they make heavy usage of pointers, so do not lend themselves to a natural functional encoding.

Queue	insertQ		deleteMin		meld	
	Lines	Time	Lines	Time	Lines	Time
Binomial	1	$O(\log n)$	13	$O(\log n)$	12	$O(\log n)$
2-3 trees	16	$O(\log n)$	41	$O(\log n)$	26	$O(n)$
Sorted list	5	$O(n)$	1	$O(1)$	6	$O(n)$
Heaps	7	$O(\log n)$	17	$O(\log n)$	21	$O(n)$

Table 1: Differences between some Haskell implementations of priority queues

The usual functional implementation of priority queues is to use heaps, see Paulson [9], for example. The advantage of binomial queues over heaps is that the meld operation is more efficient. Imperatively Jones [7] reports that binomial queues are one of the most complex implementations. In Haskell the operations on tree and list data structures are far cleaner than in an imperative language. Functionally, binomial queues are in many ways more elegant than heaps. They are easier to program and understand, as well as being programmed in fewer lines of code. Similarly binomial queues have the same advantages over 2-3 trees, see Reade [10] for a functional implementation of 2-3 trees, and Aho *et al* [1] for a description of how they may be used for implementing priority queues. Sorted lists are the simplest of all implementations, and give the best performance for small queues. However, they have the worst complexity and will give slower running times for larger queues. Table 1 summarises the running times and lines of Haskell code for four different implementations.

5 Implementing decreaseKey and delete

The usual way in imperative languages to implement `decreaseKey` and `delete` is to maintain an auxiliary data structure which supports direct access, in constant time, to each item. Functionally such a data structure is not feasible. However, we do use an auxiliary data structure, a set. A set of all the items currently in the queue is maintained along with the queue.

```
type BinQueueExt = (BinQueue, Set Entry)
```

All the priority queue operations must do some extra bookkeeping to maintain the set.

```
emptyPQ :: BinQueueExt
emptyPQ = (emptyQ, emptySet)
```

```
isEmptyPQ :: BinQueueExt -> Bool
isEmptyPQ (q,s) = isEmptySet s
```

When inserting a new item we must also insert it into the set. Similarly, when two queues are melded we must take the union of their sets:

```
insertPQ :: Item -> BinQueueExt -> BinQueueExt
insertPQ i (q,s) = (insertQ i q, insSet (entry i) s)
```

```
meldPQ :: BinQueueExt -> BinQueueExt -> BinQueueExt
meldPQ (p,s) (q,t) = (meld p q, unionSet s t)
```

Deleting the minimum item must also delete it from the set:

```
deleteMinPQ :: BinQueueExt -> BinQueueExt
deleteMinPQ (p,s)
  | not (isEmptySet s) = (q, delSet (entry i) s)
  where
    One (Node i q) = minTree p
```

The `findMinPQ` operation makes no change to the set and is just expressed in terms of `findMinPQ`. When decreasing the key for an item we simply insert the item into the queue with its new key. When deleting an item we delete it from the set.

```
decreaseKey :: Item -> BinQueueExt -> BinQueueExt
decreaseKey i pq | (entry i) 'elemSet' (snd pq) = insertPQ i pq
                 | otherwise                    = pq
```

```
delete :: Item -> BinQueueExt -> BinQueueExt
delete i (q,s) = (q, delSet (entry i) s)
```

Of course, maintaining a set has an impact on the time and space complexity of the priority queue operations. The set operations may be implemented with balanced trees for a reasonable complexity. The running times of `decreaseKey` and `delete` is $O(\log n)$. Both running times being dominated by the set operations. The other operations have the same worst-case complexity as before $O(\log n)$, except `meldPQ` which is now dominated by the complexity of the set union operation $O(n + m)$ (where n and m are the sizes of the two sets). Furthermore, because we never physically remove items from the queue the complexity of the operations is governed by the total number of inserts made. Constant factors may be improved by doing some garbage collection, that is, physically removing items that percolate to the roots of trees.

6 Conclusions

Binomial queues are both elegant and efficient. Our functional encoding is far more elegant than a typical imperative one (for instance the C code for meld is some four times larger). This is mainly because Haskell handles data structures like lists and trees very well. The implementation is also more elegant than other known efficient functional implementations of priority queues. It is hard to quantify elegance, but the code is smaller and I believe it's easier to understand and prove correct.

7 Acknowledgements

I am very grateful to the Workshop referees: Kieran Clenaghan, Simon Peyton Jones and Malcolm Wallace for their valuable comments. I am also grateful to the Glasgow FP group, especially those on the Workshop committee for their helpful feedback.

References

- [1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [2] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.
- [3] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal of Computing*, 7(3):298–319, 1978.
- [4] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [5] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.
- [6] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard Kieburtz, Rishiyur S. Nikhil, Will Par-tain, and John Peterson. Report on the functional programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.
- [7] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.
- [8] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Massachusetts, 1973.

- [9] L. C. Paulson. *ML for the working programmer*. Cambridge University Press, Cambridge, 1991.
- [10] Chris M. P. Reade. Balanced trees with removals: an exercise in rewriting and proof. *Science of Computer Programming*, 18:181–204, 1992.
- [11] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 235–245, Boston, Massachusetts, April 1983. ACM.
- [12] Robert E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.
- [13] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978.