Fishspear: A Priority Queue Algorithm

(EXTENDED ABSTRACT)

Michael J. Fischer
Yale University
New Haven, Connecticut

Michael S. Paterson
University of Warwick
Coventry, England

Abstract

The Fishspear priority queue algorithm is presented and analyzed. Fishspear makes fewer than 80% as many comparisons as heaps in the worst case, and its relative performance is even better in many common situations. The code itself embodies an unusual recursive structure which permits highly dynamic and data-dependent execution. Fishspear also differs from heaps in that it can be implemented efficiently using sequential storage such as stacks or tapes, making it possibly attractive for implementation of very large queues on paged memory systems. (Details of the implementation are deferred to the full paper.)

1 Introduction

A priority queue is an abstract data type consisting of a finite multiset $P$ over a linearly ordered universe $D$ together with the following operations:

- **MAKEEMPTY**: Sets $P := \emptyset$.
- **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 least element in $P$.

Priority queues find application in discrete event simulation, computational geometry, shortest path computations, and many other areas of computer science.

A simple implementation of priority queues keeps the elements in an ordered list. Insertions are performed by binary search and take $\lceil \log h \rceil$ comparisons to yield a list of size $h$, and the remaining operations take no comparisons. 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 there is no confusion, by a "heap" we mean a balanced binary tree with elements $z_i$ labelling each node $i$ such that for any nodes $i,j$, if $i$ is an ancestor of $j$, then $z_i \leq z_j$.

One of the first applications of heaps was to an algorithm for sorting $n$ items using $O(n \log n)$ comparisons [5]. Since $\Omega(n \log n)$ is a lower bound on the number of comparisons for sorting, it follows that the amortized cost 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.

Another intriguing property of heaps is that they exploit the ability to randomly access memory. The pattern of memory accesses is dynamically determined by the data, and there is no apparent way of implementing heaps while maintaining the logarithmic amortized operation cost 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.

\footnote{All logarithms are taken to the base 2 unless specified otherwise.}

\footnote{The amortized cost of a sequence of operations is the total cost of the sequence divided by the number of operations [8], [4].}
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 heaps in two senses which are made more precise in the next section. First of all, it has similar amortized efficiency to heaps in the worst case ($O(\log n)$ comparisons per queue operation), but the coefficient of $\log n$ is actually less (1.2 versus 1.5) on sequences that start and end with the queue empty. Secondly, the number of comparisons is "little-oh" of the number made by heaps 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 elements to be inserted are drawn randomly with uniform distribution from the unit interval, then the amortized number of comparisons made by heaps for each such pair is about $3 \log h$ ($\log h$ for the INSERT and $2 \log h$ for the DELETE.MIN), whereas the amortized cost for Fishspear is $O(1)$. The queue at any time during this procedure contains the $h$ largest elements ever inserted; hence, the size of the smallest of these approaches 1, so the probability that a newly-inserted element will be deleted by the very next operation also approaches 1. Fishspear is particularly efficient in this 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.

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 truly 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. We hope eventually to obtain experimental data to support such a claim.

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

Fishspear is similar to self-adjusting heaps [3] 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 that occurs only rarely. Important differences are that self-adjusting heaps support an additional operation, MELD, which Fishspear does not, but 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

We now look in some detail at how to measure the performance of priority queue algorithms.

The speed of sorting algorithms, for example, is often expressed in terms of the worst-case or average numbers 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. We 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.

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.

A performance measure we shall use is based on the sequence $h = h_1, \ldots, h_n$ denoting the size of the queue immediately after the insertion of each of $n$ elements. The sequence $h$ is called the size profile for that run of the priority queue, where by run 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 $h$, the usual heap implementation may use $\log h_j$ comparisons at the $j$th insertion and a corresponding $2\log h_j$ comparisons for that deletion which subsequently first takes the queue from size $h$ down to size $h - 1$. Hence an upper bound for the worst-case number of comparisons is approximately $3 \sum \log h_j$.

The comparisons for the naive list implementation are $\sum [\log h_j]$. As a lower bound, we have

**Theorem 1** The worst-case number of comparisons used by any priority queue algorithm on runs with size

---

3Details are deferred to the full paper.

4Here the parameter $n$ is the number of insertions, not the total length of the operation sequence.
profile \( h \) is at least
\[
\left\lceil \sum_{j=1}^{n} \log h_j \right\rceil.
\]

**Proof:** Consider all possible queue runs with size profile \( h \) and distinct input elements. The priority queue algorithm is required to determine the unique correct order of the output elements. Elements simultaneously in the queue are output in order, so each possible way of inserting a new element into the queue yields a distinct output sequence. There are \( h_j + 1 \) 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 \( \log \prod h_j = \left\lceil \sum \log h_j \right\rceil \) binary comparisons to distinguish among these runs. \( \square \)

Fix a run and let \( x_i \) be the \( i \)-th element inserted into the queue. Now consider any element \( y \) in the queue at a particular time. It will be convenient to associate with each such \( y \) a distinct \( i \) such that \( y = x_i \). If the \( x_i \) are all distinct, the association is obvious, but since we permit the queue to be a multiset, there may be more than one way to make the correspondence. For definiteness, if the queue contains \( k \) copies of \( y \) at a particular time \( r \), we associate those copies with the \( k \) largest elements of \( \{ i \leq n_r \mid y = x_i \} \), where \( n_r \) is the number of INSERT operations up to time \( r \). Implicit in our use of the notation "\( x_i \)" is that \( i \) is associated with the element \( x_i \), so we say "\( x_i \) is in the queue at time \( r \)" to mean that \( x_i \) is contained in the multiset at time \( r \) and is associated with index \( i \).

