EFFICIENT SPECIAL PURPOSE PRIORITY QUEUES

DONALD B. JOHNSON
Computer Science Department
The Pennsylvania State University
University Park, Pennsylvania 16802

ABSTRACT

Priority queues, as usually implemented, require $O(\log n)$ time for each INSERT and MIN operation when a sequence containing n INSERT operations is executed. It is shown how to reduce the average operation cost to $o(\log n)$ when there is a large number of UPDATE operations and to $o(\log s)$ for certain sequences when the domain of integer priorities is of size s. Data structures and application to routing and connection problems are discussed.

PRIORITY QUEUES

A priority queue with operations INSERT and MIN can be implemented so each operation will require $O(\log n)$ time, where there are at most n elements in the queue.† A standard method among the numerous methods which have been proposed [1,2,3,8,9] stores elements in a heap, a binary tree in which each path from the root is nondecreasing in priority. Heap order can be maintained under both INSERT and MIN by adjustments which are confined to the vicinity of a path in the tree of length $\lfloor \log n \rfloor$. If the domain of priorities is an arbitrary set with a total order, a running time of $o(\log n)$ cannot be achieved simultaneously for both operations since such a result would give an algorithm for comparison sorting which runs in $o(n \log n)$ time. Tradeoffs are easily obtained, however, in which either INSERT or MIN requires $O(n)$ time. If the queue is stored as an unordered list, then INSERT takes constant time and MIN requires $O(n)$ time. If the queue is stored as a sorted list then MIN takes constant time while INSERT is $O(n)$. More refined tradeoffs can be had with a repetoire of operations which includes UPDATE, the change of the priority of a queue element for the better [7]. The number of UPDATEs may dominate the other operations in certain algorithms with embedded priority queues, for example, algorithms for finding optimum spanning trees [7] and shortest paths [6]. For such problems a priority queue in which UPDATE and INSERT are asymptotically less costly than MIN can be desirable. We call such a priority queue an unbalanced priority queue. In contrast, the conventional queues in which INSERT and MIN are of equal asymptotic cost are called balanced.

It may be that the domain of priorities is restricted to some set of integers. In the case where the priorities are drawn from the set $\{1,\ldots,s\}$ the queue operations mentioned can be implemented to run in $O(\log\log s)$ time [10,11]. This result is achieved on a RAM under the uniform cost criterion (see [1]). The rather intricate data structure used is initialized at a cost of $O(s)$ which, when distributed over a sequence of $O(s)$ queue operations, does not dominate the sum of the individual operation costs. The main result in this paper is obtained on sequences of queue operations in which MIN returns nondecreasing values. Such operation sequences characterize the applications mentioned above. We improve the asymptotic running times obtained in [10,11] in cases where intialization dominates because the priority domain is large compared to the length

---

†Unless specified otherwise, log is $\log_2$ and ln is the natural logarithm.

of the operation sequence. Our queues (which are unbalanced) are simpler and use less space, a significant consideration when the domain of priorities is large compared with the length of the sequence of queue operations.

In the following sections we consider priority queues which support the operations defined below, where Q is a priority queue, x is the name of a queue element, and p is a priority drawn from a set P with a total order.

$$\begin{aligned}
\text{INIT}(Q) \;::=\;& Q \leftarrow \emptyset \\
\text{INSERT}(p,Q) \;::=\;& \text{let } x \notin Q \\
& \pi(x) \leftarrow p \\
& Q \leftarrow Q \cup \{x\} \\
& \underline{\text{return}}(x) \\
\text{DELETE}(x,Q) \;::=\;& Q \leftarrow Q - \{x\} \\
\text{MIN}(Q) \;::=\;& \text{let } x \text{ satisfy } \pi(x) = \min_{y \in Q} (\pi(y)) \\
& \text{DELETE}(x,Q) \\
& \underline{\text{return}}(x,\pi(x))
\end{aligned}$$

$$\text{UPDATE}(x,p,Q) \;::=\; \underline{\text{if}}\ p < \pi(x)\ \ \underline{\text{then}}\ \ \pi(x) \leftarrow p\ .$$

Let $\sigma(Q)$ be a sequence of the above operations, on a given priority queue Q, which contains exactly one INIT(Q) operation (at the beginning). The sequence $\sigma(Q)$ will contain n INSERT operations, u UPDATE operations, and m DELETE and MIN operations taken together. Thus the length of $\sigma(Q)$, denoted $|\sigma(Q)|$, is $1 + n + u + m$. We are concerned with the cost of $\sigma(Q)$ on a uniform cost RAM as a function of the variables n, u, and m. Our discussion will be confined to sequences for which $m \leq n$. For the purposes of asymptotic analysis of the cost of $\sigma(Q)$, families of sequences must be considered in which fixed functional relationships between m, n, and u are preserved as, say, n grows. When considered in the context of a family of sequences for which $n + u = O(m)$, a sequence will be called balanced. Otherwise, if $n + u = O(m)$ does not hold for the family of sequences, a member sequence will be called unbalanced.

