

# Online Median Finding

Larry Denenberg

(draft)

## Abstract

The **Online Median** problem requires us to add elements to a set and at any time to find the median of the elements added so far. A straightforward solution using two heaps permits adding elements in time  $O(\log N)$  and finding the current median in constant time. We show how to add elements in expected time  $O(1)$ ; in fact, using  $2 + o(1)$  comparisons in the expected case.

“Who the hell cares how much time it takes?  
If it’s too slow for you, buy a faster machine.”  
—Don Knuth, *Full Retraction*, §3.16

## 1 Introduction

The median of a set of totally-ordered elements can be found in linear time; the best algorithm to date requires  $2.942N + o(N)$  element comparisons to find the median of  $N$  elements[BF, DZ]. We are interested in the online problem, where elements arrive one by one and we might be asked at any time for the median of the elements added so far. We require two operations:

- `AddElement( $x$ )`, add a new element  $x$  to the current set, and
- `CurrentMedian()`, return the median of the current set of elements.

A straightforward implementation uses two priority queues, one with operations `Insert` and `DeleteMin`, and one with “inverted” comparisons, i.e., with `DeleteMax` in place of `DeleteMin`. The first queue stores elements greater than the current median and the second stores elements smaller than the current median. We keep the queues balanced: At all times, either they have the same number of elements, or one has one more element than the other. A new element is placed in the appropriate queue, and if that queue is then too large we *rebalance* by transferring its extreme element to the other queue.

When the queues have different sizes, the current median is the extreme element of the larger. Otherwise, we take the extreme element of either queue as the median (i.e., the median of an even number of elements may be either of the two central elements). The algorithm is described precisely in Figure 1; call it the *basic method*.

With priority queues implemented by heaps as in [LD], `AddElement` requires time  $O(\log N)$  in the worst case; fetching the current median is instantaneous. In this note we employ several coding tricks to obtain surprising improvements in the time for `AddElement`, focussing on expected time with the assumption of random inputs.

As usual, we use number of element comparisons as the measure of time, but since we’re interested in actual running time we will not use techniques where comparisons do not reflect clock time. For example, heap `Insert` can always be performed in  $O(\log \log n)$  comparisons by using binary search to find the correct location of the new element on the path from the new leaf

```

class OnlineMedianFinder
{
    public int CurrentMedian; // the current median, always available

    MinHeap BigElements = new MinHeap(); // elements > CurrentMedian
    MaxHeap SmallElements = new MaxHeap(); // elements < CurrentMedian
    int balance = 0; // tells which heap (if any) has an extra element

    public void AddElement(int newval) {

        switch (balance) {

        case 0: // the two heaps have the same number of elements
            if (newval <= CurrentMedian) { // "<=" vs "<" is irrelevant
                CurrentMedian = SmallElements.Insert(newval); // returns new max
                balance = -1;
            } else {
                CurrentMedian = BigElements.Insert(newval); // returns new min
                balance = +1;
            }
            return;

        case +1: // BigElements has an extra element
            if (newval <= CurrentMedian) { // "<=" vs "<" is significant
                // this Insert brings us back into balance
                SmallElements.Insert(newval);
            } else {
                BigElements.Insert(newval);
                // BigElements now has two extra elements; must rebalance
                SmallElements.Insert(BigElements.DeleteMin());
            }
            balance = 0; // we're now balanced in any case
            return; // note that CurrentMedian doesn't change

        case -1: // SmallElements has an extra element
            // [the code is parallel to the +1 case]
        }
    }
}

```

Figure 1: The basic method, implemented in Java.

to the root, then swapping the new element up to that location without element comparisons. But the physical time is not  $O(\log \log n)$  so this trick won't be used.

We ignore space issues, and in particular assume that all heaps are magically as big as required. (A completely different approach to online median finding, where very little space is used but we find only an approximation to the median, has been described and analyzed by several researchers; see [CH] for more information. In the final section we consider how we might proceed when space is limited.)

For simplicity, we generally use terminology appropriate for a `MinHeap`, the heap containing larger elements and implementing `DeleteMin`. For example, we say that “rebalance requires a single `DeleteMin`” even though some rebalances use `DeleteMax` instead.

## 2 Analysis of the Basic Method

At each call on `AddElement`, either the two heaps have equal size or one of them has one more element than the other; these two states alternate. So half the time, on equal-sized heaps, the algorithm performs a single `Insert`.

