Yale University
Department of Computer Science

FISHSPEAR: A PRIORITY QUEUE ALGORITHM
(Extended Abstract)

Michael J. Fischer and Michael S. Paterson

YALEU/DCS/RR-333
August 1984
(reformatted and reissued August 2009)

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.[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 there is no confusion, by a "heap" we mean a balanced binary tree with elements x_i labelling each node i such that for any nodes i, j, if i is an ancestor of j, then $x_i \leq x_j$.

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[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.

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 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

[†]This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, and by the National Science Foundation under Grant MCS-8116678.

[1]All 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 [3], [4].

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.[3] 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.

[3]Details are deferred to the full paper.

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 $\mathbf{h} = h_1, \ldots, h_n$ denoting the size of the queue immediately *after* the insertion of each of n elements.[4] The sequence $\mathbf{h}$ 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 $\mathbf{h}$, the usual heap implementation may use $\log h_j$ comparisons at the jth insertion and a corresponding $2 \log h_j$ comparisons for that 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 $\sum \lceil \log h_j \rceil$. As a lower bound, we have

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

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

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

Proof: Consider all possible queue runs with size profile $\mathbf{h}$ 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 $h_{j-1}+1=h_j$ 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. ∎

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 τ, we associate those copies with the k largest elements of $\{i \le n_\tau \mid y=x_i\}$, where n_τ is the number of INSERT operations up to time τ. 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 τ" to mean that x_i is contained in the multiset at time τ and is associated with index i.

We now define a strong total ordering $\prec$ on the x_i's. $x_i \prec x_j$ if either $x_i < x_j$, 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* $\mathbf{m}$ for a run as the sequence $m_1, m_2, \ldots$, where m_j is the maximum depth attained in the queue by element 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), where the coefficient c is less than 2.4.

Less apparent is that the upper bound for Fishspear holds even if "m_j" is replaced by "h_j". 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* $\mathbf{m}$ *and size profile* $\mathbf{h}$. *There exists a permutation* π *such that* $m_i \le h_{\pi(i)}$ *for all* i, $1 \le i \le n$.

Proof: Suppose there is some pair i, j with $i<j$ and $x_i \succ x_j$, 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,

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

while after

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

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

1. $M \le M'$.
   Before: $\langle M', M'' \rangle$. After: $\langle \max\{M, M''\}, M' \rangle$.
2. $M' \le M$.
   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, $m_i = h_i$ since the initial depth of any element, which will be h_i 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. ∎

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 $U \leq V$ iff $x \leq y$ for every $x \in U$ and $y \in V$. A *k-barbed fishspear* consists of (possibly empty) segments $U, W_k, \ldots, W_1$ and $V_k, \ldots, V_1$. Segments $U, W_k, \ldots, W_1$ are linearly ordered and form the *shaft* of the spear, that is, $U \leq W_k \leq W_{k-1} \leq ... \leq W_1$. Segments $V_k, \ldots, V_1$ are the *barbs* of the spear and satisfy $U \leq V_k$, $U \leq V_i$ and $W_j \leq V_i$ for all i, j with $k \geq j > i \geq 1$. A spear is illustrated in Figure 1.

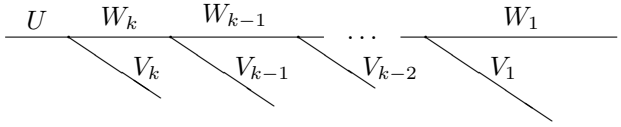

Figure 1: A k-barbed fishspear.

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 $W_{k+1} := U$, $U :=$ 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 barbs 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 DELETE_MIN that are defined recursively but rather a process Q which runs autonomously, alternately massaging the fishspear and processing priority-queue op-

erations. In other words, we regard Q 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. Q is separate from the "user" process which is issuing the priority queue operations, although Q could be implemented as a coroutine just as well. This view is illustrated in Figure 2.

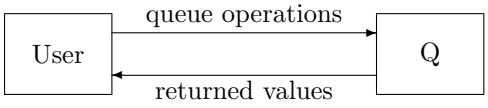

Figure 2: Process structure of the Fishspear algorithm.