UNBALANCED QUEUES FOR ARBITRARY PRIORITY DOMAINS

In all of the balanced queues cited in the preceding section, the data structure used is some variant on a binary tree. Perhaps ternary nodes are permitted in the tree and, perhaps, queue elements appear only at leaf nodes instead of at every node in the tree. The 2-3 trees of [1], for example, employ both of these variations. Trees with nodes of fixed, small degree are asymptotically optimal for balanced operation sequences. The β-heap of [7] is a full tree of degree $\beta \geq 2$, that is, a heap-ordered tree in which every node except possibly one is of degree 0 or β and all nodes of degree less than β are on the two lowest levels of the tree. In a β-heap, restoring heap order in DELETE and MIN requires movement of an element toward the leaves of the tree, so these operations may require as many as $(\beta - 1)\lfloor \log_\beta n \rfloor$ comparisons. The operations INSERT and UPDATE move elements rootward, requiring at most $\lfloor \log_\beta n \rfloor$ comparisons. Initiation of a β-heap is of constant cost and requires no comparisons. Let the number of comparisons executed on $\sigma(Q)$ be $C(\sigma(Q))$. We conclude from the above discussion that, for a β-heap,

$$C(\sigma(Q)) \leq ((\beta - 1)m + n + u)\log_\beta n\ .$$

This bound is attainable, to within a uniform constant factor, for all allowable values of m, n, and u. If we let $\beta = \lfloor n^{1/r} \rfloor$ for $r \geq 1$ then the bound becomes

$$C(\sigma(Q)) \leq r(m(n^{1/r} - 1) + n + u) . \qquad (1)$$

It should be noticed that the bound in (1) reduces to $(m + n + u)\log n$ for $r = \log n$, that is, for the balanced queue with $\beta = 2$. For sufficiently unbalanced sequences $\sigma(Q)$ other choices for r are better. For instance, for fixed $\epsilon > 0$, if $m = n > 2^{1.44/\epsilon}$ and $u = n^{1+\epsilon}$ then as n grows the r which optimizes (1) approaches

$$r = \frac{1}{\epsilon - \frac{\log\log n}{\log n}} .$$

For $n \leq 2^{1.44/\epsilon}$ choosing $r = \log n$ is near to the optimal choice. Priority queues with $m = n$ arise in computing minimum spanning trees and shortest paths in graphs. If the graphs in question have at least $n^{1+\epsilon}$ edges for some fixed ϵ, where n is the number of vertices, the above discussion may be applied (see [6] and [7]). It may be seen that unbalanced queues will be of value in practical cases for values of ϵ no smaller than, say, 1/10. When $\epsilon = 1/5$, for $n = 2^{16}$ the optimal choice of r in (1) gives about one half the number of comparisons obtained when $r = \log n = 16$. If $\epsilon = 1/8$, the improvement is roughly a factor of 2/3 for $n = 2^{16}$.

When $r = \log\log n$, the bound in (1) becomes

$$C(\sigma(Q)) \leq (\log\log n)(m(n^{1/\log\log n} - 1) + n + u) .$$

We notice that the cost of INSERT and UPDATE is of the order of the bound (measured in terms of domain size s) achieved in the best known results for queues over limited priority domains. This improvement, however, is obtained at a cost for MIN and DELETE which grows considerably faster than $\log n$.