We now define a strong total ordering \( \preceq \) on the \( x_i \)'s. \( x_i \prec x_j \) if either \( x_i \prec x_j \), or \( x_i = x_j \) and \( i < j \). By the conventions of the preceding paragraph, it is clear that if \( x_i \prec x_j \) and \( x_i, x_j \) are simultaneously in the queue, then \( x_i \) will appear as output before \( x_j \).

The **depth** of \( x_i \) at a time when it is in the queue is one plus the number of elements \( x_j \prec x_i \) 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 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 which does not disturb the deeper elements unnecessarily.

For a more refined analysis of complexities, we may define the **max-depth profile** \( m \) for a run as the sequence \( m_1, m_2, \ldots \), where \( m_j \) is the maximum depth attained in the queue by element \( z_j \) during the run. While the usual heap implementations appear to derive no advantage when \( m \prec h \), 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), where the coefficient \( c \) is less than 2.4.

Less apparent is that the upper bound for Fishspear holds even if \( \langle m_j \rangle \) is replaced by \( \langle h_j \rangle \). Indeed, an individual element can attain depth in the queue much greater than the size of the queue when it was first inserted. Nevertheless, on the average, the \( m \)'s are no bigger than the \( h \)'s.

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

**Proof:** 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 \( \prec \) of all the elements. We consider the effect of interchanging \( x_i \) and \( x_j \) in the run.

If \( x_i \) leaves the queue before \( x_j \) enters, this interchange does not affect \( m_i \) or \( m_j \). If not, let \( M \) be the maximum depth attained by \( x_i \) before \( x_j \) enters and let \( M' \) and \( M'' \) be the maximum depths attained by \( x_i \) and \( x_j \) respectively after this time. Note that \( M' > M''. \) Before the interchange,

\[
m_i = \max\{M, M'\} \quad \text{and} \quad m_j = M'',
\]
while after

\[
m_i = \max\{M, M''\} \quad \text{and} \quad m_j = M'.
\]

We consider two cases and compare the pairs \( \langle m_i, m_j \rangle \) before and after the interchange.

1. \( M \leq M' \).
   Before: \( \langle M', M'' \rangle \). After: \( \langle \max\{M, M''\}, M' \rangle \).

2. \( M' \leq M \).
   Before: \( \langle M, M'' \rangle \). After: \( \langle M, M' \rangle \).

In each case the pair, regarded as a multiset, increases in value in one element or remains the same.

We can repeat this process wherever there is a pair of elements with adjacent values where the larger value is inserted first. The final result will be a "FIFO" run in which the elements are inserted in order of increasing value. For such a run, \( m_1 = h_1 \) since the initial depth of any element, which will be \( h_1 \), here, cannot be increased by subsequent insertions. Since each interchange on the way to constructing the "FIFO" run could only increase the values of \( \{m_1, m_2, \ldots\} \) as a multiset, the result follows at once. \( \square \)
3 The Fishspear Algorithm

The algorithm which we present in Section 3.2 is an instance of a general class of (non-deterministic) 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. The collection is partially ordered by the rule that $U < V$ if $z < y$ for every $z \in U$ and $y \in V$. A $k$-barbed fishspear consists of (possibly empty) segments $U, W_k, \ldots, W_1, V_k, \ldots, V_1$. Segments $U, W_k, \ldots, W_1$ are linearly ordered and form the shaft of the spear, that is, $U \leq W_k \leq W_{k-1} \leq \ldots \leq W_1$. Segments $V_k, \ldots, V_1$ are the barbs of the spear and satisfy $U \leq V_k, U \leq V_i, W_j \leq V_i$ for all $i, j$ with $k \geq j > i \geq 1$. A spear is illustrated in Figure 1.

| $U$ | $W_k$ | $W_{k-1}$ | $\ldots$ | $W_1$ |
|-----|-------|-----------|----------|
|     | $V_k$ | $V_{k-1}$ | $V_{k-2}$ | $V_1$ |

Figure 1: A k-barbed fishspear.

Five primitive operations can be performed on the data structure:

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 $V_k$ and appending the smaller one to $U$. (If $V_k$ is empty, the first element of $W_k$ is appended to $U$.)

BARB_MERGE: Assumes $k > 1$ and $W_k$ is empty. Merges $V_k$ into $V_{k-1}$, and sets $k := k - 1$. The result is a $(k - 1)$-barbed fishspear.

TOP_CAT: Assumes $k = 1$ and $W_1$ is empty. Appends $V_1$ to $U$ and sets $k := 0$. The result is a 0-barbed fishspear (i.e. the entire queue is sorted and resides in $U$).

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

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

In addition to the above, we assume the existence of basic operations for testing and comparing the lengths of the various segments. The priority queue operation EMPTY? is implemented by testing if all of the fishspear segments are empty, and MAKEEMPTY can be defined in terms of EMPTY? and DELETE_MIN. To do INSERT($x$), one merely performs BARB_CREATE($\{x\}$) on the fishspear data structure. To do 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:

if $U$ is empty then begin
  while $k > 1$ and $W_k$ is empty do BARB_MERGE;
  if $W_k$ is non-empty then
    PMERGE
  else TOP_CAT
end

Performing this code before every 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 $n$-fishspear with one element in each barb and an empty shaft. At the time of the first DELETE_MIN, the above code combines all $n$ bars in a series of unbalanced merges requiring $\Omega(n^2)$ comparisons.

