International Journal Of Engineering Research & Management Technology ISSN: 2348-4039

September- 2014 Volume 1, Issue-5

Email: editor@ijermt.org www.ijermt.org

Optimal Purely Functional Priority Queues

Abhinav Sharma
Department of Information Technology
Dronacharya College of Engineering
Gurgaon,India

Mohit Kataria
Department of Information Technology
Dronacharya College of Engineering
Gurgaon,India

ABSTRACT

Brodal recently introduced the first implementation of imperative priority queues to support findMin, insert, and meld in O(1) worst-case time, and delete Min in O(log n) worst-case time. These bounds are asymptotically optimal among all comparison-based priority queues. In this paper, we adapt Brodal's data structure to a purely functional setting. In doing so, we both simplify the data structure and clarify its relationship to the binomial queues of Vuillemin, which support all four operations in O(log n) time. Specifically, we derive our implementation from binomial queues in three steps: first we reduce the running time of insert to O(1) by eliminating the possibility of cascading links; second, we reduce the running time of find Min to O(1) by adding a global root to hold the minimum element; and finally, we reduce the running time of meld to O(1) by allowing priority queues to contain other priority queues. Each of these steps is expressed using ML-style functors. The last transformation, known as data-structural bootstrapping, is an interesting application of higher-order functors and recursive structures.

1. INTRODUCTION

Purely functional data structures differ from imperative data struc- tures in at least two respects. First, many imperative data structures rely crucially on destructive assignments for efficiency, whereas purely functional data structures are forbidden from using destructive assign- ments. Second, purely functional data structures are automatically persistent (Driscoll et al., 1989), meaning that, after an update, both the new and old versions of a data structure are available for further accesses and updates. The design of efficient purely functional data structures is thus of great theoretical and practical interest to functional programmers, as well as to imperative programmers for those occasions when a persistent data structure is required. In this paper, we consider the design of an efficient purely functional priority queue. The priority queue is a fundamental abstraction in computer pro- gramming, arguably surpassed in importance only by the dictionary and the sequence. Many implementations of priority queues have been proposed over the years; a small sampling includes (Williams, 1964; Crane, 1972; Vuille min, 1978; Fredman & Tarjan, 1987; Brodal, 1996). However, all of these consider only imperative priority queues. Very little has been written about purely functional priority queues. To our knowledge, only Paulson (1991), Kaldewaij and Schoe makers (1991), Schoen makers (1992), and King (1994) have explicitly treated priority queues in a purely functional setting. We consider priority queues that support the following operations:

signature ORDERED = sig type T (∗ type of ordered elements ∗) val leq : T × T → bool (∗ total ordering relation ∗) end

signature PRIORITY QUEUE = sig structure Elem : ORDERED type T (∗ type of priority queues ∗) val empty : T val is Empty : T → bool val insert : Elem.T × T → T val meld : T × T → T exception EMPTY val find Min : T → Elem.T (∗ raises EMPTY if queue is empty ∗) val deleteMin : T → T (∗ raises EMPTY if queue is empty ∗) end

Figure 1: Signature for priority queues.

queue and a predicate is Empty. For simplicity, we will ignore empty queues except when presenting actual code. Figure 1 displays a Stan- dard ML signature for these priority queues. Brodal (1995) recently introduced the first imperative data struc- ture to support all these operations in O(1) worst-case time except deleteMin, which requires O(logn) worst-case time.

Second, we reduce the running time of find Min to O(1) by adding a global root to hold the minimum element. Third, we apply a tech- nique of Buchsbaum et al. (Buchsbaum et al., 1995; Buchsbaum & Tarjan, 1995) called data-structural bootstrapping, which reduces the running time of meld to O(1) by allowing priority queues to contain other priority queues. Each of these steps is expressed using ML-style functors. The last transformation, data-structural bootstrapping, is an interesting application of higher-order functors and recursive struc- tures. After describing a few possible optimizations, we conclude with brief discussions of related work and future work. All source code is presented in Standard ML (Milner et al., 1990) and is available through the World Wide Web from http://foxnet.cs.cmu.edu/people/cokasaki/priority.html

2. BINOMIAL QUEUES

Binomial queues are an elegant form of priority queue introduced by Vuillemin (1978) and extensively studied by Brown (1978). Although they considered binomial queues only in an imperative setting, King (1994) has shown that binomial queues work equally well in a func- tional setting. In this section, we briefly review binomial queues — see King (1994) for more details. Binomial queues are composed of more primitive objects known as binomial trees. Binomial trees are inductively defined as follows: • A binomial tree of rank 0 is a singleton node. • A binomial tree of rank r + 1 is formed by linking two binomial trees of rank r, making one tree the leftmost child of the other.