We close this section by considering a recursive construction in which the β offspring of each non-leaf node in a β-heap with $\beta = \lfloor n^{1/k} \rfloor$ for fixed k are themselves organized as a γ-heap with $\gamma = \lfloor \beta^{1/k} \rfloor$ whenever $\beta > 2$. The number of comparisons $C(n)$ required for a single DELETE or MIN satisfies the recurrence

$$C(n) \leq \begin{cases} 1 & n \leq 2 \\ kC(n^{1/k}) & n > 2 , \end{cases}$$

the solution to which is $C(n) \leq \log n$. However, INSERT and UPDATE become more costly. In fact, the number of comparisons for each of these operations may be seen to be described by the same recurrence, so there appears to be no advantage to this recursive structure. The trees induced by this recursive construction, however, are not binary. They contain a few nodes of large degree.

QUEUES ON THE PRIORITY DOMAIN {1,...,s}

When s, the size of the restricted priority domain P, is large compared to $n + m + u$, the results of the previous section will be

superior to the methods in this section or in [10,11]. However, when s is relatively small it can be advantageous to use a priority queue in which the cost of individual queue operations grows with s rather than n. The algorithm of this section is restricted to nondecreasing sequences $\sigma(Q)$, those for which if some instance of MIN returns the priority q then $q \leq p$ for any operation INSERT(p) which follows this instance of MIN in σ. (We will throughout the remaining discussion omit the argument Q when a certain queue is understood.)

Given the domain $P = \{1,\ldots,s\}$, our unbalanced queue employs a sequence of at most $\lfloor \log s \rfloor + 2$ buckets with indices i, $0 \leq left \leq i \leq right = \lfloor \log s \rfloor + 1$, where $left$ and $right$ are variables which define the range of active buckets. Bucket i has a size equal to 2^j for some j satisfying $0 \leq j < i$ and for all i, $left < i < right$, bucket i is smaller than bucket $i + 1$. Taken together, the buckets cover the set $\{q,\ldots,s\}$ where q is either the value of the last priority returned by a MIN operation or one if no MIN has yet been executed.† The operations INSERT and UPDATE are performed by a binary search over the buckets and so cost $O(\log\log s)$. MIN requires a search of buckets $left$, $left + 1,\ldots$ until a nonempty bucket i is discovered. Bucket i, of size 2^j, is replaced in the execution of MIN by a subset of the sequence of buckets of sizes $1,1,2,4,\ldots,2^{j-1}$ chosen so that the smallest priority in bucket i will fall into one of the new buckets of size one. The elements from bucket i are placed, using binary search, into the appropriate new bucket. MIN then completes by deleting and returning an element of smallest priority from the leftmost bucket of size one.

We will analyze the complexity over the sequence σ. Each MIN will require $O(\log s)$ time to find the nonempty bucket i plus $O(\log s)$ to construct the bucket refinement, knowing the minimum element in the original bucket i. Finding the minimum element plus restributing the elements from bucket i to the new buckets will cost $O(n \log s \log\log s)$ over σ since each of the n elements ever to enter the queue is always moved to a bucket no larger than half the size of the bucket it occupied before. For buckets of size one, finding the minimum element is of constant cost. Of course, a bucket of size one will not be refined.

The procedures described are implemented below, in order to verify the following theorem.

THEOREM. A priority queue can be constructed over which the cost of all nondecreasing sequences is $O(n \log s \log\log s + u \log\log s)$ where a fixed functional relationship is assumed between n and u.

Proof: The complexity argument is outlined in the preceding discussion. We note that no multiplications or divisions are required by our algorithms if a table of the values 2^i for $0 \leq i \leq \lfloor \log s + 1 \rfloor$ are precomputed at a cost of $O(\log s)$. Thus our analysis applies on a RAM under the uniform cost criterion. We omit correctness arguments. These can be constructed easily from the discussion given above. □

COROLLARY. Minimum spanning trees and all shortest paths from a single source in undirected graphs with n vertices, e edges, and edge weights drawn from $\{1,\ldots,s\}$ can be found in $O(n \log s \log\log s + e \log\log s)$ time.

†$\{q,\ldots,q + s - 1\}$ can easily be covered at no increase in complexity.

Proof: The result follows directly from the Theorem. See [6,7] for details. □

An interesting method for finding shortest paths with running time of $O(nks^{1/k} + e(\log k + 1))$ has been given in [4]. As may be seen, this bound is superior for fixed k when s is small and e is large. These are the conditions, however, when the results of the previous section might be superior to either method. The bound in [4] reduces to the bound for the algorithm described in [5] and [12] when $k = 1$ and to our bound in the Corollary when k is replaced by $\log s$.