3.2 A Particular Algorithm

The strategy of our algorithm is to selectively perform PMERGE, BARB_MERGE and TOP_CAT operations before each priority queue operation so as to maintain a kind of 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 a good 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
We assume two synchronized primitives for interprocess communication, SEND(m) and RECEIVE, where m is a message. (Cf. CSP [2].) A process executing RECEIVE blocks until the other process is ready to execute SEND(m) for some m, at which time the RECEIVE operation returns m as its value and both processes continue. Similarly, a process executing SEND(m) is forced to wait until the other process is ready to execute RECEIVE.

Messages are elements of \(D \cup \{'\text{del}', 'empty?'\} \cup \{'\text{yes}', 'no'\}\). A message in \(D\) denotes an element to be inserted, if sent by the user process, or the minimum element just deleted from the queue, if sent by Q. Messages 'del' and 'empty?' are requests by the user process to perform a DELETE_MIN or EMPTY? operation on the priority queue. 'yes' and 'no' are responses by Q to the 'empty?' request. We assume the user process performs RECEIVE immediately following each SEND('empty?') and SEND('del') request in order to receive the response.

Q maintains two pieces of global data—an integer \(k\) and a \(k\)-fishspear stored in variables \(U, V_j, W_j, j \geq 0\), as described above. All of the manipulations of this data are performed by the five fishspear primitives, which are invoked by Q.

The heart of the algorithm is the recursive procedure S. When S is called, \(U\) is assumed to be non-empty. S performs one or more RECEIVE operations, carries out the actions specified by the messages received, responds to each 'del' or 'empty?' request by issuing an a SEND with the answer, and modifies the fishspear to reflect the changes in the queue contents. When S eventually returns, the length \(k\) of the fishspear is one greater than when it was called, and \(W_k = \emptyset\).

The code for S is given in Figure 3. \(\beta\) is a tuning parameter. We are able to prove the best worst-case bounds for \(\beta = 0.7034\), but any value between 0 and 1 yields a correct algorithm. In this program, and elsewhere in this paper, we follow 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. \(u_0 := u\)
2. BASE
3. \(\text{while } w_k > 0 \text{ do}
   4. \(\text{if } v_k \geq u \text{ or } u \geq \beta u_0 \text{ then } \text{PMERGE}\)
   5. \(\text{else } \{v_k < u\} \text{ begin}
   6. \(S; \text{BARB-MERGE}\)
   7. \(\text{end}\)

**Procedure BASE:**
1. \(\text{repeat}
2. \(z := \text{RECEIVE}\)
3. \(\text{if } z = '\text{empty}' \text{ then } \text{SEND 'no'}\)
4. \(\text{else if } z = '\text{del}' \text{ then } \text{SEND DELETE-SHARP}\)
5. \(\text{until } x \in D \text{ or } u = 0\)
6. \(\text{if } x \in D \text{ then } \text{BARB-CREATE}\{\{x\}\}\)
7. \(\text{else BARB-CREATE}\{\emptyset\}\)

**Procedure S:**
1. \(U, := U\)
2. \(x := \text{RECEIVE}\)
3. \(\text{if } x = '\text{empty}' \text{ then } \text{SEND 'no'}\)
4. \(\text{else if } x = '\text{del}' \text{ then } \text{SEND DELETE-SHARP}\)
5. \(\text{until } x \in D \text{ or } u = 0\)
6. \(\text{if } x \in D \text{ then } \text{BARB-CREATE}\{\{x\}\}\)
7. \(\text{else BARB-CREATE}\{\emptyset\}\)

Finally, we give the top-level code for process Q which runs the priority queue algorithm by repeatedly calling S. Since S can only be called when the fishspear is non-empty, Q itself reads and processes messages whenever the queue is empty.

**4 Complexity Analysis**

We present an upper bound on the worst-case number of comparisons, Comp(m), made by fishspear on an input sequence with max-depth profile m.
Theorem 3 For all $\beta$, $0 < \beta < 1$, there exist $c$, $c'$ such that for all runs with $n$ insertions and max-depth profile $m$, 

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

In particular, for $\beta = .7034$, we may take $c = 2.4$.

(Further details on the interdependence of $c$, $c'$ and $\beta$ are given in the analysis below.)

The proof consists of several parts. First, we classify each comparison made by the algorithm 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 $c' \log m_i + c''n$ tolls. Summing over all the elements gives

$$
\# \text{ Type II comparisons} \leq \text{tolls collected} \leq c \sum_{i=1}^{n} \log m_i + c''n.
$$

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 first entering the shaft of the fish spear is of Type I; all other comparisons are Type II. An examination of the algorithm shows that there are only two places in which elements are compared: within the PMERGE of line 4 of $S$, and within the BARB_MERGE of line 6 of $S$. PMERGE compares the first element of $V_k$ with the first element of $W_k$ and appends the smaller (higher priority) 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 $W_k$. All comparisons made by BARB_MERGE are of Type II, since no elements enter the shaft.

Lemma 1 The algorithm makes at most $n$ Type I comparisons.

Proof: Once an element enters the shaft, it remains there until eventually deleted from the queue. Hence, at most $n$ Type I comparisons are made in the course of the algorithm since each element enters the shaft only once.

