Brief Analysis of Build Heap and Heap Sort Algorithms

Assignment 406
CS 310

Cameron Moberg

April 5, 2017

The heap is a binary-tree-based data structure, where, depending on the type, the parent of a node is always bigger or always smaller than its children. The former is called a *max heap*; the latter a *min heap*, and in this analysis we will exclusively be referring to a *max heap*. We will also be analyzing methods that heaps are integral in: `build_heap` and `heap_sort`.

For both methods, the value for n represents the number of nodes in the heap.

1 Build Heap

To analyze the `build_heap` algorithm, we chose the `percolate_down` algorithm on line 165 as the basic operation because it is called most frequently, is the most expensive, and is the philosophical heart of this algorithm. However, the runtime of `percolate_down` is unknown and thus we must first analyze it to compute the time complexity of `build_heap`.

1.1 Percolate Down

To analyze the `percolate_down` algorithm we chose both the comparison between child nodes on line 192 and the comparison between root value and the largest valued child node on line 200 as the basic operations. This is because they occur almost equally frequently and are the philosophical heart of this algorithm. Even though checking whether or not the node has children occurs most frequently, we didn't choose it as the basic operation because it's not the comparison we truly care about.

In this algorithm, we must assume that the root node's children are recursively heaps and then compare the root with both of its children. If any of the (at most 2) children are bigger in value than the root node, we swap values with the highest valued node, and then recursively use `percolate_down` on the node that was just swapped. We repeat this until the node is greater than or equal to its children or it is a leaf.

An empirical analysis of the `percolate_down` algorithm with multiple values of n produces the results shown in Figure 1. The standard function $f(n) = \log n$ with a constant multiplier has been added to illustrate the analysis.

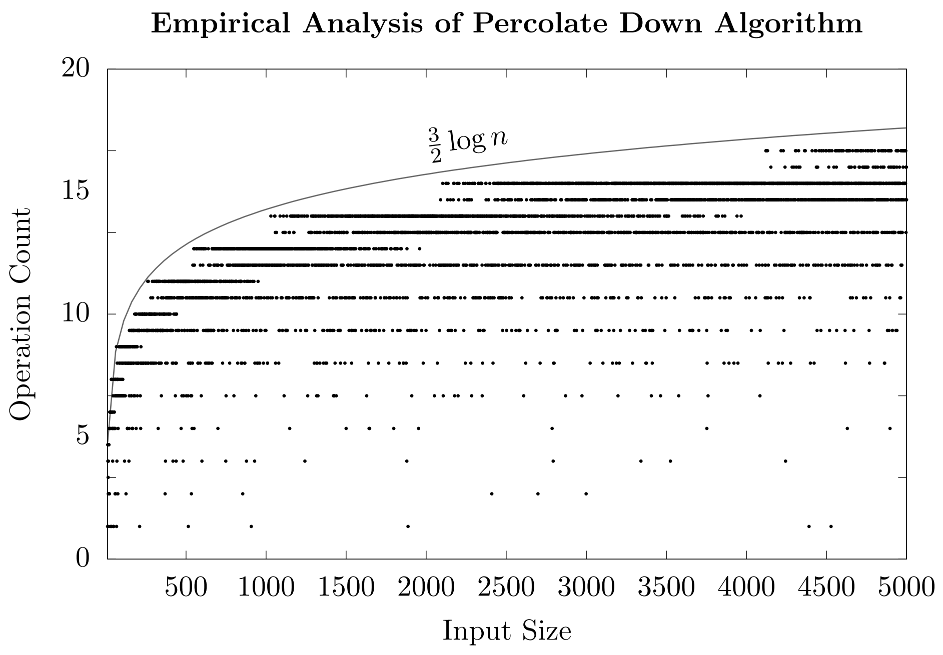

Figure 1: *Results of percolate down algorithm.*

The C++ method used to produce Figure 1 is shown below.

```
void empiric_perc( Comparable key )
{
    heap.at( 0 ) = key;       //heap is a max_heap by definition.
    heap.percolate_down( 0 ); //0 is the root node index.
}
```

In short, we set *key* to a generated (pseudo) random number, set the root node of the heap equal to it, and then `percolate_down` is used on the root node to to simulate a "worst-case scenario".

An examination of the `percolate_down` implementation explains the empirical results when we observe that, in the worst-case, the root node's value is the smallest of all nodes in the tree. This means the node has to sift down through every level of the tree and become a leaf, at the bottom of the tree. This requires a comparison at every level, or, $\log n$ comparisons [SS93].

Since this algorithm can terminate prematurely, we cannot tighten the bound to Big-Θ, and thus the `percolate_down` algorithm can be described as

$$T(n) \in O(\log n)$$
$$T(n) \in \Omega(1)$$

1.2 Build Heap cont.

An empirical analysis of running `build_heap()` with multiple values of n produces the results shown in Figure 2. The standard function $f(n) = n$ with constant multipliers has been added to further illustrate the analysis.