The binary representation of 21 is 10101, and the binomial queue contains trees of ranks 0, 2, and 4 (of sizes 1, 4, and 16, respectively). Note that a binomial queue of size n contains at most blog2(n + 1)ctrees. We are now ready to describe the operations on binomial queues. Since all the trees in a binomial queue are heap-ordered, we know that the minimum element in a binomial queue is the root of one of the trees. The analogy to binary addition also applies to melding two queues. Once again, each link corresponds to a carry. This also requires O(logn) time. Two aspects of this implementation de- serve further explanation. First, the conflicting requirements of insert and link lead to a confusing inconsistency, common to virtually all im-

functor Binomial Queue (E : ORDERED):PRIORITY QUEUE = struct structure Elem = E
type Rank = int data type Tree = Node of Elem.T × Rank × Tree list type T = Tree list (∗ auxiliary functions ∗) fun root (Node (x,r,c)) = x fun rank (Node (x,r,c)) = r fun link (t1 as Node (x1,r1,c1), t2 as Node (x2,r2,c2)) = (∗ r1 = r2 ∗) if Elem.leq (x1, x2) then Node (x1,r1+1,t2 :: c1) elseNode (x2,r2+1,t1 :: c2) fun ins (t, [ ]) = [t] |ins (t, t0 :: ts) = (∗rank t ≤ rank t0 ∗) if rank t<rank t0 then t :: t0 :: ts else ins (link (t, t0), ts)
val empty =[] fun is Empty ts = null ts
fun insert (x, ts)=ins (Node (x,0,[ ]), ts) fun meld ([ ], ts)=ts | meld (ts, [ ]) =ts | meld (t1 :: ts1, t2 :: ts2)= if rank t1 < rank t2 then t1 :: meld (ts1, t2 :: ts2) else if rank t2 < rank t1 then t2 :: meld (t1 :: ts1, ts2) else ins (link (t1, t2), meld (ts1, ts2))

Exception EMPTY
fun find Min [ ] = raiseEMPTY | find Min [t]=root t | findMin (t :: ts)= let val x = findMin ts in if Elem.leq (root t, x) then root t else x end fun deleteMin [ ] = raiseEMPTY | deleteMin ts = let fun getMin [t]=(t, [ ]) |getMin (t :: ts)= let val (t0, ts0)=getMin ts in if Elem.leq (root t, root t0) then (t ,ts) else (t 0,t:: ts0) end val (Node (x,r,c), ts)=getMin ts in meld (rev c, ts) end end

Figure 3: A functor implementing binomial queues.

plementations of binomial queues. The trees in binomial queues are maintained in increasing order of rank to support the insert opera- tion efficiently. On the other hand, the children of binomial trees are maintained in decreasing order of rank to support the link operation efficiently. The ranks of all other nodes are uniquely determined by the ranks of their parents and their positions among their siblings. King (1994) describes an alternative representation that eliminates all ranks, at the cost of introducing placeholders for those ranks corresponding to the zeros in the binary representation of the size of the queue.

3. SKEW BINOMIAL QUEUES

In this section, we describe a variant of binomial queues, called skew binomial queues, that supports insertion in O(1) worst-case time. The problem with binomial queues is that inserting a single element into a queue might result in a long cascade of links, just as adding one to a binary number might result in a long cascade of carries. We can reduce the cost of an insert to at most a single link by borrowing a technique from random-access lists (Okasaki, 1995b). Just as binomial queues are composed of binomial trees, skew bino- mial queues are composed of skew binomial trees. Skew binomial trees are inductively defined as follows: • A skew binomial tree of rank 0 is a singleton node. 8