The other half of the time, either the newly-arrived element belongs in the heap with fewer elements and we regain balance with a single `Insert`, or it belongs in the heap with more elements and we rebalance with two `Inserts` and a `DeleteMin`. These two cases have equal probability when inputs are random. In summary, we perform a three-operation rebalance with probability 1/4 and with probability 3/4 we do a single `Insert`.

Heap operations are not equally costly. To `Insert` a random element into a heap is usually cheap, since a new element will very rarely belong near the top. In fact, the expected number of comparisons for `Insert` is constant, independent of heap size. We can see this with a little handwaving: Roughly half the heap's elements, the larger ones, are in the bottommost layer, so in half of all `Inserts` the new element comes to rest after a single comparison. Similarly, with probability about 1/4 there will be only two comparisons, and so forth. Summing, we can estimate 2 as the expected number of comparisons for `Insert`.

There are several more precise estimates of the expected number of comparisons for `Insert`, depending on specific randomness assumptions[BS, PS]. Our problem seems different, no doubt because of the mixture of `Inserts` and `DeleteMins` and possibly because of the truncated distribution: A new element smaller than the heap's current minimum will almost certainly go

into the other heap! We sidestep this question by simply defining  $\gamma$  as the expected number of comparisons for heap **Insert** of a random element during basic online median finding, recognizing that this value may not actually exist. Experimentation suggests that  $\gamma$  is no more than 2.06.

**DeleteMin** is much more expensive. As we bubble an element down from the root, we need two comparisons at each level to find the smallest child of the current node. About half the heap is at the bottom, so a random element will travel nearly all the way down. But worse, the travelling element in **DeleteMin** is *not* random; it was taken from the bottom level and is therefore even more likely (though not certain) to return all the way down. The expected number of comparisons for **DeleteMin** is thus very close to  $2 \log_2 N$  where  $N$  is the number of elements in the heap, which equals half the number of calls on **AddElement**. (We use this definition of  $N$  throughout.)

Finally, the last **Insert** during rebalance is atypically expensive. The element moved from one heap to the other is necessarily an extreme element of both heaps, and when added to the new heap it bubbles all the way to the root using  $\log_2 N$  comparisons, unless there are duplicate minimal elements.

So rebalance requires roughly  $3 \log_2 N + \gamma$  comparisons, making the expected number of comparisons for **AddElement** about  $(3/4) \log_2 N + \gamma$ . (We've ignored the single comparison in the main algorithm that selects which heap will contain the new element. We'll continue to ignore that comparison—counting only comparisons in heap operations—so there's always really 1 more comparison in **AddElement** than we say.)

### 3 $d$ -ary Heaps

Instead of using binary heaps for priority queues, we can use  $d$ -ary heaps, that is, increasing trees in which each node (with possibly a single exception) has either zero or  $d$  children. Such heaps can be kept in an array exactly the same way as binary heaps: the root is at index 1, the children of the node at index  $i$  are at indices  $di - d + 2$ ,  $di - d + 1$ , . . . ,  $di + 1$ , and the parent of the node at index  $i$  is at index  $\lfloor (i + d - 2)/d \rfloor$ .

(There is a problematic hidden cost here: With  $d$ -ary heaps we must multiply and divide by  $d$  to traverse the heap, and this integer arithmetic can be expensive. In practice,  $d$  must be a power of 2 so that bit shifts can replace arbitrary multiplication and division.)

A  $d$ -ary heap is “shallower” than a binary heap by a factor of  $\log d / \log 2$ . With a shorter path to the root, **Insert** is cheaper, and if we consider **Insert**

$d$	$\gamma_d$	$d$	$\gamma_d$	$d$	$\gamma_d$
2	2.06	6	1.25	32	1.05
3	1.56	7	1.21	64	1.025
4	1.39	8	1.18	128	1.013
5	1.30	16	1.10	256	1.007

Table 1: Estimated upper bounds on  $\gamma_d$ , which is imprecisely defined and may not exist, for selected  $d$ , chiefly powers of 2.

alone there’s no downside to increasing  $d$  indefinitely—in the limit the heap is just a set plus distinguished smallest element, and **Insert** requires exactly one comparison. We now define  $\gamma_d$  as the expected number of comparisons for **Insert** into a  $d$ -ary heap during online median finding. Table 1 has several experimentally-determined approximations to  $\gamma_d$ .