We assume two synchronized primitives for interprocess communication, SEND(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 $D \cup \{\text{'del'}, \text{'empty?'}\} \cup \{\text{'yes'}, \text{'no'}\}$. 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, and 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. β is a tuning parameter. We are able to prove the best worst-case bounds for $\beta = 0.7034\ldots$, 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_α := u
2.  BASE
3.  while w_k > 0 do
4.    if v_k ≥ u or u ≥ βu_α then PMERGE
5.    else {v_k < u} begin
6.      S; BARB_MERGE
7.      end
```

Figure 3: The recursive procedure S.

The actual processing of messages takes place in the routine BASE, which is given in Figure 4. When BASE is called, U is assumed to be non-empty. BASE processes messages until either a new element is inserted into the queue or U becomes empty. In either case, BASE calls BARB_CREATE just before returning, so the resulting fishspear is one longer than at the time of call.

```
Procedure BASE:
1.  repeat
2.    x := RECEIVE
3.    if x = 'empty?' then SEND 'no'
4.    else if x = 'del' then SEND DELETE_SHARP
5.  until x ∈ D or u = 0
6.  if x ∈ D then BARB_CREATE({x})
7.  else BARB_CREATE(∅)
```

Figure 4: Code to process queue operations.

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($\mathbf{m}$), made by fishspear on an input sequence with max-depth profile $\mathbf{m}$.

Theorem 3 *For all β, $0 < \beta < 1$, there exist c, c' such that for all runs with n insertions and max-depth profile $\mathbf{m}$,*

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