Figure 4 illustrates the three kinds of links. Note that type A and type B skew links are equivalent when r = 0. Once again, there is a second, equivalent definition: a skew binomial tree of rank r>0 is a node with up to 2k children s1t1 ...sktk ($1 \leq k \leq r$), where each ti is a skew binomial tree of rank r−i and each si is a skew binomial tree of rank 0, except that sk has rank r −k (which is 0 only when k = r). Every si is optional except that sk is optional only when k = r. Although somewhat confusing, this definition arises naturally from the three methods of constructing a tree. Every sktk pair is produced by a type A skew link, and ever

Figure 5: The twelve possible shapes of skew binomial trees of rank 2. Dashed boxes surround each siti pair.
siti pair (i<k) is produced by a type B skew link. Every ti without a corresponding si is produced by a simple link. Unlike ordinary bino- mial trees, skew binomial trees may have many different shapes. For example, the twelve possible shapes of skew binomial trees of rank 2 are shown in Figure 5. Since skew binomial trees of the same rank may have different sizes, there may be several ways to distribute the ranks for a queue of any particular size. For example, a skew binomial queue of size 4 may contain one rank 2 tree of size 4; two rank 1 trees, each of size 2; a rank 1 tree of size 3 and a rank 0 tree; or a rank 1 tree of size 2 and two rank 0 trees. However, the maximum number of trees in a queue is still O(logn). We are now ready to describe the operations on skew binomial queues. The findMin and meld operations are almost unchanged. To find the minimum element in a skew binomial queue, we simply scan through the roots, taking O(logn) time. To meld two queues, we step through the trees of both queues in increasing order of rank, perform- ing a simple link (not a skew link!) whenever we find two trees of equal rank. Once again, this requires O(logn) time. Each of these steps requires O(logn) time, so the total time required is O(logn). Figures 6 and 7 present an implementation of skew binomial queues as a Standard ML functor. Like the binomial queue functor, this func- tor takes a structure specifying a type of ordered elements and produces a structure of priority queues containing elements of the specified type. Once again, lists of trees are maintained in different orders for different purposes. The trees in a queue are maintained in increasing order of rank (except that the first two trees may have the same rank), but the children of skew binomial trees are maintained in a more complicated

fun uniqify []=[] |uniqify (t :: ts)=ins (t, ts)(∗eliminate initial duplicate ∗) fun meldUniq ([ ], ts)=ts | meldUniq (ts, [ ]) =ts | meldUniq (t1 :: ts1, t2 :: ts2)= if rank t1 < rank t2 then t1 :: meldUniq (ts1, t2 :: ts2) else if rank t2 < rank t1 then t2 :: meldUniq (t1 :: ts1, ts2) else ins (link (t1, t2), meldUniq (ts1, ts2))
val empty =[] fun isEmpty ts = null ts

Figure 6: A functor implementing skew binomial queues (part I).

fun insert (x, ts as t1 :: t2 :: rest)= if rank t1 = rank t2 then skewLink (Node (x,0,[ ]),t1,t2) ::rest else Node (x,0,[ ]) :: ts | insert (x, ts)=Node (x,0,[ ]) :: ts fun meld (ts, ts0)=meldUniq (uniqify ts, uniqify ts0)exception EMPTY fun findMin [ ] = raiseEMPTY | findMin [t]=root t | findMin (t :: ts)= let val x = findMin ts in if Elem.leq (root t, x) thenroot t else x end fun deleteMin [ ] = raiseEMPTY | deleteMin ts = let fun getMin [t]=(t, [ ]) |getMin (t :: ts)= let val (t0, ts0)=getMin ts in if Elem.leq (root t, root t0) then (t ,ts) else (t 0,t:: ts0) end fun split (ts,xs,[ ]) = (ts, xs) | split (ts,xs,t :: c)= if rank t = 0 thensplit (ts,root t :: xs,c) elsesplit (t :: ts,xs,c) val (Node (x,r,c), ts)=getMin ts val (ts0,xs0)=split ([ ],[ ],c) in fold insert xs0 (meld (ts, ts0)) end order. The ti children are maintained in decreasing order of rank, but they are interleaved with the si children, which have rank 0 (except sk, which has rank r−k). Furthermore, recall that each si is optional (except that sk is optional only if k = r).