The effect on **DeleteMin** of increased  $d$  is more ambiguous. Although there are fewer levels to bubble down, there are  $d$  comparisons at each level to swap the bubbling-down element with its smallest child. Assuming as before that bubbling continues to the bottom, the total number of comparisons is  $d \log_d N$ , which has a minimum at  $d = e$ . In our case a **DeleteMin** is always paired with a worst-case **Insert**, so a better expression to minimize is  $(d + 1) \log_d N$  with minimum at approximately  $d = 3.6$ . The presence of additional **Inserts**, as in our problem, further increases the optimum value of  $d$ .

Hence we’re clearly better off taking  $d = 4$ , and the expected number of comparisons for **AddElement** becomes  $1.25 \log_4 N + \gamma_4$ , a savings of about 15%. (Raising  $d$  improves not only the expected case but also the worst case behavior.) Upcoming changes will make even larger  $d$  appropriate.

## 4 Heap Operations

Our first step toward further progress is to sharpen the heap operations to our specific needs.

Start with **DeleteMin** and note that we never use this operation without an immediately preceding **Insert** into the same heap. We do well to combine these two: Replace the root with the new element and bubble that element down as far as it goes, returning the original root element as the result. Call this operation **DeleteMinThenInsert**. (Doing the **DeleteMin** *before* the **Insert** is slightly cheaper and doesn’t affect the algorithm since the new element to be inserted can be no smaller than the current heap minimum.)

With `DeleteMinThenInsert` we save an `Insert` completely. Furthermore, `DeleteMinThenInsert` is cheaper than `DeleteMin` because now the element bubbling down from the root *is* random, and therefore more often comes to rest before reaching the bottom. We ignore this savings and continue to count `DeleteMinThenInsert` as  $d \log_d N$  comparisons.

We've already mentioned that the second (and now only) `Insert` during rebalancing is specially costly since the new element is no larger than the current heap minimum. Let's call this operation `InsertMin` (or `InsertMax` for a `MaxHeap`). In the next section we'll provide a special implementation of this operation. Here we just point out a missed opportunity for big savings: `InsertMin` can be implemented with zero comparisons by blindly swapping the new element to the root of the heap—one of our most expensive operations becomes absolutely free. In practice this saves almost no time at all and can even slow things down when there are duplicate elements. In any case we've promised not to cheat in this way.

In summary, we change the rebalancing case of the algorithm from

```
BigElements.Insert(newValue);
SmallElements.Insert(BigElements.DeleteMin());
```

to

```
SmallElements.InsertMax(
    BigElements.DeleteMinThenInsert(newValue));
```

(with corresponding change in the parallel case) yielding a small savings of  $\gamma_d/4$  in the expected cost of `AddElement`.

## 5 Slots

Suppose we give each heap a new location called a *slot*. The slot either is empty or contains a single element. If the slot contains an element, that element is the extreme element of the heap, that is, the smallest element in a `MinHeap` or the largest in a `MaxHeap`; we might say that the slot sits just above the root of the tree. In accord with our convention we'll always say that an occupied slot holds the smallest heap element.

The three heap operations become:

- `Insert(x)`: Perform normal heap `Insert`. Then, if  $x$  lands at the root, and the slot is occupied, and  $x$  is smaller than the element in the slot, exchange the contents of the root with the contents of the slot.

- **InsertMin**( $x$ ): If the slot is empty, simply put  $x$  into the slot. Otherwise, perform **Insert**( $x$ ) as above; the new element ends up in the slot (unless there are duplicate values). In either case the slot ends up occupied.
- **DeleteMinThenInsert**( $x$ ): If the slot is occupied, empty it, and return its contents after doing heap **Insert**( $x$ ) as above. Otherwise, do **DeleteMinThenInsert** the old way. In either case the slot ends up empty.

These changes add no comparisons to the cost of **Insert** unless the new element is the smallest or second-smallest in the entire heap. (The exact effect is analyzed more carefully in the next section.) But **InsertMin** becomes essentially free when the slot is empty, and expensive **DeleteMinThenInsert** becomes cheap **Insert** when the slot is occupied.

How do these changes affect the expected running time of **AddElement**? That is, when do we realize gains from favorable status of the slot?

Both slots are initially empty. **Insert** does not change the status of the slot, so nothing happens until the first rebalance. At that point one heap executes **InsertMin** and its slot becomes occupied. The other heap executes **DeleteMinThenInsert** and its slot remains empty.