```
Process Q:
1.   k := 0; U := ∅
2.   repeat forever
3.     if u = 0 then begin
4.       x := RECEIVE
5.       if x = 'empty?' then SEND 'yes'
6.       else if x = 'del' then error
7.       else BARB_CREATE({x})
8.       end
9.     else begin
10.      S
11.      end
12.    TOP_CAT
```

Figure 5: The top-level driver.

In particular, for $\beta = .7034$, 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 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 $c \log m_i + c''$ tolls. Summing over all the elements gives

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

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 first 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 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 fishspear changes over time. At any time τ, let U_τ be the set of elements in segment U, let V_τ be the set of elements in segment V_k, let V'_τ be the set of elements in segment V_{k-1}, assuming $k > 1$ at that time, and let W_τ be the set of elements in W_k. These definitions depend on the current value of k, so in particular, V_τ always refers to the top barb of the fishspear, and V'_τ always refers to the second-from-top barb. As usual, the corresponding lower case letter refers to the cardinality of the set, so $u_\tau = |U_\tau|$, etc.

Now consider a single instance of a call on S and the computation that takes place between the time α of the call and the time ω of the return. Let α' be the time just before line 3 of S is executed for the first time, and let τ 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_τ	=	set of elements inserted into the queue after time α and still present in the queue at time τ;
OUT_τ	=	set of elements present in the queue at time α but gone from the queue by time τ;
U_τ^{old}	=	$U_\tau \cap U_\alpha$, the set of old elements in U at time τ;
U_τ^{new}	=	$U_\tau \cap IN_\tau$, the set of new elements in U at time τ.

We often omit the subscript τ when τ is clear from context. The relationships that exist among these sets are given in Figure 6 and are easily proved by induction on τ, for τ between α' and ω.

Lemma 2 *(Progress Lemma) Let τ be any time, $\alpha' \leq \tau \leq \omega$, such that the test $u \geq \beta u_\alpha$ in line 4 of* S *has*

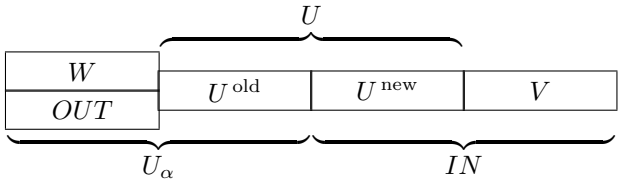

Figure 6: Relations among the basic sets after time α'.

never evaluated to 'true' anytime during the interval from α' to τ, and control is between lines of S. *Then*

$$v_\tau \geq u_\tau^{\text{old}} - 1.$$

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_α nor decrease u.

We proceed to prove the lemma. At time $\tau = \alpha'$, U is empty, so $u_\tau^{\text{old}} = 0$ 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 τ 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 τ. Then $u_\tau < \beta u_\alpha$, so $v \geq u \geq u^{\text{old}}$ just before the PMERGE. The PMERGE moves one element from either V_k or W_k into U. If it moves an element from V_k, then v decreases by 1 but u^{old} remains unchanged (since V_k consists entirely of new elements). If it moves an element from W_k, then u^{old} increases by 1 but v remains unchanged. In either case, $v \geq u^{\text{old}} - 1$ 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 fishspear. The call on BARB_MERGE then merges the top two barbs together, leaving the fishspear 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^{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), 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, $v_\omega \geq u_\omega^{\text{old}} - 1$. Also, $w_\omega = 0$ since the 'while' loop of line 3 terminated. Thus, using Figure 6, we see that $in_\omega = u_\omega^{\text{new}} + v_\omega$ and $out_\omega = u_\alpha - u_\omega^{\text{old}}$. Hence,

