Research Report 221

Fishspear: A Priority Queue Algorithm

Fischer MJ and Paterson MS

RR221

The Fishspear priority queue algorithm is presented and analyzed. Fishspear is comparable to the usual heap algorithm in its worst case running time, and its relative performance is much better in many common situations. Fishspear also differs from the heap method in that it can be implemented efficiently using sequential storage such as stacks or tapes, making it potentially attractive for implementation of very large queues on paged memory systems.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

June 1992

Fishspear: A Priority Queue Algorithm[†]

Michael J. Fischer
Department of Computer Science
Yale University
New Haven, Connecticut, USA

Michael S. Paterson
Department of Computer Science
University of Warwick
Coventry, England

Abstract

The Fishspear priority queue algorithm is presented and analyzed. Fishspear is comparable to the usual heap algorithm in its worst case running time, and its relative performance is much better in many common situations. Fishspear also differs from the heap method in that it can be implemented efficiently using sequential storage such as stacks or tapes, making it potentially attractive for implementation of very large queues on paged memory systems.

1 Introduction

A *priority queue* is an abstract data type consisting of a finite set P over a universe U of *elements*. With each element $x \in U$ is associated a value, key(x), from a totally ordered domain $(D, <)$. The total order on keys induces a partial order on the elements, also denoted by $<$. The priority queue has the following operations:

EMPTY?: Returns true if $P = \emptyset$, false otherwise.

INSERT(x): Sets $P := P \cup \{x\}$.

DELETE_MIN: Sets $P := P - \{y\}$ and returns y, where y is a minimal element in P.

(We omit a MAKE_EMPTY operation for setting $P := \emptyset$, since it can be easily simulated using EMPTY? and DELETE_MIN.) We assume that the elements inserted (though not necessarily their values) are all distinct. Priority queues find application in discrete event simulation, computational geometry, shortest path computations, and many other areas of computer science.

[†]This work was supported in part by the Office of Naval Research under Contracts N00014–82–K–0154 and N00014–89–J–1980, by the National Science Foundation under Grants MCS–8116678, MCS–8405478, and IRI–9015570, by the ESPRIT BRA Program of the EC under contracts 3075 and 7141 (ALCOM, ALCOM II), and by a Senior Fellowship of the SERC. A preliminary version of this paper appeared in *Proc. 25th IEEE Symp. on Foundations of Computer Science* (1984), 375–386.

A simple implementation of priority queues keeps the elements in an ordered list. An insertion is performed by binary search using $\lceil \log h \rceil$ comparisons when it yields a list of size h, and the remaining operations take no comparisons.[1] However, the *time* per insertion is $\Omega(h)$, making the algorithm unattractive in practice for all but very small queues.

The *heap* [1] is a standard data structure for implementing priority queues which, like the ordered list, uses $O(\log h)$ comparisons per operation, but the time per operation is linear in the number of comparisons and so is also $O(\log h)$. Indeed, heaps are so common as to be often identified with the abstract data type which they implement. So that there is no confusion, by a "heap" we mean a balanced binary tree with elements x_i labeling each node i such that for any nodes i, j, if i is an ancestor of j, then $\text{key}(x_i) \leq \text{key}(x_j)$.

One of the first applications of heaps was to an algorithm for sorting n items using $O(n \log n)$ comparisons [8]. Since $\Omega(n \log n)$ is a lower bound on the number of comparisons for sorting, it follows that the amortized cost[2] of a priority queue operation is $\Omega(\log n)$ in the worst case, where n is the length of the operation sequence. Since heaps achieve this bound, they are in some sense optimal.

A significant property of heaps is that they exploit the ability to access memory randomly. The pattern of memory accesses is dynamically determined by the data, and there is no apparent way of maintaining the logarithmic amortized operation cost when implementing heaps on more restrictive types of memory such as tapes or stacks.

Other data structures such as 2-3 trees, etc., can also implement priority queues with similar complexity bounds, but all require random-access storage. Thus, priority queues have seemed to be an example of an abstract data type whose efficient implementation required random-access storage, and heaps are a simple implementation which seemed optimal.

In this paper, we show that both intuitions are wrong by presenting a new priority queue algorithm, *Fishspear*, which can be implemented with *sequential* storage (using a fixed number of pushdown stacks), and which is *more efficient* than a heap in a sense that is made precise in the next section. Although it has similar amortized efficiency to a heap in the worst case ($O(\log n)$ comparisons per queue operation), the number of comparisons is "little-oh" of the number made by a heap for many classes of input sequences that are likely to occur in practice. For example, if the queue builds to a certain size h and then receives alternately a very large number of INSERT and DELETE_MIN operations, where the keys of the elements to be inserted are drawn randomly with uniform distribution from the unit interval, then the amortized number of comparisons made by a heap for each such pair is $\Omega(\log h)$, whereas the amortized cost for Fishspear is $O(1)$. (The queue at any time during this procedure contains the h largest elements ever inserted, so the value of the smallest of these approaches 1. The probability that a newly-inserted element will very soon be deleted becomes large, and Fishspear is particularly efficient in such a situation.)

More generally, the number of comparisons required by Fishspear depends only on the size of the "active" part of the queue, not on the overall size. In the above example, the active part shrinks over time as the queue fills with larger and larger elements. This notion is quantified more precisely in the next section.

[1] Logarithms are taken to the base 2 unless specified otherwise.

[2] The *amortized* cost of a sequence of operations is the total cost of the sequence divided by the number of operations [5, 6].

Fishspear can be implemented using sequential storage such as tapes or stacks so that the overall run time is proportional to the total number of comparisons. Sequential storage algorithms such as Fishspear are attractive on typical paged computer systems since they tend to exhibit better paging performance than random-access algorithms such as heaps. This, together with the better behavior on common but restricted classes of operation sequences, could make Fishspear an attractive alternative to heaps in certain practical situations.

The principal disadvantages of Fishspear are that it is more complicated to implement than a heap, and the overhead per comparison may be greater.

Fishspear is similar to self-adjusting heaps [6, 7] in that the behavior depends dynamically on the data and the cost per operation is low only in the amortized sense—individual operations can take time $\Omega(n)$ even though this occurs only rarely. Important differences are that self-adjusting heaps support an additional operation, MELD, which Fishspear does not, whereas Fishspear does not require random-access storage. We do not know the relative performance of the two algorithms on restricted classes of operation sequences.

2 Performance Bounds

New criteria are needed to measure adequately the performance of priority queue algorithms. The speed of sorting algorithms, for example, is often expressed in terms of the worst-case or average number of comparisons used in sorting n input elements. They are useful expressions in that context, since in many applications it is reasonable to assume that all initial orderings of the inputs are about equally probable, and thus the parameter n provides an adequate description of the problem.

The case of priority queues presents no such single natural parameter. The total number of INSERT and DELETE_MIN operations performed is one possible measure, but in many applications the maximum length of the queue attained is expected to be far less than the total number of elements inserted. We require a measure more sensitive to the demands made on the priority queue. We also need the further assurance that the running time can be closely related to the number of comparisons made, so that the more combinatorial analysis of the number of comparisons yields results on program performance.

A performance measure we shall use is based on the sequence $\mathbf{h} = (h_1, \ldots, h_n)$ denoting the size of the queue immediately *after* the insertion of each of n elements.[3] The sequence $\mathbf{h}$ is called the *size profile*. By a *run* of the priority queue we shall mean any sequence of priority queue operations for which DELETE_MIN is never applied to an empty queue and the queue is initially and finally empty. For a run with size profile $\mathbf{h}$, the usual heap implementation may use $\log h_j$ comparisons at the jth insertion and a corresponding $2 \log h_j$ comparisons for that first subsequent deletion which takes the queue from size h_j down to size $h_j - 1$. Hence an upper bound for the worst-case number of comparisons is approximately $3 \sum \log h_j$. (More complicated heap implementations reduce this bound to $\sum(\log h_j + O(\log \log h_j))$ [2, 3].) For the naive list implementation, the worst case is $\sum \lceil \log h_j \rceil$. As a lower bound, we have the following result, which appears also in [7].

[3]Here the parameter n is the number of insertions, not the total length of the operation sequence.

Theorem 1 *The worst-case number of comparisons used by any priority queue algorithm on runs with size profile* $\mathbf{h}$ *is at least*

$$\left\lceil \sum_{j=1}^{n} \log h_j \right\rceil .$$

Proof: Consider all possible queue runs with size profile $\mathbf{h}$ and input elements with distinct values. The priority queue algorithm is required to determine the unique correct output order of the elements. Elements simultaneously in the queue are output in order, so each relative ordering of a new element with respect to the elements currently in the queue yields a distinct output sequence. There are h_j places where the j^{th} element can be inserted relative to the other elements in the queue at that time, and each of these yields a different output order; hence, the number of runs which must be distinguished is $\prod h_j$. By the usual information-theoretic argument, any algorithm requires at least $\lceil \log \prod h_j \rceil = \lceil \sum \log h_j \rceil$ binary comparisons to distinguish among these runs. □

For a more refined complexity analysis, we would like to define the depth of an element in the queue at any particular time. When the input elements have distinct values the definition is straightforward, but for the general case a careful treatment is needed. In Fishspear, it is convenient to operate a LIFO regime, but in general the treatment of elements with equal values in a priority queue may be quite arbitrary. For definiteness, we impose the following "a posteriori" total ordering on queue elements in a particular run.

Fix a run and let x_i be the i^{th} element inserted into the queue. We strengthen the given ordering "$<$" on the element values to a total order on the elements x_i as follows. Define $x_i \prec x_j$ if $\text{key}(x_i) < \text{key}(x_j)$, or if $\text{key}(x_i) = \text{key}(x_j)$ and x_i is output before x_j. (This is well defined since we assume all elements are distinct.) The action of the priority queue is consistent with this total order. We define the *depth* of x_i at a time when it is in the queue as one plus the number of elements x_j with $x_j \prec x_i$ in the queue at that time.

There are several applications where most of the elements inserted attain only a relatively shallow depth during their residence in the queue. An example is when the values of input elements are drawn from a uniform distribution and the profile remains at an approximately constant level for long periods. We would like to take advantage of such behavior with an algorithm that does not disturb the deeper elements unnecessarily.