4. ADDING A GLOBAL ROOT

We next describe a simple module-level transformation on priority queues to reduce the running time of findMin to O(1). Although this transformation can be applied to any priority queue module, it is only useful on priority queues for which findMin requires more than O(1) time. Most implementations of priority queues represent a queue as a single heap-ordered tree so that the minimum element can always be found at the root in O(1) time. Unfortunately, binomial queues and skew binomial queues represent a queue as a forest of heap-ordered trees, so finding the minimum element requires scanning all the roots in the forest. However, we can convert this forest into a single heap-ordered tree, thereby supporting findMin in O(1) time, by simply adding a global root to hold the minimum element. In general, this tree will not be a binomial or skew binomial tree, but this is irrelevant since the global root will be treated separately from the rest of the queue. The details of this transformation are quite routine, but we present them anyway as a warm-up for the more complicated transformation in the next section. Given some type Pα of primitive priority queues containing elements of type α, we define the type of rooted priority queues RPα to be RPα = {empty}+(α×Pα) In other words, a rooted priority queue is either empty or a pair of a single element (the root) and a primitive priority queue. We maintain the invariant that the minimum element of any non-empty priority queue is at the root. For each operation f on priority queues, let functor AddRoot (Q : PRIORITY QUEUE):PRIORITY QUEUE = struct structure Elem = Q.Elem datatype T = Empty | Root of Elem.T × Q.T val empty = Empty fun isEmpty Empty = true | isEmpty (Root) = false fun insert (y, Empty)=Root (y, Q.empty) | insert (y, Root (x, q)) = if Elem.leq (y, x) thenRoot (y, Q.insert (x, q)) else Root (x, Q.insert (y, q)) fun meld (Empty, rq)=rq | meld (rq, Empty)=rq | meld (Root (x1, q1), Root (x2, q2)) = if Elem.leq (x1, x2) thenRoot (x1, Q.insert (x2, Q.meld (q1, q2))) else Root (x2, Q.insert (x1, Q.meld (q1, q2)))

Figure 8: A functor for adding a global root to existing priority queues.

and f0 indicate the operations on Pα and RPα, respectively. Then, findMin0(hx,qi)= x insert0 (y,hx,qi)= h x,insert (y,q)i if x ≤ y insert0 (y,hx,qi)= h y,insert (x,q)i if y<x meld0 (hx1,q1i,hx2,q2i)=h x 1 , insert (x2,meld (q1,q2))i if x1 ≤ x2 meld0 (hx1,q1i,hx2,q2i)=h x 2 , insert (x1,meld (q1,q2))i if x2 <x 1 deleteMin0(hx,qi)= h findMin (q),deleteMin (q)i In Figure 8, we present this transformation as a Standard ML functor that takes a priority queue structure and produces a new structure incorporating this optimization. When applied to the skew binomial queues of the previous section, this tranformation produces a priorityqueue that supports both insert and findMin in O(1) time. However, meld and deleteMin still require O(logn) time. If a program requires several priority queues with different element types, it may be more convenient to implement this transformation as a higher-order functor (MacQueen & Tofte, 1994). First-order functors can only take and return structures, but higher-order functors can take and return other functors as well. Although the definition of Standard ML (Milner et al., 1990) describes only first-order functors, some implementations of Standard ML, notably Standard ML of New Jersey, support higher-order functors. A priority queue functor, such as BinomialQueue or SkewBinomi- alQueue, is one that takes a structure specifying a type of ordered ele- ments and returns a structure of priority queues containing

elements of the specified type. The following higher-order functor takes a priority queue functor and returns a priority queue functor incorporating the AddRoot optimization.

Note that this functor is curried, so although it appears to take two arguments, it actually takes one argument (MakeQ) and returns a functor that takes the second argument (E). The sharing constraint is necessary to ensure that the functor Make Q returns a priority queue with the desired element type. Without the sharing constraint, MakeQ might ignore E and return a priority queue structure with some arbi- trary element type. Now, if we need both a string priority queue and an integer priority queue, we can write

functor Rooted Skew Binomial Queue = AddRootToFun (functor Make Q = Skew Binomial Queue) structure StringQueue = Rooted Skew Binomial Queue (String Elem) structure Int Queue = Rooted Skew Binomial Queue (IntElem)

where StringElem and IntElem match the ORDERED signature and define the desired orderings over strings and integers, respectively.