4.2 The Progress Lemma

We now take a more detailed look at the recursive structure of the algorithm and the actions which it performs. We first introduce some notation to allow us to talk about the way the fish spear changes over time. At any time $t$, let $U_t$ be the set of elements in segment $U$, let $V_t$ be the set of elements in segment $V_k$, let $V'_t$ be the set of elements in segment $V_{k-1}$, assuming $k > 1$ at that time, and let $W_t$ be the set of elements in $W_k$. These definitions depend on the current value of $k$, so in particular, $V_t$ always refers to the top barb of the fish spear, and $V'_t$ always refers to the second-from-top barb. As usual, the corresponding lower case letter refers to the cardinality of the set, so $u_t = |U_t|$, etc.

Now consider a single instance of a call on $S$ and the computation that takes place between the time $\alpha$ of the call and the time $\omega$ of the return. Let $\alpha'$ be the time just before line 3 of $S$ is executed for the first time, and let $\tau$ be a time at which control is between lines of $S$ such that $\alpha' \leq \tau \leq \omega$. We define the following sets of elements:

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

We often omit the subscript $\tau$ when $\tau$ is clear from context. The relationships that exist among these sets...
are given in Figure 6 and are easily proved by induction on \( r \), for \( r \) between \( a' \) and \( \omega \).

**Lemma 2** *(Progress Lemma)* Let \( r \) be any time, \( a' \leq r \leq \omega \), such that the test \( u \geq \beta u_\alpha \) in line 4 of \( S \) has never evaluated to 'true' anytime during the interval from \( a' \) to \( r \), and control is between lines of \( S \). Then

\[
v_r \geq u_\alpha^{old} - 1.
\]

**Proof:** To begin with, observe that if the condition \( u \geq \beta u_\alpha \) once becomes true, then it remains true for the duration of that execution of \( S \), for as long as it is true, the 'then' branch of the condition in line 4 is always taken, and PMERGE does not change \( u_\alpha \) nor decrease \( u \).

We proceed to prove the lemma. At time \( r = a' \), \( U \) is empty, so \( u_\alpha^{old} = 0 \) and the lemma holds. Subsequently, the only places where \( U \) or \( V \) are modified are in lines 4 and 6 of \( S \). We consider them in turn.

Suppose \( r \) is a time just after the PMERGE in line 4 of \( S \) has been performed, and suppose the conditions of the lemma are satisfied at time \( r \). Then \( u_\alpha < \beta u_\alpha \), so \( v \geq u \geq u_\alpha^{old} \) just before the PMERGE. The PMERGE moves one element from either \( V_k \) or \( W \) into \( U \). If it moves an element from \( V_k \), then \( v \) decreases by 1 but \( u_\alpha^{old} \) remains unchanged (since \( V_k \) consists entirely of new elements). If it moves an element from \( W_k \), then \( u_\alpha^{old} \) increases by 1 but \( v \) remains unchanged. In either case, \( v \geq u_\alpha^{old} - 1 \) afterwards.

Now consider the effect of line 6 on \( U \) and \( V \). The recursive call on \( S \) modifies \( U \) and adds a new barb to the fish spear. The call on BARB.MERGE then merges the top two barbs together, leaving the fish spear with the same number of segments as it had before the recursive call. Line 6 can only decrease (or leave unchanged) the size of \( U \), 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), and BARB.MERGE does not affect \( U \). Line 6 can only increase (or leave unchanged) the size of \( V \), for its overall effect is to add to \( V \) those elements which the recursive call on \( S \) placed in the new barb, and these are all new elements inserted during the recursive call. Hence, lines 6 preserves the truth of the conclusion of the lemma. The lemma then follows by induction.

The following is a direct consequence of the Progress Lemma.

**Lemma 3** For any execution of \( S \), either

\[
in_\omega + out_\omega \geq u_\alpha - 1
\]

or

\[
in_\omega \geq \beta u_\alpha - 1.
\]

**Proof:** There are two cases, depending on whether the test \( u \geq \beta u_\alpha \) in line 4 of \( S \) ever evaluated to 'true'.

Case 1: The test never evaluated to 'true'. Then by Lemma 2, \( v_\alpha \geq u_\omega^{old} - 1 \). Also, \( u_\omega = 0 \) since the 'while' loop of line 3 terminated. Thus, using Figure 6, we see that \( in_\omega = u_\omega^{new} + v_\omega \) and \( out_\omega = u_\alpha - u_\omega^{old} \). Hence,

\[
in_\omega + out_\omega \geq u_\alpha + u_\omega^{new} - 1 \geq u_\alpha - 1.
\]

Case 2: The test first evaluated 'true' in an execution of line 4 which began at time \( r \). Then by Lemma 2, \( v_r \geq u_\omega^{old} - 1 \). From time \( r \) to \( \omega \), only PMERGE's are done, and no elements are deleted from the queue, so

\[
in_\omega = u_\omega^{new} + v_\omega = u_\omega^{new} + v_r.
\]

Hence,

\[
in_\omega \geq u_r^{old} - 1 = u_r - 1.
\]

Since the test was about to evaluate 'true', we have \( v_r \geq \beta u_\alpha \), so

\[
in_\omega \geq \beta u_\alpha - 1.
\]

**4.3 The Toll Economy**

We now describe our method of analyzing the number of Type II comparisons. We associate with each element inserted into the queue two infinite sets of tokens, the *in-tokens* and the *out-tokens*. The tokens in each set are numbered sequentially beginning with 1. In addition, each element has two *base-tokens*. The value of in-token (out-token) number \( d \) is \( t_d / d \) (\( l_a / d \)), and the value of the base token is \( t_d \), where \( t_1 \), \( t_\omega \), and \( t_d \) are positive constants to be specified later. They will depend on a parameter \( \delta \) which can be chosen arbitrarily from the open interval \( (0, \beta / 2) \).
We collect tolls by removing tokens from elements that are or were in the queue. The tolls collected $T$ is the total value of all tokens so taken. We ensure that any in-tokens and out-tokens taken satisfy the following:

**Tolling Rule**: The number $p$ of the highest numbered token collected from any element $z_i$ satisfies $p < \frac{(m_i + 1)\beta}{6}$.