We now define the *max-depth profile* $\mathbf{m}$ for a run as the sequence $(m_1, m_2, \dots)$, where m_j is the maximum depth attained in the queue by element x_j during the run. While the usual heap implementations appear to derive no advantage when $\mathbf{m} \ll \mathbf{h}$, our main theorem shows that Fishspear requires at most

$$c \sum_{j=1}^{n} \log m_j + O(n)$$

comparisons on a run with n insertions (and n deletions), and the coefficient c is less than 2.4.

We may obtain as a corollary that this upper bound for Fishspear holds when "m_j" is replaced by "h_j". Although an individual element can attain a depth in the queue much greater than the size of the queue when it was first inserted, the following shows that, on the average, the m's are no bigger than the h's.

Theorem 2 *Consider a priority queue run with max-depth profile* $\mathbf{m}$ *and size profile* $\mathbf{h}$. *There exists a permutation* π *such that* $m_i \leq h_{\pi(i)}$ *for all* i, $1 \leq i \leq n$.

Proof: Given any run in which the values of the x_i are not all distinct, there is another run with the same sequence of priority queue operations but different elements in which all elements inserted have distinct values, and the total ordering $\prec$ (for corresponding elements in the two runs) is the same in both. This means that the input-output behavior of the queue and the max-depth profile is the same in both runs. So without loss of generality we may suppose that the values of the elements are all distinct and that the element orderings $<$ and $\prec$ coincide.

Suppose there is some pair i, j with $i < j$ and $x_i > x_j$, where x_i, x_j are adjacent in the total ordering. We shall consider the effect of interchanging x_i and x_j in this ordering. Thus, we shall consider a new run identical to the original except that the i^{th} element inserted, $\overline{x}_i$, is x_j, and the j^{th} element inserted, $\overline{x}_j$, is x_i. We let $\overline{m}_i$ and $\overline{m}_j$ denote the maximum depth attained by $\overline{x}_i$ and $\overline{x}_j$, respectively, in this new run.

If, in the original run, x_i leaves the queue before x_j enters, the interchange would not affect m_i or m_j, so $\overline{m}_i = m_i$ and $\overline{m}_j = m_j$. Otherwise, x_i and x_j are simultaneously in the queue. Let M be the maximum depth attained by x_i before x_j enters and let M' and M'' be the original maximum depths that were attained by x_i and x_j respectively after this time. Note that $M' > M''$ since $x_i > x_j$ and x_j leaves the queue first. Thus, in the original run,

$$m_i = \max\{M, M'\} \text{ and } m_j = M'',$$

while in the new run,

$$\overline{m}_i = \max\{M, M''\} \text{ and } \overline{m}_j = M'.$$

We consider two cases and compare the pairs $\{m_i, m_j\}$ and $\{\overline{m}_i, \overline{m}_j\}$.

1. $M \leq M'$. Then $m_i = \overline{m}_j$ and $m_j \leq \overline{m}_i$.
2. $M' \leq M$. Then $m_i = \overline{m}_i$ and $m_j < \overline{m}_j$.

In each case the pair $\{\overline{m}_i, \overline{m}_j\}$, regarded as a multiset, is equal to $\{m_i, m_j\}$ or is bigger in one element.

We can repeat this interchange process wherever there is a pair of elements satisfying the given conditions. The size profile $\mathbf{h}$ is unaltered. The final result will be a FIFO run in which the elements are inserted and deleted in the same order. For such a run, $m_i = h_i$ since the maximal depth of any element is its initial depth. Since each interchange on the way to constructing the FIFO run could only increase the value of $\{m_1, m_2, \ldots\}$ as a multiset, the result follows at once. □

3 The Fishspear Algorithm

The algorithm which we present in Section 3.2 is one instance of a general class of (nondeterministic) algorithms which all operate on the same data structure called a fishspear. The correctness of such algorithms is fairly easy to see. What is not obvious is that there is a deterministic rule for making choices that leads to good behavior.

3.1 Fishspear Data Structure

The *fishspear* data structure represents a priority queue as a collection of sorted lists called *segments*. For segments X and Y, we define $X \leq Y$ if $\text{key}(x) \leq \text{key}(y)$ for every $x \in X$ and $y \in Y$. Note that if $X \leq Y$, $Y \leq Z$ and $Y \neq \emptyset$, then $X \leq Z$. A *k-barbed fishspear* consists of (possibly empty) segments $U, W_k, \ldots, W_1$ and $V_k, \ldots, V_1$. Segments $U, W_k, \ldots, W_1$ form the *shaft* of the spear, and segments $V_k, \ldots, V_1$ are the *barbs* of the spear. The segments satisfy the following conditions:

Fishspear Invariants

1. Each segment is sorted in ascending order according to $<$.
2. $U \leq W_i$ and $U \leq V_i$ for $k \geq i \geq 1$.
3. $W_i \leq W_j$ and $W_i \leq V_j$ for $k \geq i > j \geq 1$.

A fishspear is illustrated in Figure 1.

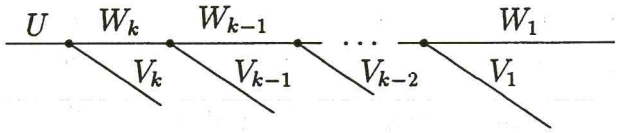

Figure 1: A k-barbed fishspear.

Four primitive operations can be performed on a fishspear:

DELETE_SHARP: Assumes U is non-empty. Deletes and returns the first (i.e., smallest) element of U.

PMERGE: Assumes W_k is non-empty. Performs a "partial merge" of V_k with W_k by comparing the first element in W_k with the first element in V_k, removing the smaller one and appending it to U. If V_k is empty, the first element of W_k is appended to U. (In this case it is convenient for our analysis to consider that a comparison has been made. Indeed in some implementations a comparison with a "sentinel" element may be the most efficient course.)

BARB_CREATE(x): Creates a new segment V_{k+1} initialized to (x). Sets $W_{k+1} := U$, $U :=$ NIL, and $k := k + 1$. The result is a $(k+1)$-barbed fishspear.

BARB_DISPOSE: Assumes $k > 0$ and W_k is empty. If $k = 1$, appends V_1 to U. If $k > 1$, merges V_k into V_{k-1}. In either case, sets $k := k - 1$. The result is a $(k-1)$-barbed fishspear. (For the same reason as for the PMERGE operation, we consider the number of comparisons made in a BARB_DISPOSE to be $|V_1|$ if $k = 1$ and $|V_k| + |V_{k-1}|$ otherwise.)

When comparisons are made between elements with equal value, the outcome is irrelevant to the correctness of the algorithm or our analysis. For definiteness, and perhaps slightly improved performance in some cases, we prefer to take the "smaller" to be the more recently inserted element, which will necessarily be the one in V_k in both PMERGE and BARB_DISPOSE.

In addition to the operations above, we assume the existence of primitive operations for testing and comparing the lengths of the various segments.

Lemma 1 *The primitive operations preserve the Fishspear Invariants.*

Proof: The only nontrivial case is the PMERGE. The invariants follow because the element x that is appended to U satisfies $U \leq x \leq V_i, W_i$, for all i. □

It is easy to implement a priority queue with the help of the above primitives if efficiency is not an issue. The priority queue operation EMPTY? corresponds to the test $U = \emptyset$, *provided that U is non-empty whenever the queue is non-empty.* INSERT(x) can be performed by a BARB_CREATE(x) on the fishspear data structure. For DELETE_MIN, an application of DELETE_SHARP suffices, *provided that U is non-empty.* The following algorithm is a lazy approach to making sure U is non-empty whenever the queue is non-empty:

```
if U is empty then begin
  while k > 0 and W_k is empty do BARB_DISPOSE;
  if k > 0 then PMERGE
end
```

Performing this code before every EMPTY? and DELETE_MIN operation will result in a correct, albeit inefficient, priority queue algorithm.

It is easy to construct examples which cause the above code to make $\Omega(n^2)$ comparisons on an n-element input sequence. For example, such behavior results on *any* sequence of n insertions followed by n DELETE_MIN operations. The n insertions produce an n-barbed fishspear with one element in each barb and an empty shaft. At the time of the first DELETE_MIN, the above code combines all n barbs in a series of unbalanced merges requiring $\Omega(n^2)$ comparisons.

3.2 A Particular Algorithm

The strategy of our algorithm is to perform PMERGE and BARB_DISPOSE operations selectively between priority queue operations so as to maintain a balance on the sizes of the various segments of the fishspear. Exactly what kind of balance our algorithm actually achieves is unclear. Through an involved analysis, we provide an upper bound on the total number of comparisons, but we have been unable to obtain a simple inductive condition on the fishspear which our algorithm preserves and from which our bound follows.

Because of the stack-like quality of the fishspear, it is natural to present our algorithm recursively. However, it is not the queue operations such as INSERT and DELETE_MIN that are defined recursively but rather a "Fishspear Process" FPR which runs autonomously, massaging the fishspear and processing priority queue operations. In other words, we regard FPR as a black box to which we send priority queue operations to be performed and which

sends answers back to us in response to those operations. FPR is separate from the "User" process which is issuing the priority queue operations, although FPR could be implemented as a coroutine just as well. This view is illustrated in Figure 2.

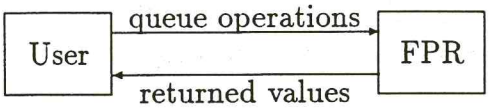

Figure 2: Process structure of the Fishspear algorithm.

We assume two synchronized primitives for interprocess communication, SEND(x) and RECEIVE, where x is a message. (Cf. CSP [4].) A process executing RECEIVE will wait until the other process is ready to execute SEND(x) for some x, at which time the RECEIVE operation returns x as its value and both processes continue. Similarly, a process executing SEND(x) is forced to wait until the other process is ready to execute RECEIVE.

Messages are either 'requests' or 'responses'. A *request* is an element of $D \cup \{\text{'empty?'}, \text{'delete'}\}$; a *response* is an element of $D \cup \{\text{'true'}, \text{'false'}\}$. The 'empty?' request receives the response 'true' or 'false' as appropriate. The 'delete' request corresponds to the DELETE_MIN operation and receives the response x, where $x \in D$ is the deleted minimum element. An operation INSERT(x), for $x \in D$, is performed by sending the request x and receives no response. It is convenient to define the following function on requests:

$$\text{OP}(r) = \begin{cases} \text{'insert'} & \text{if } r \in D \\ r & \text{if } r \in \{\text{'empty?'}, \text{'delete'}\}. \end{cases}$$

We assume process User performs RECEIVE immediately following each SEND('empty?') and SEND('delete') request, in order to obtain the response.

