Online Median Finding

Larry Denenberg

(draft)

Abstract

The **Online Median** problem requires us to add elements to a set and at any time to produce the median of the elements added so far. A straightforward solution using two heaps permits adding elements in $O(\log N)$ comparisons with the current median always available. We show how to reduce the expected number of comparisons for adding an element to $2 + \epsilon + o(1)$ while preserving the worst-case bound, or to $2 + o(1)$ with slightly higher worst-case bound.

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

A straightforward solution 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; 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. Call this algorithm the *basic method* (Figure 1).

With priority queues implemented by heaps as in [LD], `AddElement` uses $O(\log N)$ comparisons in both worst and expected cases. In this note we employ a couple of coding tricks to obtain surprising improvements in the expected time for `AddElement` under the assumption of random inputs.

As usual, our complexity measure is number of element comparisons. Since number of comparisons serves as a proxy for computation time, we eschew techniques where these two are not linearly related. For example, heap `Insert` can 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 to the root, then swapping the new element up to that location without element comparisons. But the total number of operations is not $O(\log \log N)$, so we will not use this trick.

```
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) {   // "<=" is just as good as "<"
        SmallElements.Insert(newval);
        CurrentMedian = SmallElements.Max();
        balance = -1;
      } else {
        BigElements.Insert(newval);
        CurrentMedian = BigElements.Min();
        balance = +1;
      }
      return;

    case +1:    // BigElements has an extra element
      if (newval <= CurrentMedian) {   // "<=" is better than "<"
        // 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 always balanced at this point
      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.

We (somewhat unrealistically) ignore storage requirements and assume that all heaps grow magically as required. A different approach to online median finding, where very little space is used but which produces only an approximation to the median, has been described by several researchers; see [CH] for more information.

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 calls on `AddElement` require only a single `Insert`.

When the heap sizes are not equal, the newly-arrived element either 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 `Insert`s plus a `DeleteMin`. These two cases have equal probability when inputs are random. In summary, with probability 1/4 we perform the three-operation rebalance, and with probability 3/4 we do a single `Insert`.

To `Insert` a random element into a heap is usually cheap, since the new element rarely belongs 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: Half the heap's elements, the larger ones, are childless elements in the bottom layers, so in half of all `Insert`s the new element comes to rest after a single comparison. Two comparisons are required 1/4 of the time, and so forth. Summing, we find that the expected number of comparisons for `Insert` is 2. This estimate is slightly too low because the lowest layers of the heap needn't contain the largest elements.

There are several sharper estimates of the expected number of comparisons for heap `Insert` with various randomness assumptions[BS, PS]. Our problem seems different, no doubt because of our specific pattern of operations and possibly because of the truncated distribution: A new element smaller than the heap's current minimum almost certainly will go into the other heap! We sidestep this question by simply defining γ as the expected number of comparisons for heap `Insert` of a random element during basic online median finding, recognizing that this value is somewhat ill-defined and may not exist. Experimentation suggests that γ is no more than 2.06.

`DeleteMin`, in contrast, is expensive. As we swap an element downwards we need two comparisons at each level to find the smaller child of the current node. About half the heap is at the bottom, so a random element travels nearly all the way down. But the traveling element is *not* random; it was taken from the bottom level and is therefore even more likely (though not certain) to sink to the bottom. The expected number of comparisons is thus close to the worst-case value, $2\lg(N/2)$. (Here and hereafter N is the number of calls so far to `AddElement` so that each heap contains $N/2$ elements.)