We remark that for any set $X$ of elements simultaneously present in the queue and still possessing token $p$, the Tolling Rule lets us collect token $p$ from all but $\lfloor \delta p \rfloor - 1$ elements of $X$, for those elements all have depth at least $\delta p$. 

**Lemma 4**: Any manner of collecting tolls according to the Tolling Rule results in

$$T \leq 2t_B + (t_1 + t_0) \left[ \sum_{i=1}^{n} \frac{\ln m_i + \left( \frac{2e}{\delta} \right) \cdot n}{d} \right],$$

where $\ln x$ denotes the natural logarithm of $x$.

**Proof**: Since the largest token allowed by the tolling rule is at most $\lfloor (m_i + 1)/\delta \rfloor$, we have

$$T \leq 2t_B + (t_1 + t_0) \sum_{i=1}^{n} \frac{\left( 1 + \ln \frac{m_i + 1}{\delta} \right)}{d},$$

since $\sum_{i=1}^{n} \frac{1}{d} \leq 1 + \int_1^t \frac{1}{x} \, dx = 1 + \ln t$

$$\leq 2t_B + (t_1 + t_0) \sum_{i=1}^{n} \frac{2m_i e}{\delta},$$

$$= 2t_B + (t_1 + t_0) \left[ \sum_{i=1}^{n} \ln m_i \right] + \left( \frac{2e}{\delta} \right) \cdot n].$$

Fix a run of the queue. We will associate each token collected with a particular execution of $S$. Before describing exactly how this is done, we introduce a notation for naming such executions.

We define $S_\sigma$ inductively for certain strings $\sigma$ of positive integers. Let $i \geq 1$. $S_i$ denotes the execution of $S$ which results from the $i^{th}$ execution of line 10 of the top-level program $Q$, assuming $Q$ executes line 10 at least $i$ times in the run, and otherwise $S_i$ is undefined.

Inductively, suppose $\sigma$ is a string of natural numbers, and suppose $S_\sigma$ denotes an execution of $S$ which performs line 6 a total of $r$ times. Then $S_\sigma$ denotes the execution of $S$ which results from the $i^{th}$ execution of line 6 by $S_\sigma$, $1 \leq i \leq r$. $S_\sigma$ is undefined if $i > r$ or if $S_\sigma$ is undefined. Also, $S_\sigma$ is undefined, where $\epsilon$ denotes the empty string.

Let $\alpha(\sigma)$ and $\omega(\sigma)$ denote the endpoints of the time interval spanned by the execution $S_\sigma$. The interval of $S_\sigma$ contains in the interval of $S_\sigma'$ if $\sigma$ is a prefix of $\sigma'$, and the intervals are disjoint if neither $\sigma$ nor $\sigma'$ is a prefix of the other.

$S_\sigma$ is eligible to accept a token $t$ if the following conditions hold:

- $t$ is a base token of element $z_i$, and $z_i$ was inserted or deleted during the interval spanned by $S_\sigma$.
- $t$ is in-token number $p$ of element $z_i$, $z_i$ was inserted into the queue during the interval spanned by $S_\sigma$, and $p < \min \{u_{\alpha(\sigma)}(\sigma), (m_i + 1)/\delta\}$.
- $t$ is out-token number $p$ of element $z_i$, $z_i$ was deleted from the queue during the interval spanned by $S_\sigma$, and $p < \min \{u_{\alpha(\sigma)}(\sigma), (m_i + 1)/\delta\}$.

We associate $t$ with the lowest level execution which is eligible to accept it, that is, among the executions $S_\sigma$ eligible to accept $t$, we associate $t$ with the one for which the length of $\sigma$ is maximal. That this is unique follows from the fact that two distinct strings of the same length describe non-overlapping executions. If $t$ is associated with $S_\sigma$, we say that $t$ is collected by $S_\sigma$, or that $v$ tolls are taken by $S_\sigma$, where $v$ is the value of $t$ as defined above.

Looked at from another perspective, the following tokens are collected by $S_\sigma$ if permitted by the Tolling Rule:

- A base token from whatever element was inserted or deleted from the queue by the execution of BASE in line 2 of $S_\sigma$.
- In-tokens $u_{\alpha(\sigma)}(\sigma)$ through $u_{\alpha(\sigma)}(\sigma) - 1$ of element $z_i$ if $z_i$ was inserted in the queue during the $i^{th}$ execution of line 6 of $S_\sigma$.
- Out-tokens $u_{\alpha(\sigma)}(\sigma)$ through $u_{\alpha(\sigma)}(\sigma) - 1$ of element $z_i$ if $z_i$ was deleted from the queue during the $i^{th}$ execution of line 6 of $S_\sigma$.