If one slot is empty and the other is occupied, then there are two possibilities during rebalance: Either *neither* slot's status is favorable to the heap operations, in which case the rebalance is as expensive as ever and neither slot changes status, or *both* slots are in favorable status, in which case the rebalance costs only a single cheap **Insert**, and the slots switch status, the occupied one becoming empty and the empty one occupied. That is, each rebalance is either expensive and leaves the slots alone, or is cheap and switches their status. By symmetry, these two cases occur with equal probability on random input. (It's never the case that both slots are occupied, nor are they ever both empty after the first rebalance.)

The bottom line is this: In the long run half the rebalances are unaffected by the slots, and the other half change from **DeleteMinThenInsert** plus **InsertMin** to a single **Insert**. The expensive case of the algorithm now occurs only one time in eight, rather than one time in four, and all other cases cost one **Insert**. The expected number of comparisons in **AddElement** becomes  $(5/8) \log_4 d + (7/8)\gamma_4$ .

## 6 Burrows

If one slot is good, more slots are better.

We now attach a number of slots, called the *burrow*, to our heap implementation. The burrow consists of  $k$  slots organized as a stack. If the burrow is nonempty, its top element is the heap's minimum element, and its bottom element is the element next smaller than the one in the root of the tree. (The burrow is so named because it lies “under” the root, even though we confusingly draw trees upside down with the root at the top. So the top of the burrow is farthest from the root.)

The heap operations now work like this:

- **Insert( $x$ )**: Perform **Insert( $n$ )** as in a slotless heap. Then, if  $x$  lands at the root, and the burrow is nonempty, compare  $x$  to the bottom element of the burrow. If  $x$  is smaller, swap these two, then compare  $x$  with the next element up the burrow. Continue until  $x$  reaches either a smaller element or the top of the burrow. The number of elements in the burrow is unchanged.
- **InsertMin( $x$ )**: If the burrow is not full, push  $x$  onto the top of the burrow. Otherwise, perform **Insert( $x$ )** as just above;  $x$  rises to the top of the burrow (unless there are duplicate minimal elements) and the burrow remains full.
- **DeleteMinThenInsert( $x$ )**: If the burrow is not empty, pop it, and return its contents after doing **Insert( $x$ )** as above. If the burrow is empty, do **DeleteMinThenInsert( $x$ )** in the usual (slotless) way.

How does the burrow affect running time? As before, only rebalancing changes the number of occupied slots in either burrow. And whenever we empty a slot in one heap we fill a slot in the other heap, and vice versa except during an initial period. So at any time after the initial period, one burrow will have  $i$  occupied slots and the other  $k - i$  occupied slots for some  $0 \leq i \leq k$  (indeed, the two burrows could be stored in a single array of size  $k$ ). Thus there are exactly  $k + 1$  possible states of the burrows. When we need to rebalance, exactly one of these states is unfavorable and yields an expensive rebalance; in any of the other states, the rebalance requires only a single **Insert**.

With random inputs we find ourselves in each state with equal probability. To see this let  $p_i$  be the probability of being in state  $i$  after a rebalance, and note that for  $0 < i < k$  we have  $p_i = (p_{i-1} + p_{i+1})/2$  since from each

state we transition to the adjacent states with equal probability. Moreover,  $p_0 = (p_0 + p_1)/2$  and  $p_k = (p_{k-1} + p_k)/2$  since the state doesn't change after a rebalance when the burrows are in unfavorable state. These  $k + 1$  equations in  $k + 1$  unknowns have unique solution  $p_i = 1/(k + 1)$  for each  $i$ .

Thus the probability that a given rebalance is expensive is  $p_0/2 + p_k/2 = 1/(k + 1)$ . Since we rebalance with probability  $1/4$ , the probability of an expensive rebalance is  $1/4(k + 1)$ . Note that for  $k = 0$  and  $k = 1$  this expression has the values we expect.

When we add more slots, the proportion of expensive rebalances goes down, all other calls on `AddElement` requiring just an `Insert`. Reducing the fraction of `DeleteMins` then makes it worth while to pick a larger heap arity, that is, to increase  $d$ . Doing so shifts costs from the `Inserts` to the `DeleteMins`, making it advantageous to add yet more slots! This virtuous cycle terminates only when there are so many slots that they add appreciably to the cost of `Insert`. We now take up the details of this analysis.