$$\begin{aligned} in_\omega + out_\omega &\geq u_\alpha + u_\omega^{\text{new}} - 1 \\ &\geq u_\alpha - 1. \end{aligned}$$

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

$$in_\omega = u_\omega^{\text{new}} + v_\omega = u_\tau^{\text{new}} + v_\tau.$$

Hence,

$$in_\omega \geq u_\tau^{\text{new}} + u_\tau^{\text{old}} - 1 = u_\tau - 1.$$

Since the test was about to evaluate 'true', we have $u_\tau \geq \beta u_\alpha$, 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_{I}/d (t_{O}/d), and the value of the base token is t_{B}, where t_{I}, t_{O}, and t_{B} are positive constants to be specified later. They will depend on a parameter δ 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 x_i satisfies $p < (m_i + 1)/\delta$.

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 $\lfloor \delta p \rfloor > \delta p - 1$.

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

$$T \leq 2t_{\mathrm{B}} + (t_{\mathrm{I}} + t_{\mathrm{O}}) \left[\sum_{i=1}^{n} \ln m_i + \left(\ln \frac{2e}{\delta} \right) \cdot n \right],$$

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

$$\begin{aligned} T &\leq 2t_{\mathrm{B}} + (t_{\mathrm{I}} + t_{\mathrm{O}}) \sum_{i=1}^{n} \sum_{d=1}^{\lfloor (m_i+1)/\delta \rfloor} \frac{1}{d} \\ &\leq 2t_{\mathrm{B}} + (t_{\mathrm{I}} + t_{\mathrm{O}}) \sum_{i=1}^{n} \left(1 + \ln \frac{m_i + 1}{\delta} \right) \\ &\qquad \text{since} \sum_{j=1}^{\lfloor t \rfloor} \frac{1}{j} \leq 1 + \int_1^t \frac{1}{x}\, dx = 1 + \ln t \\ &\leq 2t_{\mathrm{B}} + (t_{\mathrm{I}} + t_{\mathrm{O}}) \sum_{i=1}^{n} \ln \frac{2m_i e}{\delta} \\ &= 2t_{\mathrm{B}} + (t_{\mathrm{I}} + t_{\mathrm{O}}) \left[\left(\sum_{i=1}^{n} \ln m_i \right) + \left(\ln \frac{2e}{\delta} \right) \cdot n \right]. \end{aligned}$$

∎

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_σ inductively for certain strings σ 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 σ is a string of natural numbers, and suppose S_σ denotes an execution of S which performs line 6 a total of r times. Then $\mathrm{S}_{\sigma i}$ denotes the execution of S which results from the i^{th} execution of line 6 by S_σ, $1 \leq i \leq r$. $\mathrm{S}_{\sigma i}$ is undefined if $i > r$ or if S_σ is undefined. Also, S_e is undefined, where e denotes the empty string.

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

S_σ is *eligible* to accept a token t if the following conditions hold:

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

We associate t with the lowest level execution which is eligible to accept it, that is, among the executions S_σ eligible to accept t, we associate t with the one for which the length of σ 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_σ, we say that t is *collected* by S_σ, or that v *tolls* are taken by S_σ, where v is the value of t as defined above.

Looked at from another perspective, the following tokens are collected by S_σ 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_σ.
- In-tokens $u_{\alpha(\sigma i)}$ through $u_{\alpha(\sigma)} - 1$ of element x if x was inserted in the queue during the i^{th} execution of line 6 of S_σ.
- Out-tokens $u_{\alpha(\sigma i)}$ through $u_{\alpha(\sigma)} - 1$ of element x if x was deleted from the queue during the i^{th} execution of line 6 of S_σ.

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