This characterization holds because we assume $\beta < 1$, so the test in line 4 of $S$ then ensures that $u_{\alpha(\sigma)}(\sigma) < u_{\omega(\sigma)}$. Thus, if $p \geq u_{\alpha(\sigma)}$, it follows inductively that
Sui7 is not eligible to collect any token number p for any string γ.

In the remainder of this section, we assume that δ, β ∈ (0, 1), δ < β/2, and that t'1, t'O, tI, tO, tB are positive constants which satisfy the following:

\[ t'_1 \geq \max \left\{ \frac{2 + \beta}{\beta(1 - \ln \beta)} \cdot \frac{1}{\ln \beta} \right\} \]  
\[ t'_1 q - (t'_1 + t'O(1 - q)) \ln q - 2 - q \geq 0 \]  
holds if 0 < q < β.

\[ t'_1 \leq t_1 \left(1 - \frac{\delta}{\beta} \frac{t_1 + t'O}{t_1} \right) \]  
\[ t'O \leq t_1 \left(1 - \frac{\delta}{\beta} \frac{t_1 + t'O}{t_1} \right) \]  
\[ tB \geq t_1 (1 - \delta) \]  

Let T(σ) be the total value of all tokens collected by Sσ. We now derive a lower bound on T(σ).

**Lemma 5 (Tolls Lemma).** Let Sσ be an execution of S, and let α = α(σ) and ω = ω(σ). Then

\[ T(σ) \geq 2u_\alpha + u_\omega. \]

**Proof:** Consider the times α(σi), i = 1, 2, ..., immediately preceding the successive executions of line 6 during the while-loop of S. Let μ1 = α(σ1) and let μr+1 = α(σi) where i is the least number such that Sσi is defined and uα(σi) > uμr. Finally, let s be the largest index for which μs is defined. As a notational convenience, we write (j) for μj.

Each of IN(σj), INω, and OUTω are sets of elements which are simultaneously in the queue—the elements of IN(σj) are all present at time μj, the elements of INω are all there at time ω, and the elements of OUTω were all in the queue at time α. By the remark following the Tolling Rule, we can collect in-token number p from all but [δp] - 1 of the elements in IN(σj) or INω, for a total value of at least

\[ T_2(σ) = \sum_{j=1}^{s} \sum_{p=\alpha(σj)}^{u-1} t_1 (u(σj) - (δp - 1)) \frac{1}{p} \]  

Let u(σ) ≤ p ≤ uα - 1. By the remark following the Tolling Rule, in-token p is collected from all but [δp] - 1 of the elements in IN(σj) or INω, for a total value of at least

\[ T_3(σ) = \sum_{p=\alpha(σ)}^{u-1} t_1 (\max\{$ i\in\sigma, i_\omega \} - (δp - 1)) \frac{1}{p} \]  

Let u(σ) ≤ p ≤ uα - 1. By the remark following the Tolling Rule, out-token p is collected from all but [δp] - 1 of the elements in OUTω, for a total value of at least

\[ T_4(σ) = \sum_{p=\alpha(σ)}^{u-1} t_1 (out_\omega - (δp - 1)) \frac{1}{p} \]

Thus, T(σ) ≥ \[ \sum_{k=1}^{s} T_k(σ) \].

By Lemma 2 and Figure 6, in(σj) ≥ u(σj) - 1. Since also u(σj) > p in the summation, Equation 7 yields

\[ T_2(σ) \geq \sum_{j=1}^{s} t_1 (1 - \delta) u(σj) \sum_{p=\alpha(σj)}^{u(σj)-1} \frac{1}{p} \]

Using the fact that

\[ u(σj) \sum_{p=\alpha(σj)}^{u(σj)-1} \frac{1}{p} \geq u(σj) - u(σj-1), \]

we in turn get

\[ T_2(σ) \geq \sum_{j=1}^{s} t_1 (1 - \delta) \{ u(σj) - u(σj-1) \} = t_1 (1 - \delta) \{ u(σ) - 1 \} \]

By Lemma 2 and Figure 6, in(σj) ≥ u(σj) - 1, and by Lemma 3, we have βuα ≤ inω + outω + 1 ≤ max{u(σ), inω} + outω + 1. Using the fact that

\[ \sum_{p=\alpha(σ)}^{u-1} \frac{1}{p} \geq \ln u_\alpha \left( \frac{u(σ)}{u_\alpha} \right) \]

383
Equation 8 then yields

\[
T_3(\sigma) \geq t_1(\max\{\inw, \in\} - (\delta u_\alpha - 1)) \sum_{p=\inw}^{u_\alpha - 1} \frac{1}{p} \\
\geq \left[ t_1 \max\{\inw, \in\} + 1 \right] \delta - t_1 \beta \max\{\inw, \in\} + 1) \\
\geq t_1 \delta \left( \max\{\inw, \in\} + 1 \right) \ln \frac{u_\alpha}{u(\sigma)}. \tag{13}
\]

Also, Equation 9 yields

\[
T_4(\sigma) \geq t_4(\in\omega - (\delta u_\alpha - 1)) \sum_{p=\in\omega}^{u_\alpha - 1} \frac{1}{p} \\
\geq \left[ t_4(\in\omega + 1) \right] \delta - t_4 \beta \max\{\inw, \in\} + 1) \\
\geq t_4 \delta \left( \max\{\inw, \in\} + 1 \right) \ln \frac{u_\alpha}{u(\sigma)}. \tag{14}
\]