FPR maintains two pieces of global data: an integer k and a k-barbed fishspear stored in variables U, V_j, and W_j, $1 \le j \le k$, as described above. All of the manipulations of this data are performed by the four fishspear primitives, which are invoked from time to time by FPR.

The top-level code MAINR for process FPR is shown in Figure 3. Braces $\{\cdots\}$ are used in the code to delimit comments.

Procedure MAINR:
  { Initially $k = 0$ and $U = \emptyset$ }
  repeat S **forever**

Figure 3: The top-level driver.

The heart of the algorithm is the recursive procedure S. It performs one or more RECEIVE operations, carries out the actions specified by the messages received, responds to each 'delete' or 'empty?' request by issuing a SEND with the answer, and modifies the fishspear to reflect the changes in the queue contents.

The code for S is given in Figure 4. The constant β is a tuning parameter. We are able to prove the best worst-case bounds for $\beta = 0.7034\cdots$, but any value strictly between 0 and 1 yields a correct algorithm. In this program, and elsewhere in this paper, we use the convention that segments and sets are named by upper case letters and their cardinalities are denoted by the corresponding lower case letter. Thus, u denotes the length of U, etc.

```
Procedure S:
1.   local variable b
2.   b := ⌈βu⌉ ..................................... α
3.   repeat
4.     x := RECEIVE
5.     case OP(x) of
6.       'empty?':
7.         SEND(u = 0)
8.         done := false
9.       'delete':
10.        SEND(DELETE_SHARP)
11.        done := (u = 0)
12.      'insert':
13.        BARB_CREATE(x)
14.        while w_k > 0 do ..................... α'
15.          if v_k ≥ u or u ≥ b then PMERGE
16.          else {v_k < u < b} S ................ ω'
17.        BARB_DISPOSE
18.        done := true
19.    end_case
20.  until done
                                  ..... ω
```

Time reference points

Figure 4: The recursive procedure S.

3.3 Correctness of FPR

To show that algorithm FPR correctly implements a priority queue, we must show that it uses the four fishspear primitives correctly and that it always continues to process new inputs.

We begin by examining the recursive structure of FPR and the actions which it performs in more detail. The top-level procedure MAINR (see Figure 3) repeatedly calls the recursive procedure S. Each instance of S may make one or more recursive calls on itself. An instance of a call on S is an *activation.*

We collect some useful properties of FPR in the following lemma.

Lemma 2

1. *If control is in* MAINR *then* $k = 0$.
2. *When any activation of* S *terminates, the fishspear has the same number* k *of barbs as when it was started.*
3. *At lines 4 and 7,* $u = 0$ *if and only if the fishspear is empty.*
4. *At line 10,* $u \neq 0$.

Proof: Parts (1) and (2) follow from the structure of the recursion and the observation that BARB_CREATE increments k, BARB_DISPOSE decrements k, and nothing else changes k.

For part (3), if $u \neq 0$, then the fishspear is clearly non-empty. Suppose $u = 0$ at lines 4 or 7. The assignments in lines 11and 18, and the test in line 20 ensure that $u = 0$ when the current activation began and that the fishspear is the same at line 7 as it was at the start of the activation. It follows from the test in line 15 that only MAINR can call S when $u = 0$. Hence, by part (1), $k = 0$ and the entire queue is empty.

For part (4), the queue is not empty at line 10 since in a run, DELETE_MIN is never applied to an empty queue. The proof that $u \neq 0$ at line 10 follows the reasoning used to establish part (3). □

Theorem 3 *Process* FPR *correctly implements priority queue operations.*

Proof: Each 'insert' or 'delete' request causes an immediate corresponding BARB_CREATE or DELETE_SHARP which ensures that the set of elements represented in the fishspear is correct. Since FPR uses only the four fishspear primitives to change the fishspear, Lemma 1 ensures that the Fishspear Invariants are preserved. Hence, DELETE_SHARP correctly returns a minimum element in the queue. An 'empty?' request is answered according to the truth value of '$u = 0$', which is correct by part (3) of Lemma 2. The fact that the preconditions for DELETE_SHARP are satisfied follows from part (4) of Lemma 2.

It remains to show that FPR continually processes requests. If not, there is a point of time after which control never reaches line 4. If control remains forever within the same activation, it repeatedly executes PMERGE operations in the **while**-loop at line 14. This is impossible since a PMERGE decreases $v_k + w_k$. No new activation is started since then its line 4 would be reached. The remaining alternative is that the current activation is terminated and the recursion level is reduced. But this can happen only a finite number of times. □

3.4 An Example

Figure 5 shows a portion of a sample run of Fishspear. It begins with the fishspear that results from the following sequence of priority queue operations: I71, I15, D, I93, I55, D, I19, I31, I34, I16, I83, D, D, I21, D, D, I13, I92, I50, I48, I40, I51, I17, I91, I80, I12, I33. ("Ix" stands for INSERT(x) and "D" stands for DELETE_MIN.) It then shows the sequence of fishspear primitives triggered by the additional operation I63.

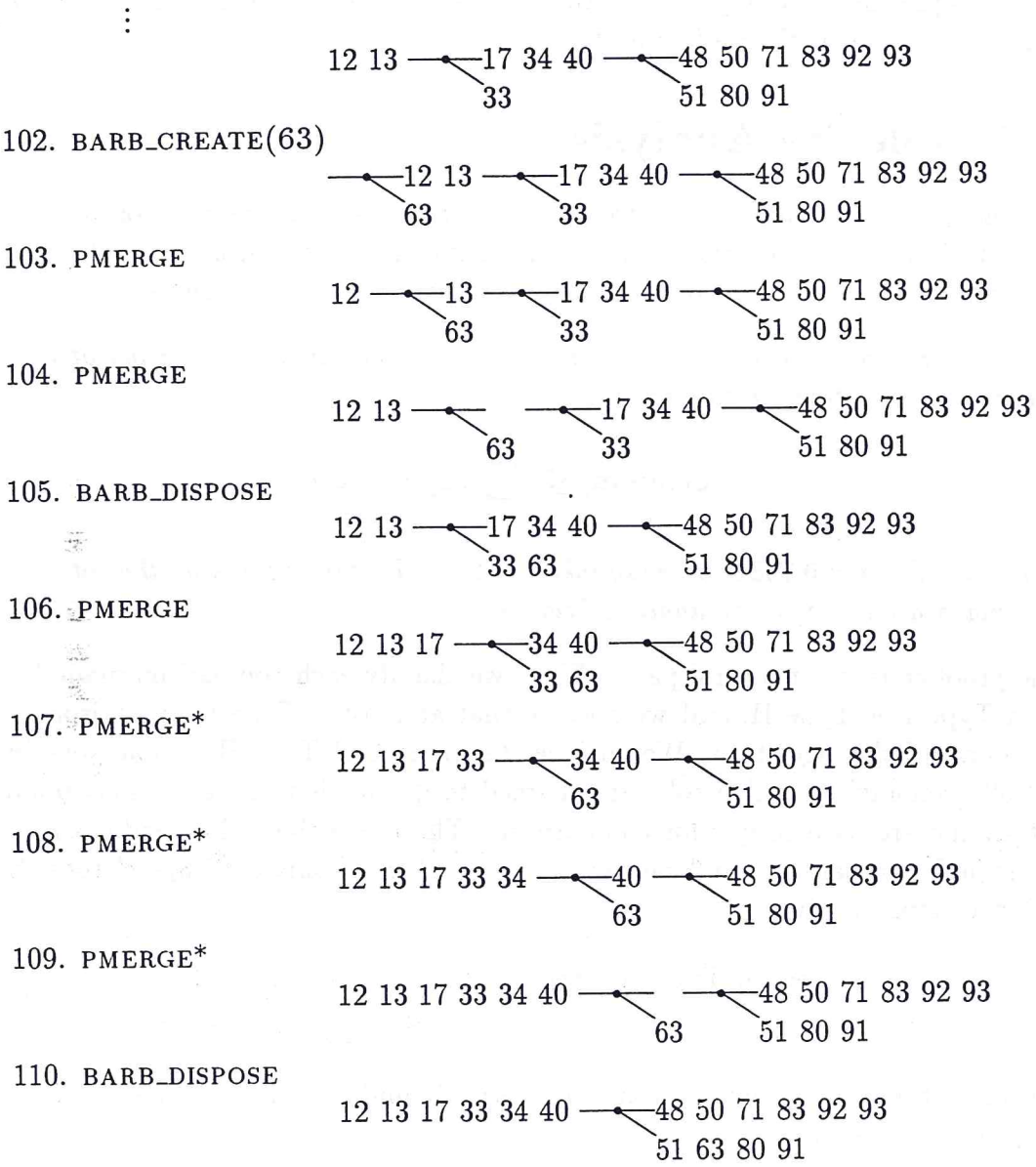

Figure 5: A portion of a run of Fishspear

Each fishspear is described by two lines of text. The first line shows the sequence of segments $U, W_k, \ldots, W_1$. Adjacent segments are separated by a "fork"; consecutive forks denote null segments. The second line shows the sequence of barbs $V_k, \ldots, V_1$. Not shown in the diagram is the b-stack. The values of b in this example are $b_3 = 2$, $b_2 = 3$, and $b_1 = 7$.

A PMERGE operation can be triggered by either of two conditions in line 15 of Figure 4. Those PMERGE operations for which the condition $u \geq b$ holds are indicated by a star $(*)$. In this example, there is only one BARB_CREATE but two BARB_DISPOSE operations, and the fishspear finishes with one less barb.

4 Complexity Analysis

We present an upper bound on the worst-case number of comparisons, Comp($\mathbf{m}$), made by FPR on an input sequence with max-depth profile $\mathbf{m}$. Recall from the definitions of the primitive operations that our convention is extravagant with comparisons.