In the remainder of this section, we assume that $\delta, \beta \in (0, 1)$, $\delta < \beta/2$, and that $t'_{\mathrm{I}}, t'_{\mathrm{O}}, t_{\mathrm{I}}, t_{\mathrm{O}}, t_{\mathrm{B}}$ are positive constants which satisfy the following:

$$t'_{\mathrm{I}} \geq \max \left\{ \frac{2 + \beta}{\beta(1 - \ln \beta)}, \frac{1}{-\ln \beta} \right\}. \tag{1}$$

$$t'_{\mathrm{I}}q - (t'_{\mathrm{I}}q + t'_{\mathrm{O}}(1-q))\ln q - 2 - q \geq 0 \qquad (2)$$

holds if $0 < q < \beta$.

$$t'_{\mathrm{I}} \leq t_{\mathrm{I}}\left(1 - \frac{\delta}{\beta}\frac{(t_{\mathrm{I}} + t_{\mathrm{O}})}{t_{\mathrm{I}}}\right). \qquad (3)$$

$$t'_{\mathrm{O}} \leq t_{\mathrm{O}}\left(1 - \frac{\delta}{\beta}\frac{(t_{\mathrm{I}} + t_{\mathrm{O}})}{t_{\mathrm{O}}}\right). \qquad (4)$$

$$t_{\mathrm{B}} \geq t_{\mathrm{I}}(1-\delta). \qquad (5)$$

Let $T(\sigma)$ be the total value of all tokens collected by S_σ. We now derive a lower bound on $T(\sigma)$.

Lemma 5 *(Tolls Lemma). Let S_σ be an execution of* S*, and let $\alpha = \alpha(\sigma)$ and $\omega = \omega(\sigma)$. Then*

$$T(\sigma) \geq 2u_\alpha + v_\omega.$$

Proof: Consider the times $\alpha(\sigma i)$, $i = 1, 2, \ldots$ immediately preceding the successive executions of line 6 during the while-loop of S. Let $\mu_1 = \alpha(\sigma 1)$ and let $\mu_{r+1} = \alpha(\sigma i)$ where i is the least number such that $\mathrm{S}_{\sigma i}$ is defined and $u_{\alpha(\sigma i)} > u_{\mu_r}$. Finally, let s be the largest index for which μ_s is defined. As a notational convenience, we write $\langle j \rangle$ for μ_j.

Each of $IN_{\langle j \rangle}$, IN_ω, and OUT_ω are sets of elements which are simultaneously in the queue—the elements of $IN_{\langle j \rangle}$ are all present at time μ_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. By the remark following the Tolling Rule, we can collect in-token number p from all but $\lfloor \delta p \rfloor - 1$ of the elements in $IN_{\langle j \rangle}$ or IN_ω. Similarly, we can collect out-token p from all but $\lfloor \delta p \rfloor - 1$ of the elements of OUT_ω.

We now total up the tokens we know are collected by S_σ, thereby giving a lower bound on $T(\sigma)$.

1. At least one base token is collected by S_σ since the call on BASE in line 2 of S causes at least one element to be inserted or deleted. It has value
$$T_1(\sigma) = t_{\mathrm{B}}. \qquad (6)$$

2. Let $1 \leq j \leq s$ and let $u_{\langle j-1 \rangle} \leq p \leq u_{\langle j \rangle} - 1$. (For technical convenience, we take $u_{\langle 0 \rangle} = 1$.) By the remark following the Tolling Rule, in-token number p is collected from all but $\lfloor \delta p \rfloor - 1$ of the elements in $IN_{\langle j \rangle}$ for a total value of at least $t_{\mathrm{I}}(in_{\langle j \rangle} - (\delta p - 1))/p$. Summing over j and p gives a total value of
$$T_2(\sigma) = \sum_{j=1}^{s} \sum_{p=u_{\langle j-1 \rangle}}^{u_{\langle j \rangle}-1} t_{\mathrm{I}}(in_{\langle j \rangle} - (\delta p - 1))\frac{1}{p} \qquad (7)$$

3. Let $u_{\langle s \rangle} \leq p \leq u_\alpha - 1$. By the remark following the Tolling Rule, in-token p is collected from all but $\lfloor \delta p \rfloor - 1$ of the elements in $IN_{\langle s \rangle} \cup IN_\omega$ for a total value of at least
$$T_3(\sigma) = \sum_{p=u_{\langle s \rangle}}^{u_\alpha - 1} t_{\mathrm{I}}\big(\max\{in_{\langle s \rangle}, in_\omega\} - (\delta p - 1)\big)\frac{1}{p} \qquad (8)$$

4. Let $u_{\langle s \rangle} \leq p \leq u_\alpha - 1$. By the remark following the Tolling Rule, out-token p is collected from all but $\lfloor \delta p \rfloor - 1$ of the elements in OUT_ω for a total value of at least