An expensive rebalance occurs with probability  $1/4(k + 1)$  and consists of a `DeleteMinThenInsert` with empty burrow, using  $d \log_d N$  comparisons as before, plus an `InsertMin` with a full burrow:  $\log_d N + k$  comparisons, since the inserted element always comes to the top of the burrow.

In all other cases we do just an `Insert`. The burrow will on average be half full, with  $k/2$  occupied slots. Therefore with probability  $k/2N$  a new element enters the burrow, and if it does it travels on average halfway up the burrow. So elements that enter the burrow use  $k/4$  comparisons plus  $\log_d N$  comparisons before reaching the burrow. Other elements use  $\gamma_d$  comparisons as before. Thus the expected number of comparisons for `Insert` is  $(1 - k/2N)\gamma_d + k/2N(\log_d N + k/4)$ .

Putting it all together, the expected number of heap comparisons for `AddElement` is at most

$$(1 - 1/4(k + 1))((1 - k/2N)\gamma_d + (k/2N)(\log_d N + k/4)) + (1/4(k + 1))((d + 1) \log_d N + k)$$

which is less than

$$\gamma_d + (k/2N) \log_d N + k^2/8N + ((d + 1)/4k) \log_d N + 1/4$$

A consequence of this expression is that if we let the number of slots grow dynamically, say  $k = \Theta(\log N)$ , then our algorithm runs in expected time  $\Theta(1)$ . It's not hard to let  $k$  grow with  $N$  if we can expand the array of slots; perhaps just before each expensive rebalance we check  $N$  and sometimes allocate a new slot.

We also see that we can have many, many slots without asymptotic expected cost. The expected number of comparisons remains  $\gamma_d + 1/4 + o(1)$  as long as  $k$  is  $o(\sqrt{N})$ . (It is because of this point that we limit the number of slots at all. Otherwise we might simply permit the burrow to grow without bound, making all `InsertMins` free. But if we do so, the number of occupied slots is a (bounded) random walk, and its maximum extent will be  $\Omega(\sqrt{N})$ .)

## 7 Circular Burrows and Burgeoning Heaps

Although the expected time for adding an element is low, the worst-case time can be very bad. Suppose, for example, that we have a billion elements (half a billion in each heap) using  $d = 128$  and  $k = 6000$ . Although we expect an expensive rebalance in only about one `AddElement` in every 24000, that rebalance may take  $129 \lceil \log_{128} 2^{29} \rceil + 6000 = 6645$  comparisons, six thousand times worse than the expected case. Even a simple (non-rebalance) `Insert` might take as many as 6004 comparisons. Another coding trick considerably mitigates this problem and also further improves the expected time.

We implement the burrow as a deque instead of a stack, permitting manipulation at both ends. The burrow is still physically stored in an array, but that array is now “circular”: the ends of the burrow can be anywhere, and the burrow may (or may not) wrap around from the high-index end to element 0. The extra index arithmetic that this scheme introduces can be minimized by using sentinels at the ends of the array, avoiding the need to test for the end of the array as we move from slot to slot.

We can now implement `InsertMin` much more efficiently. If the burrow is not full, just push the new element onto the top as before. Otherwise we must free up a slot. We previously did this with a normal `Insert` of the new element—the new element goes into a new leaf of the tree and then bubbles to the top of the burrow, and during this process the value in the slot at the bottom of the burrow moves to the root of the tree. Instead, we can now take the value from the *root* of the tree, put it into a new leaf, and bubble that value upward; it stops just below the original root. Now the original root location is empty. We move the bottom element of the burrow into that spot, creating a free space in the burrow’s array. Finally, we transfer that free space from the bottom to the top of the burrow by adjusting the burrow boundaries, and we put the new element in that free space. (A complete heap implementation with arbitrary arity and circular burrow can be found at <http://denenberg.com/MinHeap.java>.)

The result of this change is that `InsertMin` requires only the  $\log_d N$

comparisons to bubble an element to the top of the tree;  $k$  comparisons vanish. In the example above, the worst-case number of comparisons for `AddElement` goes from 6645 to 645; still hundreds of times worse than the expected case but an immense improvement.

The circular burrow can also be used to improve `Insert`. If a new element bubbles all the way to the root of the tree, we needn't simply bubble it up the burrow. Instead, we first compare it to the element in the middle of the burrow. If it's larger than this element, we bubble it up as usual. But if the new element is smaller than the middle element, we start from the *top* of the burrow and bubble it *down* into place, adjusting the boundaries of the burrow as before to transfer the (new) free space from the bottom to the top. With this trick we never have to bubble a new element through more than half the burrow; in the example, a worst-case `Insert` is no more than 3004 comparisons, down from 6004.