Theorem 4 *For all β, $0 < \beta < 1$, there exist c, c' such that for all runs of* FPR *with n insertions and max-depth profile* $\mathbf{m}$,

$$\text{Comp}(\mathbf{m}) \leq c \sum_{i=1}^{n} \log_2 m_i + c'n.$$

In particular, for $\beta = 0.7034$, we may take $c = 2.4$. (Further details on the interdependence of c, c' and β are given in the analysis below.)

The proof consists of several parts. First, we classify each comparison made by FPR as being of Type I or Type II, and we observe that at most n Type I comparisons are made in the course of the algorithm. We analyze the number of Type II comparisons by setting up a "toll economy" in which tolls are charged to queue elements at various points in the algorithm and are used to pay for comparisons. The tolls collected are sufficient to pay for all the Type II comparisons, and each element x_i is charged only $c \log m_i + c''$ tolls. Summing over all the elements gives

$$\begin{aligned} \text{number of Type II comparisons} &\leq \text{tolls collected} \\ &\leq c \sum \log m_i + c''n. \end{aligned}$$

The theorem then follows by summing the upper bounds for the two types of comparisons and taking $c' = c'' + 1$.

4.1 Comparison Types

A comparison which results in an element entering the shaft of the fishspear is of Type I; all other comparisons are Type II. An examination of the algorithm shows that there are only two places at which elements are compared: within the PMERGE of line 15 of S, and within the BARB_DISPOSE of line 17 of S. PMERGE compares the first element of V_k with the first element of W_k and appends the smaller to U. Thus, that comparison is of Type I

if the smaller element came from V_k and is of Type II if the smaller element came from W_k. (Recall that we consider a comparison to have been made even if V_k is empty. Such a "pseudo-comparison" is of Type II by this definition.) In BARB_DISPOSE, if $k > 1$ then all comparisons are of Type II, since no elements enter the shaft. If $k = 1$ then, by convention, we consider v_1 comparisons to be made, and these are of Type I.

Lemma 3 *The algorithm makes at most n Type I comparisons.*

Proof: Each element enters the shaft only once, since it can never leave the shaft until eventually deleted from the queue. □

4.2 Progress of FPR

We begin our analysis of the Type II comparisons by relating the amount of progress made by a single activation of S in processing priority queue operations to the size and shape of the fishspear before and after the activation.

We require a notation for naming activations. Names are certain strings σ of positive integers. S_σ, the activation named by σ, is defined inductively on $|\sigma|$. For $i \geq 1$, S_i denotes the i^{th} activation of S from the top-level program MAINR. Inductively, if σ is a non-empty string of natural numbers then $S_{\sigma i}$ denotes the activation of S which results from the i^{th} execution of line 16 by S_σ. If σ is the empty string or a string which does not correspond to an activation of S, then S_σ is undefined. The *level* of S_σ is defined to be $|\sigma|$ and is the recursion depth of the activation.

Each run of the queue is naturally partitioned according to the most recent activation S_σ, if any. We associate with any activation of S all of its computational steps except those carried out in recursive activations called from S.

Now consider a particular activation S_σ. Let $\alpha(\sigma)$ and $\omega(\sigma)$ denote the times at which the activation begins and ends, respectively. If an 'insert' is received, then the block from line 13 to 18 is executed, after which S_σ terminates. In this case, we define $\alpha'(\sigma)$ to be the time just after line 13 of S_σ is executed, and $\omega'(\sigma)$ to be the time just before line 17 is executed. Clearly,

$$\alpha(\sigma) < \alpha'(\sigma) < \omega'(\sigma) < \omega(\sigma).$$

These time points are shown in Figure 4. We omit the name σ of the activation when it is clear from context.

Let τ be a time between steps of S_σ. It is an easy consequence of the definitions that if $\alpha'(\sigma) \leq \tau \leq \omega'(\sigma)$ then $k = |\sigma|$, otherwise $k = |\sigma| - 1$. Thus, the level of recursion is always within one of the number of barbs of the fishspear.

The following notation allows us to talk about the way the fishspear changes over time. Fix a particular activation S_σ of S, and let τ be a time such that $\alpha \leq \tau \leq \omega$ and control is between lines of S in S_σ. We say τ is a time *in* S_σ. Let U_τ be the set of elements in segment U, and k_τ the value of variable k, at time τ. If $k_\tau > 0$, then V_τ and W_τ denote the sets of elements in segments V_{k_τ} and W_{k_τ} at time τ, respectively. We take V_τ and W_τ to be the empty set when $k_\tau = 0$.

Since these definitions depend on the current value of k, V_τ always refers to the top barb of the fishspear, and W_τ refers to the portion of the shaft between the top two barbs (if they exist). As usual, the corresponding lower case letter refers to the cardinality of the set, so $u_\tau = |U_\tau|$, etc.

We define the following sets of elements with respect to an activation S_σ and time τ in S_σ:

$$\begin{array}{lcl} IN_\tau & = & \text{set of elements inserted into the queue after time } \alpha \text{ and still} \\ & & \text{present in the queue at time } \tau; \\ OUT_\tau & = & \text{set of elements present in the queue at time } \alpha \text{ but gone from} \\ & & \text{the queue by time } \tau; \\ U_\tau^{\text{old}} & = & U_\tau \cap U_\alpha, \text{ the set of old elements in } U \text{ at time } \tau; \\ U_\tau^{\text{new}} & = & U_\tau \cap IN_\tau, \text{ the set of new elements in } U \text{ at time } \tau. \end{array}$$

We often omit the subscript τ when τ is clear from context. Some important relationships among these sets are shown in Figure 6 for τ between α' and ω' and are easily proved by induction on τ.

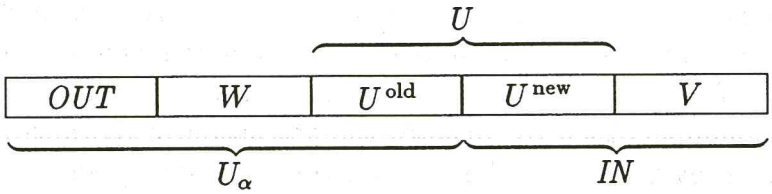

Figure 6: Relations among the basic sets between times α' and ω'.

Lemma 4 *Consider a time τ between the reference points α' and ω'.*

1. *$OUT, W, U^{\text{old}}, U^{\text{new}}, V$ are disjoint sets.*
2. *$U = U^{\text{old}} \cup U^{\text{new}}$.*
3. *$IN = U^{\text{new}} \cup V$.*
4. *$U_\alpha = OUT \cup W \cup U^{\text{old}}$.* □

In the following lemmas, we prove some key properties of S. The reader may find it helpful to refer to Figure 7, which repeats the **while**-loop of S from Figure 4. Note that k, the number of barbs of the fishspear, is incremented by the BARB_CREATE operation and restored to its original value by the corresponding BARB_DISPOSE. Thus the segment W_k is the same before and after any activation S_σ of S. The only segments that change as a result of S_σ are U and V_k. The test condition '$u \geq b$' in S is important in maintaining the proportions of the fishspear.

Lemma 5 *Let S_σ be an activation of* S. *Let τ be any time in S_σ, $\alpha'(\sigma) \leq \tau \leq \omega'(\sigma)$, such that line 15 of S_σ is never reached with $u \geq b$ at any time during the interval from α' to τ. Then*

$$in_\tau \geq u_\tau - 1.$$

```
13.     BARB_CREATE(x)                                  ..... α'
14.     while w_k > 0 do
15.       if v_k ≥ u or u ≥ b then PMERGE
16.       else {v_k < u < b} S
17.     BARB_DISPOSE                                    ..... ω'
```

Figure 7: While-loop from the recursive procedure S.

Proof: Observe that if the condition '$u \geq b$' once becomes true then it remains true for the duration of S_σ, because as long as it is true the 'then' branch of the condition in line 15 is always taken, and PMERGE never decreases u.

We proceed to prove the lemma. At time $\tau = \alpha'$, the lemma holds trivially since $u_\tau = 0$. Subsequently, the only places where *IN* or U is modified are in lines 15 and 16 of S. Let τ be a time just *after* one of those two lines has been performed, and suppose the conditions of the lemma are satisfied at time τ. Let π be the time just *before* the execution of that line. By induction, we assume the lemma holds at time π. We consider the two cases in turn.

Case 1: The PMERGE in line 15 was performed. Since $u_\pi < b$ by assumption, and control reached the PMERGE, we have $v_\pi \geq u_\pi$. It follows from Lemma 4 that $in_\pi \geq v_\pi$. Since the PMERGE does not change *IN* and it moves exactly one element into U, we have $in_\tau = in_\pi \geq u_\pi = u_\tau - 1$ as desired.

Case 2: The recursive call in line 16 was performed. Consider its effect on U and V. The fishspear is left with the same number of segments as it had previously. Line 16 can only *decrease* (or leave unchanged) the size of U^{old}, for the segment U immediately after the recursive call consists entirely of elements that were in U just before the call together with new elements (that is, elements inserted into the queue during the recursive call). Line 16 can only *increase* (or leave unchanged) the size of V, for its overall effect is to add to V some new elements inserted during the recursive call. Using Lemma 4, it follows that $in - u = v - u^{\text{old}}$ can only *increase*. Since $in_\pi - u_\pi \geq -1$ by the induction hypothesis, then also $in_\tau - u_\tau \geq -1$, as desired.

The lemma follows by induction. □

The following is a direct consequence of Lemma 5 and puts a lower bound on the amount of work done by S as a function of u_α, the initial size of U.

Lemma 6 *For any activation* S_σ *of* S, *either*

$$in_\omega + out_\omega \geq u_\alpha - 1$$

or

$$in_\omega \geq b - 1,$$

where $b = \lceil \beta u_\alpha \rceil$.

Proof: There are two cases, depending on whether line 15 of S_σ was ever reached with $u \geq b$.