$$T_4(\sigma) = \sum_{p=u_{\langle s \rangle}}^{u_\alpha - 1} t_{\mathrm{O}}\big(out_\omega - (\delta p - 1)\big)\frac{1}{p} \qquad (9)$$

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

By Lemma 2 and Figure 6, $in_{\langle j \rangle} \geq u_{\langle j \rangle} - 1$. Since also $u_{\langle j \rangle} > p$ in the summation, Equation 7 yields

$$T_2(\sigma) \geq \sum_{j=1}^{s} t_{\mathrm{I}}(1-\delta)u_{\langle j \rangle} \sum_{p=u_{\langle j-1 \rangle}}^{u_{\langle j \rangle}-1} \frac{1}{p}. \qquad (10)$$

Using the fact that

$$u_{\langle j \rangle} \sum_{p=u_{\langle j-1 \rangle}}^{u_{\langle j \rangle}-1} \frac{1}{p} \geq u_{\langle j \rangle} - u_{\langle j-1 \rangle},$$

we in turn get

$$\begin{aligned} T_2(\sigma) &\geq \sum_{j=1}^{s} t_{\mathrm{I}}(1-\delta)(u_{\langle j \rangle} - u_{\langle j-1 \rangle}) \\ &= t_{\mathrm{I}}(1-\delta)(u_{\langle s \rangle} - 1) \end{aligned} \qquad (11)$$

By Lemma 2 and Figure 6, $in_{\langle s \rangle} \geq u_{\langle s \rangle} - 1$, and by Lemma 3, we have $\beta u_\alpha \leq in_\omega + out_\omega + 1 \leq \max\{u_{\langle s \rangle}, in_\omega + 1\} + out_\omega + 1$. Using the fact that

$$\sum_{p=u_{\langle s \rangle}}^{u_\alpha - 1} \frac{1}{p} \geq \ln \frac{u_\alpha}{u_{\langle s \rangle}}, \qquad (12)$$

Equation 8 then yields

$$\begin{aligned} T_3(\sigma) &\geq t_{\mathrm{I}}\big(\max\{in_{\langle s \rangle}, in_\omega\} - (\delta u_\alpha - 1)\big) \sum_{p=u_{\langle s \rangle}}^{u_\alpha - 1} \frac{1}{p} \\ &\geq \Big[t_{\mathrm{I}} \max\{u_{\langle s \rangle}, in_\omega + 1\} \\ &\quad - t_{\mathrm{I}}\frac{\delta}{\beta}(\max\{u_{\langle s \rangle}, in_\omega + 1\}) \\ &\quad - t_{\mathrm{I}}\frac{\delta}{\beta}(out_\omega + 1)\Big] \ln \frac{u_\alpha}{u_{\langle s \rangle}}. \end{aligned} \qquad (13)$$

Also, Equation 9 yields

$$\begin{aligned} T_4(\sigma) &\geq t_{\mathrm{O}}\big(out_\omega - (\delta u_\alpha - 1)\big) \sum_{p=u_{\langle s\rangle}}^{u_\alpha - 1} \frac{1}{p} \\ &\geq \Big[t_{\mathrm{O}}(out_\omega + 1) \\ &\quad -t_{\mathrm{O}}\frac{\delta}{\beta}(\max\{u_{\langle s\rangle}, in_\omega + 1\}) \\ &\quad -t_{\mathrm{O}}\frac{\delta}{\beta}(out_\omega + 1)\Big] \ln \frac{u_\alpha}{u_{\langle s\rangle}}. \end{aligned} \tag{14}$$

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

$$\begin{aligned} &T_3(\sigma) + T_4(\sigma) \\ &\geq \left[t_{\mathrm{I}}\left(1 - \frac{\delta}{\beta}\frac{(t_{\mathrm{I}} + t_{\mathrm{O}})}{t_{\mathrm{I}}}\right)\max\{u_{\langle s\rangle}, in_\omega + 1\}\right. \\ &\quad \left. +t_{\mathrm{O}}\left(1 - \frac{\delta}{\beta}\frac{(t_{\mathrm{I}} + t_{\mathrm{O}})}{t_{\mathrm{O}}}\right)(out_\omega + 1)\right]\ln\frac{u_\alpha}{u_{\langle s\rangle}} \\ &\geq [t'_{\mathrm{I}}\max\{u_{\langle s\rangle}, in_\omega + 1\} \\ &\quad +t'_{\mathrm{O}}(out_\omega + 1)]\ln\frac{u_\alpha}{u_{\langle s\rangle}} \end{aligned} \tag{15}$$

From Equation 3, we have $t_{\mathrm{I}}(1-\delta) \geq t'_{\mathrm{I}}$. Thus, adding together Equations 6, 11, and 15, and using Equation 5, we get

$$\begin{aligned} &T(\sigma) \\ &\geq t_{\mathrm{B}} + t_{\mathrm{I}}(1-\delta)(u_{\langle s\rangle} - 1) \\ &\quad +[t'_{\mathrm{I}}\max\{u_{\langle s\rangle}, in_\omega + 1\} + t'_{\mathrm{O}}(out_\omega + 1)]\ln\frac{u_\alpha}{u_{\langle s\rangle}} \\ &\geq t'_{\mathrm{I}}u_{\langle s\rangle} + [t'_{\mathrm{I}}\max\{u_{\langle s\rangle}, in_\omega + 1\} \\ &\quad +t'_{\mathrm{O}}out_\omega]\ln\frac{u_\alpha}{u_{\langle s\rangle}}. \end{aligned} \tag{16}$$