5. OPTIMIZATIONS

Although bootstrapped skew binomial queues as described in the pre- vious section are asymptotically optimal, there are still further opti- mizations we can make. Consider the type of priority queues resulting from inlining Skew Binomial Queue for Make Q: data type Tree = Node of Root × Rank × Tree list and Root = Root of Elem.T × Tree list datatype T = Empty | NonEmpty of Root In this representation, a node has the formNode(Root(x,f),r,c), where x is an element, f is a list of trees representing a forest, r is a rank, and c is a list of trees representing the children of the node. Since every node contains both x and f we can flatten the representation of nodes to be data type Tree = Node of Elem. T × Tree list × Rank × Tree list In many implementations, this will eliminate an indirection on every access to x. Next, note that f is completely ignored until its root is deleted. Thus, we do not require direct access to f and can in fact store it at the tail of c, combining the two into a single list representing c ++ f. If r = 1, then c consists of either one or two rank 0 nodes. If r>1, then c ends with either a pair of nodes of the same non-zero rank or a rank 1 node followed by one or two rank 0 nodes. The only ambiguities involve rank 0 nodes: it is sometimes impossible to distinguish the case where c ends with two rank 0 nodes from the case where c ends with a single rank 0 node and f begins with a rank

6. RELATED WORKS

Although there is an enormous literature on imperative priority queues, there has been very little work on purely functional priority queues. Paulson (1991) describes a (non-meldable) priority queue combining the techniques of implicit heaps (Williams, 1964), which traditionally are implemented using arrays, with a balanced-tree representation of arrays supporting extension at the rear. functional languages support binomial queues quite elegantly. Schoenmakers (1992), extending earlier work with Kaldewaij (1991), uses functional notation to aid in the derivation of amortized bounds for a number of data structures, including three priority queues: skew heaps1 (Sleator & Tarjan, 1986), Fibonacci heaps (Fredman & Tarjan, 1987), and pairing heaps (Fredman et al., 1986). Schoenmakers also discusses splay trees (Sleator & Tarjan, 1985), a form of self-adjusting

binary search tree that has been shown by Jones (1986) to be particu- larly effective as a non-meldable priority queue. Each of these four data structures is efficient only in the amortized sense. Although he uses functional notation, Schoenmakers restricts his attention to ephemeral uses of data structures, where only the most recent version of a data structure may be accessed or updated. Ephemerality is closely related to the notion of linearity (Wadler, 1990). Finally, our data structure borrows techniques from several sources. Skew linking is borrowed from the random-access lists of Okasaki (1995b), which in turn are a modification of the random-access stacks of Myers (1983). (Buchsbaum et al., 1995; Buchsbaum & Tarjan, 1995) to sup- port catenation for double-ended queues, much as we use it to support melding for priority queues.

See Okasaki (1995a; 1996) for a fuller discussion of the interaction between lazy evaluation and amortization. Next, we note that imperative priority queues often support two ad- ditional operations, decreaseKey and delete, that decrease and delete a specified element of the queue, respectively.

ACKNOWLEDGMENTS

Thanks to Yashvardhan Soni for their comments and suggestions on an earlier draft of this paper.

REFERENCES

1. Brodal, Gerth Stølting. (1995). Fast meldable priority queues. Pages 282–290 of: Workshop on Algorithms and Data Structures. LNCS, vol. 955. Springer-Verlag.
2. Brodal, Gerth Stølting. 1996 (Jan.). Worst-case priority queues. Pages 52–58 of: ACM-SIAM Symposium on Discrete Algorithms.
3. Brown, Mark R. (1978). Implementation and analysis of binomial queue algorithms. SIAM Journal on Computing, 7(3), 298–319.
4. Buchsbaum, Adam L., & Tarjan, Robert E. (1995). Confluently persistent deques via data structural bootstrapping. Journal of Algorithms, 18(3), 513–547.
5. Buchsbaum, Adam L., Sundar, Rajamani, & Tarjan, Robert E. (1995). Data-structural bootstrapping, linear path compression, and catenable heap-ordered double-ended queues. SIAM Journal on Computing, 24(6), 1190–1206
6. .Crane, Clark Allan. 1972 (Feb.). Linear lists and priority queues as balanced binary trees. Ph.D. thesis, Computer Science Department, Stanford University. Available as STAN-CS-72-259.