Case 1: This inequality was never true at line 15. Then by Lemma 5, $in_{\omega'} \geq u_{\omega'} - 1$. Also, $w_{\omega'} = 0$ since the 'while' loop of line 14 terminated, so it follows from Lemma 4 that $out_{\omega'} = u_\alpha - u_{\omega'}^{\text{old}}$. Hence,

$$\begin{aligned} in_\omega + out_\omega \;=\; in_{\omega'} + out_{\omega'} &\geq (u_{\omega'} - 1) + (u_\alpha - u_{\omega'}^{\text{old}}) \\ &\geq u_\alpha - 1. \end{aligned}$$

Case 2: The inequality was true at some execution of line 15. Let τ be the time at which the first such execution began. Then by Lemma 5, $in_\tau \geq u_\tau - 1$. From time τ to ω', only PMERGE operations are done, so $in_{\omega'} = in_\tau$. Since the test came out true, we have $u_\tau \geq b$. Hence

$$in_\omega = in_{\omega'} \geq u_\tau - 1 \geq b - 1. \qquad \square$$

4.3 The Toll Economy

To bound the number of Type II comparisons, we associate "tokens" with each element x_i inserted into the queue, the total value of which depends on the maximum depth m_i attained by x_i. Lemma 7 gives an upper bound on the total value T of all tokens.

Each Type II comparison is naturally associated with the unique activation in which it occurred, as described in Section 4.2. We similarly assign each token to a unique activation in the run. Lemma 9 shows that the total number of Type II comparisons assigned to S_σ is at most the total value of all tokens assigned to S_σ. Summing over all activations shows that the total number of Type II comparisons is at most T, and hence the upper bound on T applies to the number of Type II comparisons as well.

4.3.1 Tokens

Associated with each element x_i inserted into the queue are two sets of tokens, the *in-tokens* and the *out-tokens*. The tokens in each set are numbered sequentially from 1 to $\lfloor (m_i + 1)/\gamma \rfloor$, where γ is a small positive constant. (Certainly $\gamma < \beta/2$.) In addition, each element has two *base tokens*. The *value* of in-token (out-token) number d is t_{I}/d (t_{O}/d), and the value of a base token is t_{B}, where t_{I}, t_{O}, and t_{B} are positive constants to be specified later.

Let T be the total value of all tokens. The following lemma gives an upper bound on the tolls collected.[4]

Lemma 7

$$T \leq 2nt_{\text{B}} + (t_{\text{I}} + t_{\text{O}}) \left(\sum_{i=1}^{n} \ln m_i + n \ln \frac{2e}{\gamma} \right).$$

Proof: Since the largest number of a token associated with x_i is $\lfloor (m_i + 1)/\gamma \rfloor$, we have

$$\begin{aligned} T \;&=\; 2nt_{\text{B}} + (t_{\text{I}} + t_{\text{O}}) \sum_{i=1}^{n} \sum_{d=1}^{\lfloor (m_i+1)/\gamma \rfloor} \frac{1}{d} \\ &\leq\; 2nt_{\text{B}} + (t_{\text{I}} + t_{\text{O}}) \sum_{i=1}^{n} \left(1 + \ln \frac{m_i + 1}{\gamma} \right) \end{aligned}$$

[4] $\ln x$ denotes the natural logarithm of x.

since

$$\sum_{j=1}^{\lfloor t \rfloor} \frac{1}{j} \leq 1 + \int_1^t \frac{1}{x}\, dx = 1 + \ln t.$$

Hence,

$$T \leq 2nt_B + (t_I + t_O) \sum_{i=1}^{n} \ln \frac{2m_i e}{\gamma}$$

and the result follows. □

4.3.2 Parameters

A number of parameters have been introduced so far. β affects the performance of the algorithm, and γ, t_B, t_I, and t_O affect the amount of tolls collected. Rather than simply fix these parameters to particular constants and then prove our bounds, we choose to leave them unspecified until the end, introducing instead only the constraints needed to make our analysis work. Our final step will be to show that the constraints are satisfiable and to optimize the constants in the final bound. In order to express the constraints, we introduce two additional parameters, θ and ψ.

Constraint Set *We assume that $\beta, \gamma, t_B, t_I, t_O, \theta, \psi$ are positive constants which satisfy the following:*

$$2\gamma < \beta < 1, \tag{1}$$

$$\theta = \max\left\{ \frac{2+\beta}{\beta(1-\ln\beta)}, \frac{1}{-\ln\beta} \right\}, \tag{2}$$

$$\theta t - (\theta t + \psi(1-t)) \ln t - 2 - t \geq 0 \text{ for all } t \in (0, \beta), \tag{3}$$

$$\theta \leq t_I - \frac{\gamma}{\beta}(t_I + t_O), \tag{4}$$

$$\psi \leq t_O - \frac{\gamma}{\beta}(t_I + t_O), \tag{5}$$

$$t_B = t_I(1-\gamma). \tag{6}$$

We define the following function:

$$F(p,q,r) = \theta p - (\theta \max\{p,q\} + \psi r) \ln p - 2 - q.$$

F arises as a residual term in our analysis, with suitable terms substituted for p, q, and r. Note that the inequality of (3) is just $F(t,t,1-t) \geq 0$. The following claim is a purely analytical fact that we will need later.

Claim *Let $q, r \geq 0$. If $0 < p < \beta$ and either $q + r \geq 1$ or $q \geq \beta$, then the Constraint Set implies $F(p,q,r) \geq 0$.*

Proof: Consider the partial derivative when $p < q$:

$$\begin{aligned}\frac{\partial F}{\partial p} &= \theta - (\theta q + \psi r)\frac{1}{p} \\ &= \theta\left(1 - \frac{q}{p}\right) - \psi\frac{r}{p} \\ &< 0.\end{aligned}$$

This shows that F decreases as p increases to q.

We have three cases depending on how q relates to p and β.

Case 1: $q \le p < \beta$. Then $r \ge (1-q) \ge (1-p)$. Also, $p < \beta < 1$, so $\ln p < 0$. Hence,

$$\begin{aligned}F &= \theta p - (\theta p + \psi r)\ln p - 2 - q \\ &\ge \theta p - (\theta p + \psi(1-p))\ln p - 2 - p.\end{aligned}$$

By (3), $F \ge 0$ as desired.

Case 2: $p < q < \beta$. Again $r \ge 1 - q$. Since the partial derivative of F with respect to p is negative, we can replace p by q to get

$$\begin{aligned}F &= \theta p - (\theta q + \psi r)\ln p - 2 - q \\ &\ge \theta q - (\theta q + \psi(1-q))\ln q - 2 - q.\end{aligned}$$

Again, (3) gives $F \ge 0$ as desired.

Case 3: $p < \beta \le q$. The partial derivative of F with respect to p is again negative, so we can replace p by β and r by 0 to get

$$\begin{aligned}F &= \theta p - (\theta q + \psi r)\ln p - 2 - q \\ &\ge \theta\beta - \theta q \ln\beta - 2 - q \\ &= (\theta\beta - 2) - q(\theta\ln\beta + 1). \qquad (7)\end{aligned}$$

There are two subcases.

Subcase 1: $\beta \ge -2\ln\beta$. Then by (2) we have $\theta = -1/\ln\beta \ge 2/\beta$. Hence, by (7),

$$\begin{aligned}F &\ge (\theta\beta - 2) - q(\theta\ln\beta + 1) \\ &\ge 0.\end{aligned}$$

Subcase 2: $\beta < -2\ln\beta$. By (2) we have

$$\theta = \frac{2+\beta}{\beta(1-\ln\beta)} > \frac{1}{-\ln\beta}, \qquad (8)$$

and so, using the Case 3 assumption that $\beta \le q$, (7) gives

$$\begin{aligned}F &\ge (\theta\beta - 2) - \beta(\theta\ln\beta + 1) \\ &= \theta\beta(1-\ln\beta) - 2 - \beta \\ &= 0.\end{aligned}$$

Thus, in all three cases, $F \ge 0$, establishing the claim. □

4.3.3 Assigning Tokens to Activations

We give rules for assigning tokens to particular activations. A token assigned to activation S_σ is said to be *collected* by S_σ.

S_σ collects tokens as follows, provided they exist:

- One base token and in-tokens $1, \ldots, u_{\alpha(\sigma)} - 1$ from any element that was received for insertion in the queue in line 4 of S_σ.
- One base token and out-tokens $1, \ldots, u_{\alpha(\sigma)} - 1$ from any element that was deleted in response to a 'delete' received in line 4 of S_σ.
- In-tokens $u_{\alpha(\sigma i)}, \ldots, u_{\alpha(\sigma)} - 1$ of element x if activation $S_{\sigma i}$ is defined and x was inserted in the queue during the time interval spanned by $S_{\sigma i}$.
- Out-tokens $u_{\alpha(\sigma i)}, \ldots, u_{\alpha(\sigma)} - 1$ of element x if activation $S_{\sigma i}$ is defined and x was deleted from the queue during the time interval spanned by $S_{\sigma i}$.

Note that, since $\beta < 1$, the test in line 15 of S ensures that $u_{\alpha(\sigma i)} < u_{\alpha(\sigma)}$. Hence, no token is collected by more than one activation. In particular, in-token number d of element x (if it exists) is collected by S_σ, where σ is the longest string such that $d < u_{\alpha(\sigma)}$ and x is inserted in the queue during the time interval spanned by S_σ. It follows that any in-tokens $u_{\alpha(\sigma i)}, \ldots, u_{\alpha(\sigma)} - 1$ which exist are collected from each element x_j that entered the queue during the i^{th} execution of line 16 of S_σ. A similar observation holds for out-tokens. We let $T(\sigma)$ denote the total value of all tokens collected by activation S_σ.

4.3.4 Counting Tolls

The tolls collected during S_σ can be related to the progress parameters of Section 4.2. The analysis is made difficult by the fact that token d only exists for an element x_j if m_j is sufficiently large. While any individual m_j might be small, the following lemma shows that most elements of any large group of elements simultaneously in the queue necessarily have large m's. Thus, we obtain lower bounds on the tolls collected by identifying points during the activation in which large sets of elements are simultaneously present in the queue and counting up the value of the tokens they contain.

Lemma 8 *For any set X of elements present simultaneously in the queue, token number d exists for all but at most $\lfloor \gamma d \rfloor - 1$ of them.*

Proof: For all except $\lfloor \gamma d \rfloor - 1$ elements of X, the current depth, and hence the maximum depth attained, equals or exceeds $\lfloor \gamma d \rfloor$. Hence these elements all have tokens numbered up to at least $\lfloor (\lfloor \gamma d \rfloor + 1)/\gamma \rfloor$, and so at least d. □

Let $\text{type}_{\text{II}}(\sigma)$ be the number of Type II comparisons associated with activation S_σ. This should be bounded above by the tolls collected.