These improvements also affect the expected number of comparisons. The change to `Insert` has small significance since it happens so rarely; the term  $k^2/8N$  in the expression for the expected number of comparisons is cut in half to  $k^2/16N$ .

The improvement to `InsertMin` is more important. The  $k$  comparisons that create space in the burrow during an unfavorable rebalance are responsible for the term  $1/4$  in the expected-case number of comparisons ( $k$  comparisons with probability about  $1/4k$ ), and these no longer exist. So the expected number of comparisons, assuming that the maximum number of slots is both  $\omega(\log N)$  and  $o(\sqrt{N})$ , becomes  $\gamma_d + o(1)$ . The optimum  $k$  in this range can be estimated by differentiating the expression for the number of comparisons with respect to  $d$ . This process is simplified by ignoring the small term  $(k/2N)\log_d N$ . The best  $k$  for given  $N$  and  $d$  turns out to be approximately  $\sqrt[3]{2N(d+1)\log_d N}$ .

However, for best theoretical performance, not only  $k$  but also  $d$  must increase with  $N$ . The standard implementation of heaps makes this difficult; it is not practical to restructure a heap on fly to increase its arity. Instead, we can use a data structure we might call a **burgeoning heap**, in which the arity of the tree is not constant, but increases as the tree becomes higher.

Such a heap is easy to implement with linked data structures. It can also be stored in an array without loss of space in reasonably practical manner: We keep two auxiliary arrays that store, for each level of the tree, the arity at that level plus a "displacement" that permits us to use every location in the array. If these arrays are called `Arity` and `Displacement`, then the

$N$	mean comps, $d = 2$ & $k = 0$	alternative $d$	alternative $k$	mean comps, this $d$ & $k$
100	6.87	8	10	2.10
1000	9.04	32	100	1.70
100000	14.26	32	2500	1.31
1000000	16.50	64	2000	1.23
100000000	21.72	128	3500	1.072

Table 2: Some experimental results.

index of the leftmost child of the node with index  $i$  is

$$i * \text{Arity}[\mathbf{d}(i)] + \text{Displacement}[\mathbf{d}(i)]$$

where  $\mathbf{d}(i)$  is the depth of node  $i$ . This scheme is reasonable in practice because we always traverse the tree from top to bottom or bottom to top, and hence we can keep track of the current level rather than computing or storing it. The rate of burgeoning can be anything we like, since at the start of each new level we can pick  $d$  arbitrarily then adjust the displacement so that the next array element used is the next one free. (The code for the burgeoning heap modifications can be found at <http://denenberg.com/MinBurgeoningHeap.java>.)

With burgeoning heaps and growing burrow we can let both  $k$  and  $d$  increase with  $N$ , yielding an algorithm that runs with an expected number of heap comparisons  $1 + o(1)$  (still not counting 1 in the main algorithm).

## 8 Results

Table 2 gives some experimental results for various  $N$ ,  $k$ , and  $d$ . We report the average number of comparisons during heap operations calculated over many runs of the algorithm. Number of comparisons is measured “at the margin”: we start counting only after 90% of the calls on `AddElement` have completed. We compare the basic method as described in Figure 1 (using slotless binary heaps) against the final algorithm with circular buffer and fixed  $k$  and  $d$ .

**Acknowledgement:** The author thanks Google Inc. for posing the problem and for providing ample leisure time to work on the solution.

## References

- [BF] Blum, Floyd, Pratt, Rivest, and Tarjan, “Time bounds for selection,” J. Comput. System Sci. **7** (1973) 448-461.
- [BS] Bollabas and Simon, “Repeated random insertion into a priority queue,” J. Alg **6** (1985) 466–477.
- [CH] Cantone and Hofri, “Analysis of An Approximate Median Selection Algorithm,”  
<ftp://ftp.cs.wpi.edu/pub/techreports/pdf/06-17.pdf>
- [DZ] Dor and Zwick, “Selecting the Median,” SIAM J. Comput. 28, **5** (May 1999) 1722–1758.
- [LD] Lewis and Denenberg, *Data Structures and Their Algorithms*, Harper-Collins, 1991, pp 110ff.
- [PS] Porter and Simon, “Random insertion into a priority queue structure,” IEEE Transactions on Software Engineering **1** (1975), 292–298.