In the procedures below, a facility is assumed which supplies new queue nodes x with link fields with which to chain nodes together in buckets, a field bucket(x) which contains the index of the bucket to which x belongs, and $\pi(x)$, the priority of x. The array ℓ contains bucket boundaries; $\ell(i)$ is the least integer which can be contained in bucket i. The array c is maintained so that $c(i)$ is the number of nodes in bucket i.

```
procedure INIT
   left ← min {i + 1 | 2^i ≥ s}
   right ← left
   ℓ(left) ← 1
   ℓ(left + 1) ← 2^(left-1) + 1
   for i ← 0 to left + 1 do  c(i) ← 0  end
end INIT

procedure PLACE(x,i,j)
   let k , i ≤ k ≤ j , satisfy  ℓ(k) ≤ π(x) < ℓ(k + 1)
   put node x in bucket k
   bucket(x) ← k
   c(k) ← c(k) + 1
end PLACE

procedure INSERT(p)
   let x be an unused node
   π(x) ← p
   PLACE(x,left,right)
   return(x)
end INSERT

procedure DELETE(x)
   remove node x from bucket bucket(x)
   c(bucket(x)) ← c(bucket(x)) - 1
   bucket(x) ← -1
end DELETE

procedure UPDATE(x,p)
   if left ≤ bucket(x) ≤ right  and  p < π(x)  then
      [DELETE(x)
      π(x) ← p
      PLACE(x,left,right)]
end UPDATE
```

```
procedure MIN
   while  c(left) = 0  do
      if  left > right  then  [left ← left - 1
                              return(Λ,0)]
      left ← left + 1
   end
   if  ℓ(left + 1) - ℓ(left) > 1  then
      [let  x  minimize  π(x)  in bucket left
      span ← π(left + 1) - π(x)
      let  t  satisfy  ℓ(left + 1) - ℓ(left) = 2^(t+1)
      i ← left
      while  span > 1  do
         if  span - 2^t > 0  then
            [span ← span - 2^t
            ℓ(left) ← ℓ(left + 1) - 2^t
            left ← left - 1]
         t ← t - 1
      end
      c(i) ← 0
      ℓ(left) ← ℓ(left + 1) - 1
      move nodes in bucket  i  to a dummy bucket
      for  each  x  in the dummy bucket  do
         PLACE(x,left,i)
      end]
   let  x ∈ bucket left
   DELETE(x)
   return(x,π(x))
end MIN
```

CONCLUSION

It has been shown that unbalanced queues can be more efficient than balanced queues over unbalanced operation sequences, those in which the number of MIN operations is dominated by the number of INSERT and UPDATE operations. In an unbalanced queue over an arbitrary priority domain, the cost of INSERT and UPDATE grows more slowly with n than the cost of MIN. When operation sequences are known to be nondecreasing in the values returned by MIN then there are unbalanced queues in which operation costs grow with s, the size of the domain of priorities (assumed to be a set of consecutive integers). When n is large in relation to s, these queues are superior.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

2. M. R. Brown, The analysis of a practical and nearly optimal priority queue, Tech. Rep. STAN-CS-77-600, Computer Science Department, Stanford University, 1977.

3. C. A. Crane, Linear lists and priority queues as balanced binary trees, Ph.D. Thesis, Computer Science Department, Stanford University (STAN-CS-72-259), February 1972.

4. E. V. Denardo and B. L. Fox, Shortest route methods: 1. Reaching, pruning, and buckets. (Preliminary version), (unpublished), May 1977.

5. R. B. Dial, Algorithm 360: Shortest-path forest with topological ordering, Comm. ACM 12, 11(Nov. 1969), 632-633.

6. D. B. Johnson, Efficient algorithms for shortest paths in sparse networks, J. ACM 24, 1(Jan. 1977), 1-13.

7. D. B. Johnson, Priority queues with update and finding minimum spanning trees, Information Processing Letters 4, 3(Dec. 1975), 53-57.

8. A. Jonassen and O-J. Dahl, Analysis of an algorithm for priority queue administration, BIT 15 (1975), 409-422.

9. D. E. Knuth, The Art of Computer Programming, Volume 3/Sorting and Searching, Addison-Wesley, 1973.

10. P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Mathematical Systems Theory 10 (1977), 99-127.

11. P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Information Processing Letters 6, 3(June 1977), 80-82.

12. R. A. Wagner, A shortest path algorithm for edge-sparse graphs, J. ACM 23, 1(Jan. 1976), 50-57.