Lemma 9 *(Tolls Lemma). Let S_σ be an activation of* S. *Then*

$$T(\sigma) > \text{type}_{\text{II}}(\sigma).$$

Proof: We consider first the case where $u_\alpha = 0$. The test condition of line 15 ensures that this case only occurs when S_σ is called from MAINR. Therefore $k = 0$ and the queue is empty when S_σ begins. Control remains in S_σ and no comparisons are made until an 'insert' request is received. (Recall that 'delete' requests are not legal when the queue is empty.) At this point, lines 13, 14, and 17 are performed, causing one Type I and no Type II comparisons to be made. Hence, $T(\sigma) = 1$ and $\text{type}_{\text{II}}(\sigma) = 0$, and the result follows.

We assume for the remainder of the proof that $u_\alpha > 0$.

Recall that S_σ collects at most one in-token d from each element x that was inserted into the queue during the time interval spanned by S_σ, but in-token d might not be collected from x, either because it does not exist, or because $d \geq u_{\alpha(\sigma)}$, or because x was inserted during the time interval spanned by $S_{\sigma i}$ (the i^{th} recursive call on S) and $d < u_{\alpha(\sigma i)}$. Similar remarks apply to the counting of out-tokens. Thus, a careful analysis is required to avoid overcounting.

Our strategy for counting the in-tokens collected by S_σ is to identify for each d a set of elements which are simultaneously in the queue and from which S_σ is allowed to collect any in-tokens d that exist. Lemma 8 guarantees that in-token d exists for all but $\lfloor \gamma d \rfloor - 1$ of the elements in the set. To be sure that S_σ is allowed to collect these in-tokens, we require that $d < u_{\alpha(\sigma)}$, and we show that every element in the set was either received for insertion by line 4 of S_σ, or was inserted into the queue during an activation $S_{\sigma i}$ for which $d \geq u_{\alpha(\sigma i)}$. Similarly, we count out-tokens collected by S_σ by identifying for each d a set of elements which are simultaneously in the queue and for which S_σ is allowed to collect any out-tokens d that exist. Even though we fail to count all of the tokens actually collected by S_σ, the tokens we do count are sufficient to pay for all of the comparisons made by S_σ.

Before defining the sets from which we count in-tokens, we describe particular activations $S_{\sigma i}$ whose start times will be of particular interest. Consider successive executions of line 16 during the while-loop of S_σ. Let $i_1 = 1$ and let i_{r+1} be the least number i such that $S_{\sigma i}$ exists and $u_{\alpha(\sigma i)} > u_{\alpha(\sigma i_r)}$. Finally, let s be the largest index for which i_s is defined. If S_σ does not execute line 16 at all, then all i_r are undefined, and we take $s = 0$. For each r for which i_r is defined, define $\mu(r) = \alpha(\sigma i_r)$; thus $\mu(r)$ is the time at which activation $S_{\sigma i_r}$ begins. (For technical convenience, we take $u_{\mu(0)} = 1$.) Note that the μ's have been chosen so that $u_{\mu(1)} < u_{\mu(2)} < \ldots < u_{\mu(s)}$, and $u_{\alpha(\sigma i)} \leq u_{\mu(r)}$ for all i, r such that $1 \leq r < s$ and $i < i_{r+1}$ or $r = s$. Note also that $u_{\mu(s)} < u_\alpha$ by the test in line 15 of S.

Each of $IN_{\mu(j)}$, IN_ω, and OUT_ω are sets of elements which are simultaneously in the queue—the elements of $IN_{\mu(j)}$ are all present at time $\mu(j)$, the elements of IN_ω are all there at time ω, and the elements of OUT_ω were all in the queue at time α. If $u_{\mu(j-1)} \leq d < u_{\mu(j)}$, we count in-token d for all but $\lfloor \gamma d \rfloor - 1$ of the elements in $IN_{\mu(j)}$. Similarly, if $u_{\mu(s)} \leq d < u_\alpha$, we count in-token d for all but $\lfloor \gamma d \rfloor - 1$ of the elements in IN_ω. Finally, if $u_{\mu(s)} \leq d < u_\alpha$, we count out-token d for all but $\lfloor \gamma d \rfloor - 1$ of the elements in OUT_ω. Because $u_{\mu(1)} < u_{\mu(2)} < \ldots < u_{\mu(s)}$, in-token d is counted for at most one of the sets $IN_{\mu(j)}$, $1 \leq j \leq s$, and IN_ω.

We must argue that each token so counted is indeed collected by S_σ according to the rules in Section 4.3.3. Suppose in-token d from element $x \in IN_{\mu(j)}$ is counted, so $u_{\mu(j-1)} \leq d < u_{\mu(j)}$. If x was received for insertion into the queue at line 4 of S_σ, then S_σ indeed collects d since $d < u_{\mu(j)} < u_\alpha$. Otherwise, x was inserted into the queue during the time spanned by activation $S_{\sigma i}$ for some $i < i_j$. But again S_σ collects d since $u_{\alpha(\sigma i)} \leq u_{\mu(j-1)} \leq d < u_{\mu(j)} < u_\alpha$.

Similar arguments apply to in-tokens counted from elements in IN_ω and out-tokens counted from elements in OUT_ω.

We now derive a lower bound on the value of the tokens collected by S_σ.

1. At least one base token is collected by S_σ because at least one element is inserted or deleted. It has value
$$T_1(\sigma) = t_B. \tag{9}$$

2. Let $1 \le j \le s$ and let $u_{\mu(j-1)} \le d < u_{\mu(j)}$. By Lemma 8, in-token number d is collected from all but $\lfloor \gamma d \rfloor - 1$ of the elements in $IN_{\mu(j)}$ for a total value of at least $t_I(in_{\mu(j)} - (\gamma d - 1))/d$. Summing over j and d gives a total value of at least
$$T_2(\sigma) = \sum_{j=1}^{s} \sum_{d=u_{\mu(j-1)}}^{u_{\mu(j)}-1} t_I(in_{\mu(j)} - (\gamma d - 1))\frac{1}{d}. \tag{10}$$

3. Let $u_{\mu(s)} \le d < u_\alpha$. By Lemma 8, in-token d is collected from all but $\lfloor \gamma d \rfloor - 1$ of the elements in $IN_{\mu(s)} \cup IN_\omega$ for a total value of at least
$$T_3(\sigma) = \sum_{d=u_{\mu(s)}}^{u_\alpha - 1} t_I(\max\{in_{\mu(s)}, in_\omega\} - (\gamma d - 1))\frac{1}{d}. \tag{11}$$

4. Let $u_{\mu(s)} \le d < u_\alpha$. By Lemma 8, out-token d is collected from all but $\lfloor \gamma d \rfloor - 1$ of the elements in OUT_ω for a total value of at least
$$T_4(\sigma) = \sum_{d=u_{\mu(s)}}^{u_\alpha - 1} t_O(out_\omega - (\gamma d - 1))\frac{1}{d}. \tag{12}$$

Thus, $T(\sigma) \ge \sum_{k=1}^{4} T_k(\sigma)$.

By Lemma 5, $in_{\mu(j)} \ge u_{\mu(j)} - 1$, and since $u_{\mu(j)} - 1 \ge d$ in the summation, (10) yields

$$\begin{aligned} T_2(\sigma) &\ge \sum_{j=1}^{s} \sum_{d=u_{\mu(j-1)}}^{u_{\mu(j)}-1} t_I(d - (\gamma d - 1))\frac{1}{d} \\ &\ge \sum_{j=1}^{s} t_I(1-\gamma)(u_{\mu(j)} - u_{\mu(j-1)}) \\ &= t_I(1-\gamma)(u_{\mu(s)} - 1). \end{aligned} \tag{13}$$

With the inequality
$$\sum_{d=u_{\mu(s)}}^{u_\alpha - 1} \frac{1}{d} \ge \ln \frac{u_\alpha}{u_{\mu(s)}},$$
equations (11) and (12) yield

$$\begin{aligned} T_3(\sigma) + T_4(\sigma) &\ge \left(t_I \max\{in_{\mu(s)}, in_\omega\} + t_O out_\omega - (t_I + t_O)(\gamma u_\alpha - 1)\right) \sum_{d=u_{\mu(s)}}^{u_\alpha - 1} \frac{1}{d} \\ &\ge \left(t_I \max\{u_{\mu(s)}, in_\omega + 1\} + t_O(out_\omega + 1) - (t_I + t_O)\gamma u_\alpha\right) \ln \frac{u_\alpha}{u_{\mu(s)}} \end{aligned} \tag{14}$$

since $in_{\mu(s)} \geq u_{\mu(s)} - 1$ by Lemma 5. Whichever case of Lemma 6 holds, we have $\beta u_\alpha \leq in_\omega + out_\omega + 1 \leq \max\{u_{\mu(s)}, in_\omega + 1\} + out_\omega + 1$; hence,

$$\gamma u_\alpha \leq (\gamma/\beta)\max\{u_{\mu(s)}, in_\omega + 1\} + (\gamma/\beta)(out_\omega + 1). \tag{15}$$

Using (4) and (5) from the Constraint Set, (14) and (15) yield

$$\begin{aligned} T_3(\sigma) + T_4(\sigma) &\geq \Big(\big(t_{\mathrm{I}} - \frac{\gamma}{\beta}(t_{\mathrm{I}} + t_{\mathrm{O}})\big)\max\{u_{\mu(s)}, in_\omega + 1\} \\ &\quad + \big(t_{\mathrm{O}} - \frac{\gamma}{\beta}(t_{\mathrm{I}} + t_{\mathrm{O}})\big)(out_\omega + 1)\Big)\ln\frac{u_\alpha}{u_{\mu(s)}} \\ &\geq \big(\theta\max\{u_{\mu(s)}, in_\omega + 1\} + \psi(out_\omega + 1)\big)\ln\frac{u_\alpha}{u_{\mu(s)}}. \end{aligned} \tag{16}$$