To complete the proof of the lemma, we show that

$$\begin{aligned} &t'_{\mathrm{I}}u_{\langle s\rangle} + \big(t'_{\mathrm{I}}\max\{u_{\langle s\rangle}, in_\omega + 1\} + t'_{\mathrm{O}}out_\omega\big)\ln\frac{u_\alpha}{u_{\langle s\rangle}} \\ &\qquad \geq 2u_\alpha + v_\omega. \end{aligned}$$

Let

$$p = \frac{u_{\langle s\rangle}}{u_\alpha},\ q = \frac{in_\omega + 1}{u_\alpha},\ \text{and } r = \frac{out_\omega}{u_\alpha}.$$

and define

$$F = t'_{\mathrm{I}}p - [t'_{\mathrm{I}}\max\{p, q\} + t'_{\mathrm{O}}r]\ln p - 2 - q.$$

It suffices to show $F \geq 0$ since $in_\omega + 1 \geq v_\omega$.

We make use of two constraints on p, q, r. First of all, the test in line 4 of S ensures that $u_{\langle s\rangle} < \beta u_\alpha$, so $p < \beta$. Secondly, Lemma 3 implies that either $q + r \geq 1$ or $q \geq \beta$.

Before proceeding, consider the partial derivative when $p < q$:

$$\begin{aligned} \frac{\partial F}{\partial p} &= t'_{\mathrm{I}} - [t'_{\mathrm{I}}q + t'_{\mathrm{O}}r]\frac{1}{p} \\ &= t'_{\mathrm{I}}\left(1 - \frac{q}{p}\right) - t'_{\mathrm{O}}\frac{r}{p} \\ &< 0. \end{aligned}$$

This shows that F decreases as p increases to q.

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

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,

$$\begin{aligned} F &= t'_{\mathrm{I}}p - [t'_{\mathrm{I}}p + t'_{\mathrm{O}}r]\ln p - 2 - q \\ &\geq t'_{\mathrm{I}}p - [t'_{\mathrm{I}}p + t'_{\mathrm{O}}(1-p)]\ln p - 2 - p. \end{aligned}$$

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

$$\begin{aligned} F &= t'_{\mathrm{I}}p - [t'_{\mathrm{I}}q + t'_{\mathrm{O}}r]\ln p - 2 - q \\ &\geq t'_{\mathrm{I}}q - [t'_{\mathrm{I}}q + t'_{\mathrm{O}}(1-q)]\ln q - 2 - q. \end{aligned}$$

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 β and r by 0 to get

$$\begin{aligned} F &= t'_{\mathrm{I}}p - [t'_{\mathrm{I}}q + t'_{\mathrm{O}}r]\ln p - 2 - q \\ &\geq t'_{\mathrm{I}}\beta - t'_{\mathrm{I}}q\ln\beta - 2 - q \\ &= [t'_{\mathrm{I}}\beta - 2] - q[t'_{\mathrm{I}}\ln\beta + 1]. \end{aligned} \tag{17}$$

We now consider two subcases.

Subcase 1: $\beta \geq -2\ln\beta$. Then by Equation 1 we have $t'_{\mathrm{I}} \geq -1/\ln\beta \geq 2/\beta$. Hence,

$$\begin{aligned} F &\geq [t'_{\mathrm{I}}\beta - 2] - q[t'_{\mathrm{I}}\ln\beta + 1] \\ &\geq \left[\left(\frac{2}{\beta}\right)\beta - 2\right] - q\left[\left(\frac{-1}{\ln\beta}\right)\ln\beta + 1\right] \\ &= 0. \end{aligned}$$

Subcase 2: $\beta < -2\ln\beta$. Then by Equation 1 we have

$$t'_{\mathrm{I}} \geq \frac{2 + \beta}{\beta(1 - \ln\beta)}.$$

Hence,

$$\begin{aligned} t'_{\mathrm{I}} \ln \beta + 1 &\leq \frac{(2+\beta)\ln\beta + \beta(1-\ln\beta)}{\beta(1-\ln\beta)} \\ &\leq \frac{2\ln\beta + \beta}{\beta(1-\ln\beta)} \qquad (18) \\ &< 0. \end{aligned}$$

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

$$\begin{aligned} F &\geq \left[\frac{2+\beta-2+2\ln\beta}{(1-\ln\beta)}\right] - \beta\left[\frac{2\ln\beta+\beta}{\beta(1-\ln\beta)}\right] \\ &= 0. \end{aligned}$$

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 $\mathrm{gain}(\sigma) = T(\sigma) - \mathrm{type}_{II}(\sigma)$, where $\mathrm{type}_{II}(\sigma)$ is the number of Type II comparisons made by S_σ but *excluding* comparisons made by the subrecursive calls.

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

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

Proof: Proof is by reverse induction on the length of σ, starting with the longest words σ for which S_σ is defined.