Combining Equations 13 and 14 with 3 and 4, we get

\[
T_3(\sigma) + T_4(\sigma) \geq \left[ t_1 \left( 1 - \frac{\delta (t_1 + t_4)}{t_1} \right) \max\{\inw, \in\} + 1 \right] \\
+ t_4 \left( 1 - \frac{\delta (t_1 + t_4)}{t_4} \right) \left( \max\{\inw, \in\} + 1 \right) \ln \frac{u_\alpha}{u(\sigma)} \\
\geq \left[ t_1 \max\{\inw, \in\} + 1 \right] \delta - t_1 \beta \max\{\inw, \in\} + 1) \\
+ t_4 \beta \left( \max\{\inw, \in\} + 1 \right) \ln \frac{u_\alpha}{u(\sigma)}. \tag{15}
\]

From Equation 3, we have \(t_1(1 - \delta) \geq t_1\). Thus, adding together Equations 6, 11, and 15, and using Equation 5, we get

\[
T(\sigma) \geq t_n + t_l(1 - \delta)(u(\sigma) - 1) \\
+ \left[ t_1 \max\{\inw, \in\} + 1 \right] \delta - t_1 \beta \max\{\inw, \in\} + 1) \\
\geq t_1 \max\{\inw, \in\} + 1 \right] \delta - t_1 \beta \max\{\inw, \in\} + 1) \\
+ t_4 \beta \left( \max\{\inw, \in\} + 1 \right) \ln \frac{u_\alpha}{u(\sigma)}. \tag{16}
\]

To complete the proof of the lemma, we show that

\[
t_1 \max\{\inw, \in\} + 1 \right] \delta - t_1 \beta \max\{\inw, \in\} + 1) \\
\geq 2u_\alpha + u_\omega.
\]

Let

\[
p = \frac{u(\sigma)}{u_\alpha}, \quad q = \frac{\in\omega + 1}{u_\alpha}, \quad \text{and} \quad r = \frac{\in\omega}{u_\alpha}.
\]

and define

\[
F = t_1^p - [t_1^q \max\{p, q\} + t_4 r] \ln p - 2 - q.
\]

It suffices to show \(F \geq 0\) since \(\in\omega + 1 \geq u_\omega\).

We make use of two constraints on \(p, q, r\). First of all, the test in line 4 of S ensures that \(u(\sigma) < \beta u_\alpha\), so \(p < \beta\). Secondly, Lemma 3 implies that either \(q + r \geq 1\) or \(q \geq \beta\).

Before proceeding, consider the partial derivative when \(p < q\):

\[
\frac{\partial F}{\partial p} = t_1^p - [t_1^q + t_4 r] \ln p - 2 - q
\]

\[
= t_1 \left( 1 - \frac{q}{p} \right) - t_4 r \ln p
\]

\[
< 0.
\]

This shows that \(F\) decreases as \(p\) increases to \(q\).

We now consider three cases depending on how \(q\) relates to \(p, \beta\).

Case 1: \(q \leq p < \beta\). Then \(q + r \geq 1\), so \(r \geq 1 - q \geq 1 - p\). Also, \(p < 1\) since \(\beta < 1\), so \(\ln p < 0\). Hence,

\[
F = t_1^p - [t_1^q + t_4 r] \ln p - 2 - q
\]

\[
\geq t_1^p - [t_1^q + t_4 r(1 - p)] \ln p - 2 - p.
\]

By Equation 2, \(F \geq 0\) as desired.

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

\[
F = t_1^p - [t_1^q + t_4 r] \ln p - 2 - q
\]

\[
\geq t_1^q - [t_1^q + t_4 r(1 - q)] \ln q - 2 - q.
\]

Again, Equation 2 gives \(F \geq 0\) as desired.

Case 3: \(p < \beta \leq q\). Again the partial derivative of \(F\) with respect to \(p\) is negative, so we can replace \(p\) by \(\beta\) and \(r\) by 0 to get

\[
F = t_1^p - [t_1^q + t_4 r] \ln p - 2 - q
\]

\[
\geq t_1^\beta - [t_1^q + t_4 r(1 - q)] \ln \beta - 2 - q
\]

\[
= [t_1^\beta - 2] - q[t_1^q \ln \beta + 1]. \tag{17}
\]

We now consider two subcases.
Subcase 1: $\beta \geq -2\ln \beta$. Then by Equation 1 we have $t'_1 \geq -1/\ln \beta \geq 2/\beta$. Hence,

$$F \geq \left[t'_1 \beta - 2 - q[t'_1 \ln \beta + 1] \right]$$

$$\geq \left[\left(\frac{2}{\beta}\right) \beta - 2 - q \left[\left(\frac{-1}{\ln \beta}\right) \ln \beta + 1 \right] \right]$$

$$= 0.$$

Subcase 2: $\beta < -2\ln \beta$. Then by Equation 1 we have $t'_1 \geq \frac{2 + \beta}{\beta(1 - \ln \beta)}$.

Hence,

$$t'_1 \ln \beta + 1 \leq \frac{(2 + \beta) \ln \beta + \beta(1 - \ln \beta)}{\beta(1 - \ln \beta)}$$

$$\leq \frac{2 \ln \beta + \beta}{\beta(1 - \ln \beta)} < 0.$$

Thus, using the assumption that $\beta \leq q$, Equations 17 and 18 give

$$F \geq \left[\left(\frac{2 + \beta - 2 + 2 \ln \beta}{(1 - \ln \beta)}\right) - \beta \left[\frac{2 \ln \beta + \beta}{\beta(1 - \ln \beta)}\right]\right]$$

$$= 0.$$

Thus, in all three cases, $F \geq 0$, completing the proof of the lemma.