From (1), (4) and (6), we have $t_{\mathrm{B}} = t_{\mathrm{I}}(1-\gamma) \geq \theta$. Thus, adding together (9), (13), and (16), we get

$$\begin{aligned} T(\sigma) &\geq t_{\mathrm{B}} + t_{\mathrm{I}}(1-\gamma)(u_{\mu(s)} - 1) \\ &\quad + \big(\theta\max\{u_{\mu(s)}, in_\omega + 1\} + \psi(out_\omega + 1)\big)\ln\frac{u_\alpha}{u_{\mu(s)}} \\ &\geq \theta u_{\mu(s)} + \big(\theta\max\{u_{\mu(s)}, in_\omega + 1\} + \psi out_\omega\big)\ln\frac{u_\alpha}{u_{\mu(s)}}. \end{aligned} \tag{17}$$

Substituting

$$p = \frac{u_{\mu(s)}}{u_\alpha},\ q = \frac{in_\omega + 1}{u_\alpha},\ \text{and } r = \frac{out_\omega}{u_\alpha}$$

in (17) gives

$$T(\sigma) \geq 2u_\alpha + in_\omega + 1 + u_\alpha F(p, q, r). \tag{18}$$

The test in line 15 of S_σ ensures that $u_{\mu(s)} < \beta u_\alpha$, so $p < \beta$. Lemma 6 implies that either $q + r \geq 1$ or $q \geq \beta$, so the Claim gives

$$F(p, q, r) \geq 0. \tag{19}$$

We also have $v_\alpha < u_\alpha$ for activation S_σ. This is assured for recursive activations of S by the test in line 15 of the parent activation. If S_σ was started directly by MAINR, it holds because $v_\alpha = 0$ and we have assumed $u_\alpha > 0$. Hence, $v_\omega = v_\alpha + v_{\omega'} < u_\alpha + in_\omega + 1$, so

$$2u_\alpha + in_\omega + 1 > u_\alpha + v_\omega. \tag{20}$$

Combining (18), (19), and (20), we have

$$T(\sigma) > u_\alpha + v_\omega. \tag{21}$$

To complete the proof, we note that Type II comparisons assigned to S_σ are made only in lines 15 and 17 of S. At most u_α Type II comparisons are made by PMERGE in line 15, since each such comparison removes an element from W, and W at time α' has size at most

u_α. The number of comparisons made by BARB_DISPOSE in line 17 is at most v_ω. If $k_\alpha > 0$, it simply merges together two segments to produce V_ω; if $k_\alpha = 0$, all comparisons made are of Type I. Hence, (21) yields

$$T(\sigma) > \text{type}_{\text{II}}(\sigma). \qquad \square$$

For the next lemma we sum the above inequality over all activations.

Lemma 10 *Let β, γ, t_{B}, t_{I}, t_{O} be constants for which the Constraint Set can be satisfied. The total number of Type II comparisons made by Fishspear on a run with n insertions and max-depth profile* $\mathbf{m}$ *is at most*

$$2nt_{\text{B}} + (t_{\text{I}} + t_{\text{O}}) \left(\sum_{i=1}^{n} \ln m_i + n \ln \frac{2e}{\gamma} \right).$$

Proof: The run can be partitioned into segments of operations processed by activations of S invoked directly by MAINR. Each of these activations is complete except for the last. We consider the run to have ended when all priority queue operations have been processed and control reaches line 4 of some activation S_σ. By assumption, the queue is empty at the end of the run. The tests on lines 11, 14, and 20 ensure that control returns to MAINR after the last 'delete_min' operation and before any other priority queue operation is processed. After that, S_σ is started, and only 'empty?' operations are processed. Thus, no comparisons are made by the incomplete final activation S_σ. The bound for the completed activations is an immediate consequence of Lemmas 7 and 9. $\square$

We are finally in a position to prove the main theorem of this section.

Proof of Theorem 4: We first argue that the Constraint Set is satisfiable for any positive γ, β satisfying (1). Equation (2) defines θ. The left hand side of (3) as a function of t is bounded from below over the interval $(0, \beta)$, and it is a linear function of ψ with a positive coefficient that is bounded away from zero. It follows that (3) is satisfied for sufficiently large ψ. Similarly, (4) and (5) can be satisfied by taking $t_{\text{I}} = t_{\text{O}}$ sufficiently large since $2\gamma/\beta < 1$. Finally, (6) defines t_{B}.

With

$$c = (t_{\text{I}} + t_{\text{O}}) \ln 2 \tag{22}$$

and

$$c' = 2t_{\text{B}} + 1 + (t_{\text{I}} + t_{\text{O}}) \ln \frac{2e}{\gamma},$$

the bound of Theorem 4 follows from Lemmas 3 and 10.

We get our best bounds by choosing $\beta = -2 \ln \beta = 0.7034\cdots$. Equation (2) then yields $\theta = 2.843\cdots$. Calculus and numerical evaluation show that $\psi = 0.5674\cdots$ satisfies (3), and equality holds (to within the limits of our precision) for $t = 0.141\cdots$. (The function of (3) over the interval $(0, \beta)$ is shown in Figure 8.) Thus, $\theta + \psi = 3.410\cdots$. By choosing γ sufficiently close to 0 and not insisting that $t_{\text{I}} = t_{\text{O}}$, we can make $t_{\text{I}} + t_{\text{O}}$ arbitrarily close to $3.410\cdots$. Finally, (22) shows that the constant c of Theorem 4 can be chosen arbitrarily close to

$$3.410\cdots \times \ln 2 = 2.363\cdots.$$

In particular, we may choose $c = 2.4$. $\square$

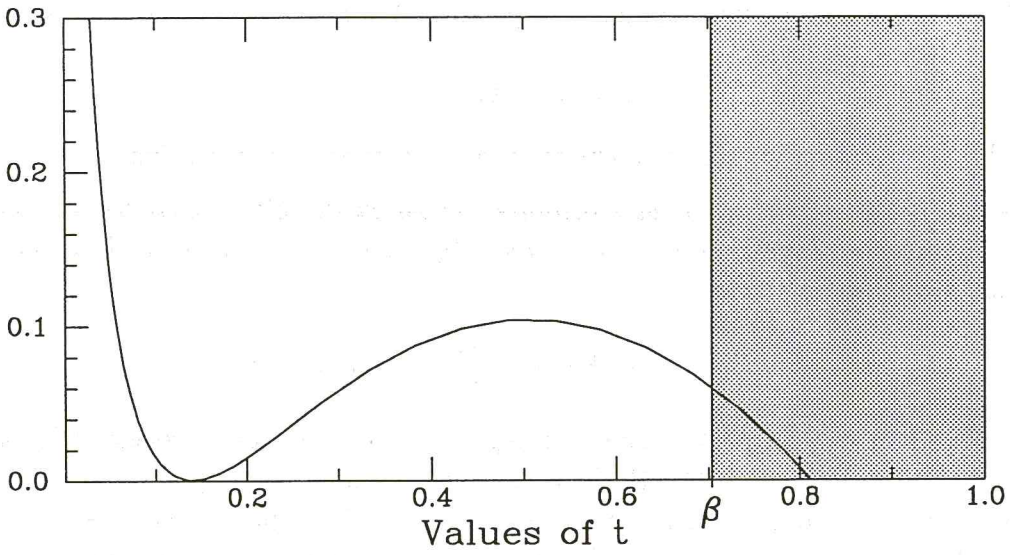

Figure 8: The function $F(t, t, 1-t)$ for $\theta = 2.844$ and $\psi = 0.5675$.

5 An Iterative Algorithm

The recursive algorithm presented in Section 3 can be expressed in an equivalent iterative form. While the recursive version is easier to analyze, the iterative version is shorter and simpler. The iterative form also lends itself to a conventional implementation of the data type as a collection of procedures and functions which operate on a shared data structure.

5.1 Iterative Algorithm FPI

The iterative algorithm includes one new procedure, CLEAN, which performs the PMERGE and BARB_DISPOSE operations involved in rebalancing the queue. It always leaves $U \neq \emptyset$ unless the entire queue is empty.

Implicit in the recursive structure of S is a pushdown stack on which the local variable b is kept. In the iterative algorithm, we keep this data on an explicit pushdown stack, the b-stack. The variable associated with the i^{th}-level current activation of S is denoted by b_i. By Lemma 2, the level of the most recent current activation of S is equal to k at line 15 of S where b is used, so in the corresponding test of the iterative algorithm, b can be replaced by b_k.

The complete iterative algorithm is presented in Figure 9. Initially, $k = 0$ and $U = \emptyset$.

5.2 Analysis of Algorithm FPI

We analyze FPI by showing it to be equivalent to FPR. The equivalence between the two algorithms is strong in the sense that both, when presented with the same sequence of requests, perform the same sequence of fishspear primitive operations, SEND and RECEIVE

```
Procedure CLEAN:
1.  local variable balanced := false
2.  while k > 0 and not balanced do
3.    if w_k = 0 then BARB_DISPOSE
4.    else if v_k ≥ u or u ≥ b_k then PMERGE
5.    else balanced := true
6.  b_{k+1} := ⌈βu⌉

Procedure MAINI:
1.  CLEAN
2.  repeat forever
3.    x := RECEIVE
4.    case OP(x) of
5.      'empty?':
6.        SEND(u = 0)
7.      'delete':
8.        SEND(DELETE_SHARP)
9.        if u = 0 then CLEAN
10.     'insert':
11.       BARB_CREATE(x)
12.       CLEAN
13.   end_case
14. end_repeat
```

Figure 9: Algorithm FPI.

operations, and updates to the b-stack. Hence, the sequences of fishspears, b-stacks, and results returned by the two algorithms are identical.

To state this more precisely, we look in some detail at the state structure of the two algorithms. A *complete state* of an algorithm is the entire collection of data that determines its future behavior. In the case of FPR, this consists of the fishspear, the recursion stack, the control point, and the two variables x and *done*. In the case of FPI, this consists of the fishspear, the b-stack, the control point, and the variables x and *balanced*. The *essential state* consists of a portion of the complete state that is common to both algorithms, namely, the fishspear and the sequence of b-values.

Next we distinguish those control points of the two algorithms that immediately precede operations which affect the essential state or produce input and output. Namely, we consider the points just before the four fishspear primitives are invoked, just before SEND and RECEIVE are performed, and just before the assignments to b. Each algorithm performs SEND at two places in its code and each other of these operations at only one place, so each algorithm has eight distinguished control points. Note that it is unnecessary to differentiate the control points in the three occurrences of CLEAN since their subsequent executions are identical.