Suppose S_σ is an execution of S and the lemma has been proved for all executions $\mathrm{S}_{\sigma'}$ with σ a proper prefix of σ'. Consider the i^{th} execution of line 6 of S (which begins at time $\alpha(\sigma i)$). The test in line 4 ensures $v_{\alpha(\sigma i)} < u_{\alpha(\sigma i)}$. By induction, $\mathrm{gain}(\sigma i) \geq u_{\alpha(\sigma i)} + v_{\omega(\sigma i)}$. Hence, $\mathrm{gain}(\sigma i) \geq v_{\alpha(\sigma i)} + v_{\omega(\sigma i)}$. 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 $V'_{\omega(\sigma i)} = V_{\alpha(\sigma i)}$ 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 PMERGE in line 4. At most u_α Type II comparisons are made, since each such comparison removes an element from W_k, and W_k initially (just after line 2) has size u_α. By Lemma 5, $T(\sigma) \geq 2u_\alpha + v_\omega$. Hence, $\mathrm{gain}(\sigma) \geq u_\alpha + v_\omega$ as desired. ∎

Putting all this together gives us

Lemma 7 *The total number of Type II comparisons made by Fishspear on a run with* n *insertions and max-depth profile* $\mathbf{m}$ *is at most*

$$2t_{\mathrm{B}} + (t_{\mathrm{I}} + t_{\mathrm{O}})\left[\sum_{i=1}^{n} \ln m_i + \left(\ln\frac{2e}{\delta}\right)\cdot n\right].$$

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 $t'_{\mathrm{I}}, t'_{\mathrm{O}}, t_{\mathrm{I}}, t_{\mathrm{O}}, t_{\mathrm{B}}$ which satisfy Equations 1–5. Use Equation 1 to define t'_{I}. 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'_{O}, it is linear with a positive coefficient that is bounded away from zero. It follows that Equation 2 is satisfied for sufficiently large t'_{O}. Similarly, Equations 3 and 4 can be satisfied by taking $t_{\mathrm{I}} = t_{\mathrm{O}}$ sufficiently large, for then $2\delta/\beta < 1$ and the right hand sides are linear in $t_{\mathrm{I}} = t_{\mathrm{O}}$ with positive coefficient. Finally, Equation 5 can be used to define t_{B}. The constant c of Theorem 3 is given by

$$c = (t_{\mathrm{I}} + t_{\mathrm{O}}) \cdot \ln 2, \qquad (19)$$

and one can take

$$c' = 2t_{\mathrm{B}} + 1 + (t_{\mathrm{I}} + t_{\mathrm{O}}) \cdot \ln\frac{2e}{\delta}.$$

We get our best bounds by choosing $\beta = -2\ln\beta = .7034\ldots$. Plugging in to Equation 1 yields $t'_{\mathrm{I}} = 2.843\ldots$. Calculus together with numerical evaluation shows that $t'_{\mathrm{O}} = .5674\ldots$ satisfies Equation 2, and equality holds (to within the limits of our precision) for $q = .141\ldots$. (The function of Equation 2 over the interval $(0,\beta)$ is shown in Figure 7.) Thus, $t'_{\mathrm{I}} + t'_{\mathrm{O}} = 3.410\ldots$. By choosing δ sufficiently close to 0, we can make $t_{\mathrm{I}} + t_{\mathrm{O}}$ arbitrarily close to $3.410\ldots$. Finally, plugging into Equation 19 shows that the constant c of Theorem 3 can be chosen arbitrarily close to

$$\ln(2) \times 3.410\ldots = 2.363\ldots.$$

In particular, $c = 2.4$ works.

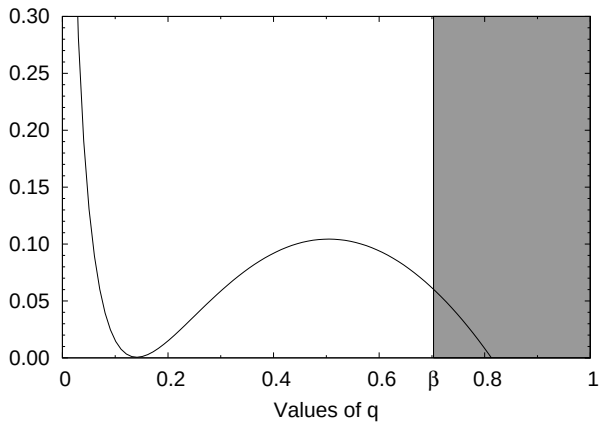

Figure 7: The function $t'_{\mathrm{I}}q-(t'_{\mathrm{I}}q+t'_{\mathrm{O}}(1-q))\ln q-2-q$ for $t'_{\mathrm{I}}=2.844$ and $t'_{\mathrm{O}}=0.5675$.

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

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

[2] Hoare, C. A. R. Communicating sequential processes. *Comm. ACM 21,* 8 (1978), 666-677.

[3] Sleator, D. D., and Tarjan, R. E. Self-adjusting binary trees. In *Proc. 15th ACM SIGACT Sympos. on Theory of Computing* (April 1983), 235-245.

[4] Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update rules. In *Proc. 16th ACM SIGACT Sympos. on Theory of Computing* (April-May 1984), 488-492.

[5] Williams, J. W. J. Algorithm 232: Heapsort. *Comm. ACM 7,* 6 (1964), 347-348.