Finally, the last `Insert` during rebalance is atypically expensive. The element transferred from one heap to the other is an extreme element of both heaps, and when inserted in the new heap it travels all the way to the root with $\lg(N/2)$ comparisons (unless there are duplicate minimal elements).

So rebalance uses roughly $3\lg(N/2) + \gamma$ expected comparisons, making the expected cost of `AddElement` about $(3/4)\lg N + \gamma + 1/4$ comparisons, including the one used to select which heap should contain the new element.

We get a small improvement by noting that we never use `DeleteMin` 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`. (The basic algorithm actually does `DeleteMin` after, not before, the `Insert`, but performing the `DeleteMin` first is slightly cheaper and doesn't affect the result since the new element to be inserted can be no smaller than the current heap minimum.)

With `DeleteMinThenInsert` we save an `Insert` on each rebalance. Furthermore, `DeleteMinThenInsert` is even 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 $2\lg(N/2)$ comparisons. Hence the savings in the expected cost of `AddElement` is just $\gamma/4$.

We've already mentioned that the second (and now only) `Insert` during rebalance is especially costly since the new element is no larger than the heap minimum. Call this operation `InsertMin`, or `InsertMax` for a `MaxHeap`. In the next section we provide a special implementation of this operation. (`InsertMin` can in fact be implemented with zero comparisons by blindly swapping the new element to the root of the heap, but we've promised not to cheat in this way.)

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

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

to

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

with corresponding change in the parallel case.

3 Burrows

To make further progress we add to our heaps an auxiliary store, called the **burrow**, where elements can reside. The resulting data structure, the **burrowed heap**, supports the same operations as a standard heap, so we need no changes to the code in Figure 1. The burrow consists of a limited number of locations, each of which either is occupied by an element or is empty.

The elements in the burrow are the extreme elements of the burrowed heap, that is, the smallest elements of a `MinHeap` or the largest of a `MaxHeap`. The burrow serves as a buffer that reduces the expected frequency of the expensive heap operations by providing quick access to these extreme elements. (The burrow is so named because it lies "under" the root of the heap, though it would be drawn above the root.)

To start, we organize the burrow as a stack, storing it in an array `B` of size k. (We invariably use k to denote the total number of available locations in the burrow.) If the burrow is nonempty, the bottom of the stack, location `B[0]`, contains the element just smaller than the root of the heap. The top of the stack contains the smallest element of the entire burrowed heap.

The heap operations are now implemented like this:

- `Insert`(e): Insert e into the heap. If e rises all the way to the root, and the burrow is nonempty, compare e to the bottom element of the burrow. If e is smaller, swap these two and compare e with the next element up the burrow. Continue swapping upward until e reaches either a smaller element or the top of the burrow. The number of elements in the burrow is unchanged.

- `InsertMin`(e): If the burrow is not full, push e onto the top of the burrow; the burrow now has one additional element. Otherwise, perform `Insert`(e) as above; e rises to the top of the burrow (unless there are duplicate minimal elements) and the burrow remains full.

- `DeleteMinThenInsert`(e): If the burrow is empty, perform standard `DeleteMinThenInsert`(e) on the heap and the burrow remains empty.

Otherwise, remove the topmost element of the burrow and return it after doing `Insert`(e) as above; in this case the number of occupied locations in the burrow is reduced by one.

We can do better with a slightly more general burrow structure. Let us also permit removing elements from the bottom of the burrow, making it an input-restricted deque. We can continue to store the burrow in an array of size k; the array becomes a circular buffer storing the burrow. Two cases of the heap operations now become less costly:

- If the burrow is full during `InsertMin`(e), we remove the bottom element of the burrow and insert it into the heap; it rises to the root of the heap. Now there is a free location in the burrow and we can push e onto the top of the burrow.

- During `Insert`(e), if e rises to the top of the heap, we compare e to the element in the *middle* of the burrow. If e is greater than that element, we proceed as before by comparing e to the bottom element of the burrow and swapping upward as far as possible. But if e is less than that element—and hence belongs in the upper half of the burrow—we remove e from the root of the heap, replacing it with the bottom element of the burrow. We can now push e onto the top of the burrow (there is now at least one free location) and then swap it *downward* as far as possible. As before, the number of occupied locations in the burrow is unchanged. With this procedure we compare the new element to at most half the elements in the burrow.