Let z be a distinguished control point of either algorithm and q an essential state. We

call the pair (z, q) a *determining state* of that algorithm. If s is a complete state with distinguished control point z and essential state q, then we say that s *has* (z, q) *as its determining part.* It is easily verified for both algorithms that for any determining state (z, q) there is a unique determining state (z', q') such that, starting from any complete state s with determining part (z, q), at the next distinguished control point (if any) the state s' has determining part (z', q'). Moreover, such a complete state s' is always reached. Thus, we can view each algorithm as making transitions from one distinguished control point to another, controlled only by the essential state.

There is a natural correspondence χ between the distinguished control points of FPI and FPR. If z is the distinguished control point of FPI immediately preceding the SEND in line 6, then $\chi(z)$ is the distinguished control point of S immediately preceding the SEND in line 7. If z is the distinguished control point of FPI immediately preceding the SEND in line 8, then $\chi(z)$ is the distinguished control point of S immediately preceding the SEND in line 10. For the other distinguished control points z of FPI, $\chi(z)$ is the (unique) control point of FPR which immediately precedes the same operation as the one immediately following z in FPI.

Let (z, q) be a determining state of FPI. We argued above that the next determining state (z', q') reached by FPI, starting at some state with determining part (z, q), is uniquely determined by (z, q). Similarly, the next determining state (z'', q'') reached by FPR, starting at some state with determining part $(\chi(z), q)$, is uniquely determined by $(\chi(z), q)$. It is tedious but straightforward to verify for all (z, q) that $(z'', q'') = (\chi(z'), q')$. It follows that the transitions on distinguished control points for both algorithms are the same when the distinguished control points from the two algorithms are identified by χ. The common transition table for the two algorithms is shown in Figure 10. The condition under which a given transition occurs is shown in the corresponding box of the table. An empty box indicates that the transition cannot occur.

The following theorem is an immediate consequence of the above discussion.

Theorem 5 *Algorithms* FPR *and* FPI *produce the same sequence of essential states and responses on the same sequence of requests.*

Corollary 1 *The bound of Theorem 4 on the number of comparisons applies also to* FPI.

5.3 A Conventional Presentation of the Iterative Algorithm

Algorithm FPI can easily be expressed as a collection of functions and procedures corresponding to the defining operations for a priority queue. They are shown in Figure 11. Initially, $k = 0$, $U = \emptyset$, and $b_1 = 0$.

6 Implementation with Sequential Storage

Until now, we have been measuring only the number of comparisons made. However, Fishspear can be implemented using sequential storage so that the total number of operations is at most a constant times the number of comparisons and priority queue operations. We describe such an implementation of the iterative algorithm FPI of Section 5.

Control point abbreviations

RECV	RECEIVE
SEND_1	SEND (in case 'empty')
DEL_S	DELETE_SHARP
SEND_2	SEND (in case 'delete')
BARB_C	BARB_CREATE
Set b_{k+1}	$b_{k+1} := \lceil \beta u \rceil$
PMERGE	PMERGE
BARB_D	BARB_DISPOSE

Test conditions

C_1	OP(x) = 'empty?'
C_2	OP(x) = 'delete'
C_3	OP(x) = 'insert'
C_4	$(k = 0) \vee ((w_k > 0) \wedge (v_k < u < b_k))$
C_5	$(k > 0) \wedge (w_k > 0) \wedge ((v_k \geq u) \vee (u \geq b_k))$
C_6	$(k > 0) \wedge (w_k = 0)$
C'_4	$(u = 0) \wedge C_4$
C'_5	$(u = 0) \wedge C_5$
C'_6	$(u = 0) \wedge C_6$

Transitions

	RECV	SEND_1	DEL_S	SEND_2	BARB_C	Set b_{k+1}	PMERGE	BARB_D
RECV		C_1	C_2		C_3			
SEND_1	true							
DEL_S				true				
SEND_2	$u \neq 0$					C'_4	C'_5	C'_6
BARB_C						C_4	C_5	C_6
Set b_{k+1}	true							
PMERGE						C_4	C_5	C_6
BARB_D						C_4	C_5	C_6

Figure 10: Transitions of algorithms FPR and FPI.

We use three pushdown stacks T, V, W in addition to a fixed number of integer registers. V and W are used to hold the fishspear segments of the same names, the segments being arranged so that the segments with highest indices are on top. Each segment is sorted according to $<$ on D, with the smallest element on top. Outside the procedure CLEAN, U is also sorted with smallest element on top and is held at the top of stack W. Thus, stack W holds the whole shaft in sorted order. When CLEAN is entered, $U = \emptyset$, and inside CLEAN, U is represented as a reverse-sorted list on T (with the largest element on top).

Registers k, u, v, and w record the number of barbs and the current lengths of U, V_k, and W_k, respectively, and are updated whenever the fishspear changes. The length of each other segment is stored on top of that segment on its stack and is copied to an appropriate register as required.

Operations are of three kinds. Integer operations have unit cost and include assignment, increment, decrement, comparison of two integers, and evaluation of $\lceil \beta u \rceil$ for integer u. Element operations also have unit cost and include value comparisons and assignments of one element variable to another. The four fishspear primitives are not unit cost and must be analyzed separately.

```
Function EMPTY?:
  return (u = 0)

Function DELETE_MIN:
  local variable x := DELETE_SHARP
  if u = 0 then CLEAN
  return x

Procedure INSERT(x):
  BARB_CREATE(x)
  CLEAN
```

Figure 11: A procedural version of algorithm FPI.

Most integer and element operations outside of the fishspear primitives can be associated with the next priority queue operation. The exceptions are those involved in iterations of the **while**-loop in CLEAN; these are associated with the next comparison made. Since at most a constant number of these operations is assigned to each priority queue operation or comparison, the total cost of these operations is $O(\text{Comp}(\mathbf{m}) + N)$, where N is the length (number of priority queue operations) of the run. We use N here, instead of n, only because of EMPTY? operations, which make no comparisons and do not change the queue.

We now consider the four fishspear primitives.

BARB_CREATE increments k, renames U as W_k, leaving it in place on stack W, and puts a new singleton barb on stack V.

BARB_DISPOSE merges together the top two barbs on stack V when $k > 1$. It copies these two barbs one each to stacks T and W, reversing them in the process, and then does the merge back on to stack V. The time for these operations is proportional to the length of the result, which is also the number of comparisons used by the merge. Hence, the cost is associated with the comparisons made by BARB_DISPOSE. If $k = 1$, it just appends the top barb on stack V to U. The time is proportional to the length of the barb, which is the number of (pseudo-)comparisons accounted for in our analysis.

PMERGE is a constant cost operation, for it simply compares the top elements of V and W and moves the smaller one to U, which is in reverse order on stack T. We associate the cost with the comparison made by PMERGE. (Recall that we are assuming a comparison is made even in the degenerate case that the barb is empty.)

DELETE_SHARP just removes the top element of U, which is at the top of stack W whenever this primitive is called.

Finally, when CLEAN is finished, it is necessary to copy and reverse U from stack T to the top of stack W. We associate the cost of copying each element with the comparison made by the PMERGE operation that put it in U.

Theorem 6 *For all β, $0 < \beta < 1$, Fishspear can be implemented so that on a run of length*

N with n insertions and max-depth profile $\mathbf{m}$*, the total number of operations is*

$$O\left(N + \sum_{i=1}^{n} \log m_i\right).$$

Proof: By carefully implementing the algorithm as described above, the total number of operations is bounded by

$$O(N + \mathrm{Comp}(\mathbf{m})).$$

The bound then follows immediately from Theorem 4. □

7 Conclusions

We have presented a priority queue algorithm with many novel properties in its implementation and complexity. There are several aspects which invite better understanding.

The complexity analysis presented here concentrated on proving a worst-case bound for the number of comparisons used and we chose a value for the tuning parameter β accordingly. We have no reason to suppose that this choice optimizes the worst-case number of comparisons. In any practical application the best choice for β would depend on the expected properties of the sequences of requests.

An unusual feature of the Fishspear algorithm is that it makes use of the values on the b-stack which record *past* information about the structure of the fishspear. This should not be too unexpected because the algorithm is designed to perform well with respect to the run profiles $\mathbf{m}$ and $\mathbf{h}$ introduced in Section 2. These values for elements currently in the queue also depend on past history. It would be of interest to know whether there are algorithms for manipulating the fishspear which require no other information than the fishspear itself, but still perform well with respect to the same or similar performance criteria.

We have implemented Fishspear for testing purposes but have not attempted to optimize the efficiency of the code. Its competitiveness vis-à-vis conventional priority queue algorithms would depend on details of the programming language and machine architecture. Our current impression is that the algorithm is very efficient in its management of the deeper levels of the priority queue, because it usually leaves these untouched for long periods and manipulates them in linear lists. A weakness of the algorithm is likely to be relatively inefficient performance at the "sharp end" of the queue, though this might be avoided by special code for that region. An attractive more general approach which we would like to explore is to combine a priority queue structure optimized for short queues, to handle shallow elements, with a fishspear to hold the deeper elements, perhaps in background storage. Such a hybrid could combine the strengths of each structure.

Acknowledgements

We are grateful to R. E. Tarjan, N. Immerman, and the anonymous referee for helpful comments. We thank T. C. Brown and E. M. Fischer for finding bugs in previous versions of the algorithm.

References

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

[2] S. Carlsson. A variant of HEAPSORT with almost optimal number of comparisons. *Information Processing Letters*, 24:247–250, 1987.

[3] G. H. Gonnet and J. I. Munro. Heaps on heaps. *SIAM J. Comput.*, 15:964–971, 1986.

[4] C. A. R. Hoare. Communicating sequential processes. *Comm. Assoc. Comput. Mach.*, 21(8):666–677, 1978.

[5] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. Assoc. Comput. Mach.*, 28(2):202–208, 1985.

[6] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. Assoc. Comput. Mach.*, 32(3):652–686, 1985.

[7] D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, 1986.

[8] J. W. J. Williams. Algorithm 232: Heapsort. *Comm. Assoc. Comput. Mach.*, 7(6):347–348, 1964.