We now relate the tolls collected to the comparisons made by the algorithm.

Let $\text{gain}(\sigma) = T(\sigma) - \text{type}_2(\sigma)$, where $\text{type}_2(\sigma)$ is the number of Type II comparisons made by $S_\sigma$ but excluding comparisons made by the subrecursive calls.

Lemma 6 Let $S_\sigma$ be an execution of $S$, and let $\alpha = \alpha(\sigma)$ and $\omega = \omega(\sigma)$. Then

$$\text{gain}(\sigma) \geq u_\alpha + v_\omega.$$  

Proof: Proof is by reverse induction on the length of $\sigma$, starting with the longest words $\sigma$ for which $S_\sigma$ is defined.

Suppose $S_\sigma$ is an execution of $S$ and the lemma has been proved for all executions $S_{\sigma'}$ with $\sigma$ a proper prefix of $\sigma'$. Consider the $i^{th}$ execution of line 6 of $S$ (which begins at time $\alpha(\sigma_i)$). The test in line 4 ensures $u_{\alpha(\sigma_i)} < u_{\omega(\sigma_i)}$. By induction, $\text{gain}(\sigma_i) \geq u_{\alpha(\sigma_i)} + v_{\omega(\sigma_i)}$. Hence, $\text{gain}(\sigma_i) \geq v_{\alpha(\sigma_i)} + v_{\omega(\sigma_i)}$. The number of comparisons made by BARB_MERGE in line 6 is at most $v_{\alpha(\sigma_i)} + v_{\omega(\sigma_i)}$, since it simply merges together the two segments $V'_{\alpha(\sigma_i)} = V_{\alpha(\sigma_i)}$ and $V_{\omega(\sigma_i)}$ in the straightforward way. Hence, the net gain of all of the executions of line 6 is non-negative.

We now consider the FMERGE in line 4. At most $u_\alpha$ Type II comparisons are made, since each such comparison removes an element from $W_\alpha$, and $W_\alpha$ initially (just after line 2) has size $u_\alpha$. By Lemma 5, $T(\sigma) \geq 2u_\alpha + v_\omega$. Hence, $\text{gain}(\sigma) \geq u_\alpha + v_\omega$ as desired.

Putting all this together gives us

Lemma 7 The total number of Type II comparisons made by Fishepear on a run with $n$ insertions and max-depth profile $m$ is at most

$$2t_B + (t_1 + t_0) \left[\sum_{i=1}^{n} \ln m_i + \left(\frac{2}{\delta}\right) \cdot n\right].$$

Proof: The run can be partitioned into segments of operations which are processed directly by $Q$ and segments which are processed by a top-level call on $S$. The former require no comparisons. That the total number required for the latter satisfies the bound in the lemma is an immediate consequence of Lemmas 4 and 6.

To complete the proof of Theorem 3, it is necessary to analyze the constants. First, note that for any $\delta, \beta \in (0, 1)$ with $\delta < \beta/2$, there exist values of $t'_1, t'_2, t_1, t_0, t_B$ which satisfy Equations 1-5. Use Equation 1 to define $t'_1$. The left hand side of Equation 2 as a function of $q$ is bounded from below over the interval $(0, \beta)$, and as a function of $t'_1$, it is linear with a positive coefficient that is bounded away from zero. It follows that Equation 2 is satisfied for sufficiently large $t'_1$. Similarly, Equations 3 and 4 can be satisfied by taking $t_1 = t_0$ sufficiently large, for then $2\delta/\beta < 1$ and the right hand sides are linear in $t_1 = t_0$ with positive coefficient. Finally, Equation 5 can be used to define $t_B$. The constant $c$ of Theorem 3 is given by

$$c = \{t_1 + t_0\} \cdot \ln 2.$$  

and one can take

$$c' = 2t_B + 1 + (t_1 + t_0) \cdot \ln \frac{2c}{\delta}.$$  

We get our best bounds by choosing $\beta = -2\ln \beta = .7034...$. Plugging in to Equation 1 yields $t'_1 = 2.843...$. Calculus together with numerical evaluation shows that $t'_1 = .5674...$ satisfies Equation 2, and equality holds (to within the limits of our precision) for $q = .141...$. (The function of Equation 2 over the interval $(0, \beta)$ is shown in Figure 7.) Thus, $t'_1 + t_0 = 3.410...$. By choosing $\delta$ sufficiently close to
0, we can make \( t_1 + t_0 \) arbitrarily close to 3.410... Finally, plugging into Equation 19 shows that the constant \( c \) of Theorem 3 can be chosen arbitrarily close to
\[
\ln(2) \times 3.410... = 2.363...
\]
In particular, \( c = 2.4 \) works.

Figure 7: The function \( t'_1 q - (t'_1 q + t'_0 (1 - q)) \ln q - 2 - q \) for \( t'_1 = 2.844 \) and \( t'_0 = 0.5675 \).

**Acknowledgement**

We are grateful to T. C. Brown of the University of Warwick for drawing our attention to a bug in a previous version of the algorithm and to Neil Immerman of Yale University for helpful discussions.

**Bibliography**