To compute the expected cost of `AddElement` with burrowed heaps, we must first understand how the burrows fill and empty. Since `Insert` does not change the number of occupied locations in the burrow, that number changes only at a rebalance. At that point, the number of occupied locations in one burrow increases by one unless that burrow is already full, and the number of occupied locations in the other burrow decreases by one unless that burrow is empty. Let m denote the total number of occupied locations in both burrows. m is initially 0 and increases only when one burrow is empty and the other is not full, i.e., contains fewer than k elements. At such a point, m is less than k. Thus m cannot increase unless its value is less than k, which implies that m never exceeds k. Similarly, the only time that m can decrease is when one burrow is contains k elements and the other is nonempty. But then $m > k$ and this is impossible.

We conclude that the total number of elements in the two burrows starts at zero, eventually increases to k (assuming random inputs), and remains k

thereafter. That is, at any point after the initial "burrow filling" period, one burrow has j elements and the other has $k-j$ elements for some j, and the only possible change is that a rebalance increases or decreases j. (Indeed, we could store both burrows in a single array of size k.) We are interested in the expected cost of `AddElement` only once the total number of elements in the burrows has become k permanently.

The presence of the burrow does not change the frequency of rebalance: It is still the case that, on average, a quarter of the calls on `AddElement` require a rebalance, and those rebalances are equally likely to move an element from the `MinHeap` to the `MaxHeap` as the other way around. Analysis of the expected running time depends on the following:

Fact. *With random inputs and k elements in the two burrows combined, the burrows are at any time equally likely to be in any of the $k+1$ possible configurations.*

Proof. Think of the states of the burrows as states of a Markov model, and for $j = 0, 1, 2, \ldots, k$ let p_j be the probability that the burrow of the `MinHeap` contains j elements and thus that the burrow of the `MaxHeap` contains $k-j$ elements. Except at $j = 0$ and $j = k$ we have $p_j = (1/2)(p_{j-1} + p_{j+1})$ since elements move from one burrowed heap to the other with equal probability. That is, the value of each p_j is the average of the adjacent values. This harmonic condition implies that p_j, as a function of j, has no local maxima or minima and must be monotonic nonincreasing or nondecreasing. By symmetry, p_j must be constant except perhaps at the endpoints. Then since $p_0 = 1/2(p_0 + p_1)$ we have $p_0 = p_1$ and similarly $p_{k-1} = p_k$. Hence the p_j are all equal and each equals $1/(k+1)$. □

We next bound the expected cost of an `Insert` into a burrowed heap with $N/2$ elements including j elements in the burrow. With probability $1 - 2j/N$ the newly-inserted element belongs in the heap as usual, and the expected cost of the insertion is just γ. Otherwise, we require $\lg(N/2 - j)$ comparisons to find the top of the heap, and then search on average $j/4$ elements in the burrow. So the total expected cost in this case is

$$(1 - 2j/N)\gamma + (2j/N)(\lg(N/2 - j) + j/4) \\ < \gamma + (2j/N)\lg N + j^2/2N.$$

When the $k+1$ possibilities for j are equally likely, we find that the expected cost of `Insert` is less than

$$\gamma + (k/N)((2k+1)/12 + \lg N).$$

Now we can write down the expected cost of the N^{th} invocation of `AddElement` with the assumption that inputs are random and the total number of elements in the burrow has already risen to k. As before, in half the cases the heaps are already balanced and the cost is that of a single `Insert`. In half of the remaining cases the heaps are unbalanced, but the new element is such that only a single `Insert` is necessary.

In the remaining cases we rebalance. With probability $(k-1)/(k+1)$ the burrows are neither empty nor full, so the `DeleteMinThenInsert` becomes an `Insert` and the `InsertMin` is free. Even in the cases where one heap is empty and the other full, the probability is 1/2 that the rebalance will be in the favorable direction, so again only a single `Insert` is required.

So the expensive case of `AddElement` happens with probability $1/4(k+1)$. Here the expected cost is that of a `DeleteMinThenInsert` on a standard heap, $2\lg(N/2) < 2\lg N$, plus the expected cost of `InsertMin` on a burrowed heap with full burrow, $\lg(N/2-k) < \lg N$.

Putting this all together, the expected cost of `AddElement` is less than

$$(4k+3)/(4k+4)(\gamma + (k/N)((2k+1)/12 + \lg N)) + (3\lg N)/(4k+4) + 1.$$

where the final 1 is, as before, the initial comparison to the current median. This is less than

$$\gamma + 1 + (k/N + 3/4k)\lg N + k^2/4N.$$

For any fixed k, this is of course still $\Theta(\lg N)$. If, however, we let k grow as $\lg N$, periodically expanding the burrow's array, then the expected number of comparisons can be made less than $\gamma + 1 + \epsilon + o(1)$ for any preassigned ϵ. The ϵ can be eliminated as well by permitting k to grow at any rate in $\omega(\log N)$, but if we do so then the rate of growth of k becomes the worst-case bound in place of $O(\log N)$ since we might have to traverse half the burrow on each `Insert`.

4 d-ary and Burgeoning 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 $\lfloor (i+d-2)/d \rfloor$. (The multiplication and

d	γ_d	d	γ_d	d	γ_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 γ_d, which is imprecisely defined and may not exist, for selected d, chiefly powers of 2.

division by d to traverse the heap can be expensive; in practice, d should be a power of 2 so that bit shifts can be used instead.)

A d-ary heap is shallower than a binary heap by a factor of $\log d/\log 2$. With shorter path to the root, `Insert` is cheaper, and if we consider `Insert` alone there's no downside to increasing d indefinitely—in the limit we get a set with distinguished smallest element, and `Insert` requires exactly one comparison. We define γ_d as the expected number of comparisons for `Insert` into a d-ary heap during basic online median finding (i.e., with no burrows). Table 1 has several experimentally-determined approximations to γ_d.

The effect on `DeleteMin` of increased d is less monotonic. Although there are fewer levels to bubble down, there are d comparisons at each level to swap the travelling 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 practice, the best d for a given application depends on the mix of `Insert`s and `DeleteMin`s; as the proportion of `Insert`s increases, the optimal value of d also increases.

With d-ary heaps the expected comparison count for `AddElement` is less than

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

which, assuming $\gamma_d \to 1$ as $d \to \infty$, can be made less than $2 + \epsilon + o(1)$ for any preassigned ϵ by choosing sufficiently large d and letting k grow as $k_0 \log N$ for appropriate k_0. The worst-case number of comparisons remains $O(\log n)$.

We can bring the expected number of comparisons down to $2+o(1)$ at the cost of relaxing the $O(\log N)$ worst-case bound by permitting d to 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 can be stored in an array without loss of space in fully

N	mean comps, $d = 2$ & $k = 0$	new d	new k	mean comps, new d & k
100	7.87	8	10	3.10
1000	10.04	32	100	2.70
100000	15.26	32	2500	2.31
1000000	17.50	64	2000	2.23
100000000	22.72	128	3500	2.072

Table 2: Experimental results. "Mean comps" counts comparisons during only the final 10% of calls on `AddElement`.

efficient 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 waste no array locations. If these arrays are called `Arity` and `Displacement`, then the index of the leftmost child of the node with index `i` is

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

where `d(i)` is the depth of node `i`. The rate of burgeoning is controlled by picking d at the start of each new heap level, then adjusting the displacement so the next array element used is the next one free. This scheme is reasonable in practice as long as we access the tree only by traversing it from top to bottom or bottom to top, so we can keep track of the current level as we go rather than computing or storing it.

With burgeoning heaps we can let d increase with N, then let k grow at any rate in $\omega(\log N)$, say $k = (\log N)^{1.1}$. The rate of growth of k becomes the worst-case bound and the expected number of comparisons is $2 + o(N)$.

5 Results

Table 2 gives some experimental results. It reports average number of comparisons for `AddElement` over many runs of the algorithm. Number of comparisons is measured "at the margin", that is, during only the final 10% of calls on `AddElement`. We compare the basic algorithm using binary heaps and no burrow against the same algorithm with burrowed heaps having fixed arity and burrows of fixed size.

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.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*, HarperCollins, 1991, pp 110ff.

[PS] Porter and Simon, "Random insertion into a priority queue structure," IEEE Transactions on Software Engineering **1** (1975), 292–298.

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