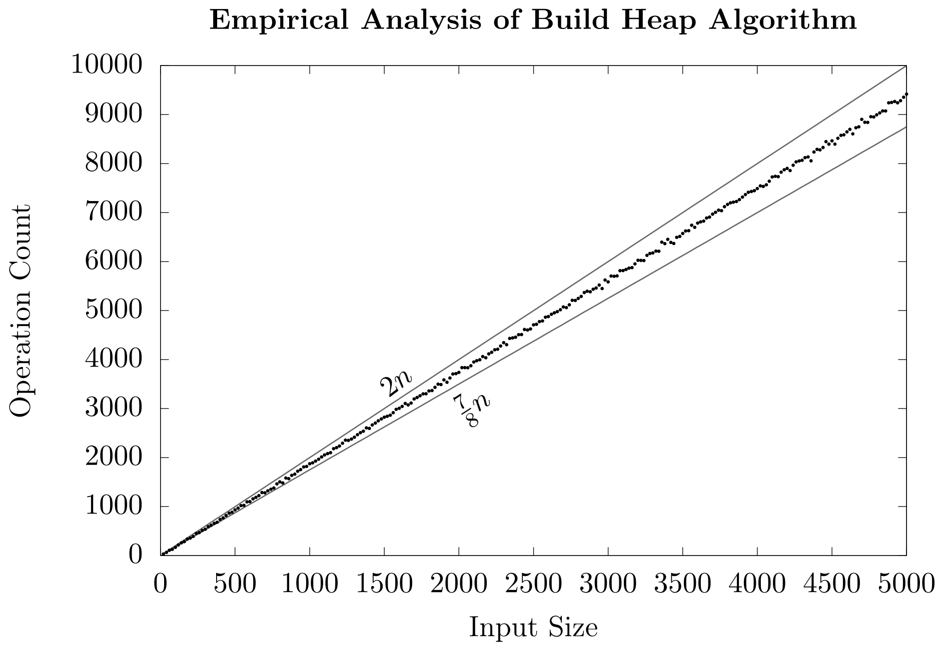

Figure 2: *Results of build heap algorithm.*

Depth of Nodes	Height of Nodes	Num. of Comparisons
h	0	0
$h-1$	1	$2(\frac{n}{4})$
$h-2$	2	$2(\frac{n}{8}+\frac{n}{4})$
$h-3$	3	$2(\frac{n}{16}+\frac{n}{8}+\frac{n}{4})$
...	...	...

Figure 3: *Breakdown of total comparisons by height of heap, in worst-case.*

At first glance, it would appear that the algorithm is described by $O(n \log n)$ since there is a single deterministic loop that iterates $\frac{n}{2} - 1$ times containing a $O(\log n)$ method. This is technically correct, but not the most optimal analysis. Upon closer inspection we can tighten the bound to $\Theta(n)$ because `percolate_down` is only $\Omega(1)$ for a single run, but, over multiple runs of a heap it is shown to be $\Theta(\log n)$ [Knu73].

In the `build_heap` algorithm, an arbitrary heap is created, followed by `percolate_down` being called in reverse level order starting on the first parent node. The total number of comparisons in a worst-case `build_heap` scenario is shown in Figure 3. We observe that nodes of height 0 (there are $\frac{n}{2}$) have 0 comparisons because they have no children, and the total number of comparisons in `build_heap` can be represented as the summation shown below.

$$\sum_{i=1}^{h} \frac{2(n+1)}{2^{i+1}} \in O(n)$$

This summation can be shown to be linear [Bec17], and after observing that the algorithm must access every element of the heap at least once, we can see the `build_heap` is $\Omega(n)$ time. Therefore, we conclude that the algorithm is of the form

$$T(n) \in \Theta(n)$$

The author never explicitly states what efficiency class the algorithm is in, but it can be assumed that they concluded that it is $O(n)$ [Lev12, p. 230]. Our analysis does agree with the author's, up to a point, but by observing that the algorithm must be at least $\Omega(n)$ we can tighten our bound to Big-Θ, instead of just Big-O.

2 Heapsort

In general, a heapsort algorithm contains two major parts: building the heap from data and repeatedly removing the highest valued node to put into the to-be-sorted array. To analyze the `heap_sort` algorithm we chose the `delete_max()` method on line 123 as the basic operation because this occurs most frequently and is the philosophical heart of this algorithm. We ruled out building the heap since it only occurs once, and was shown previously to be $\Theta(n)$.

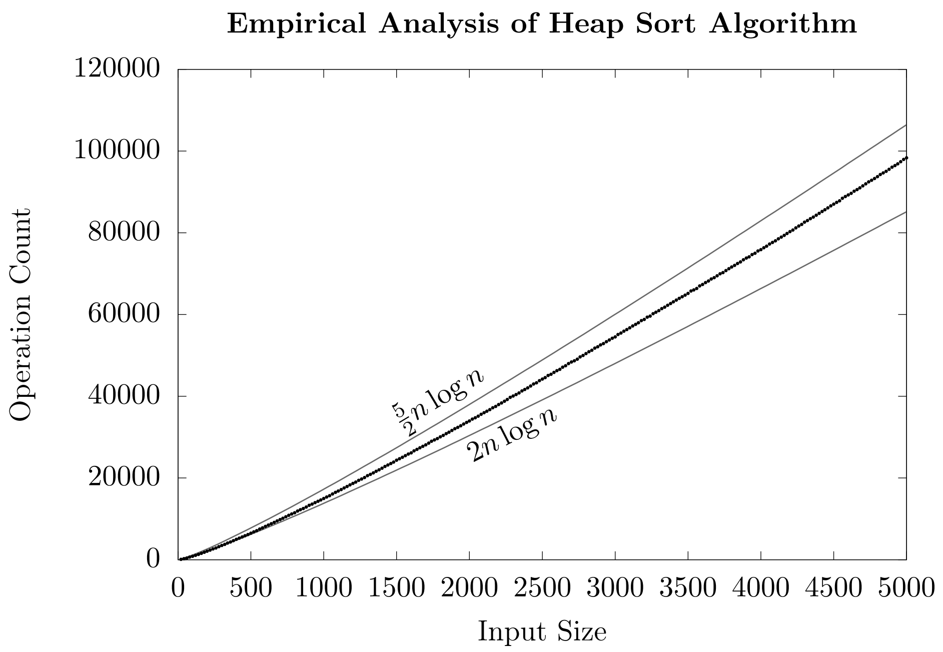

Figure 4: Results of implemented heapsort algorithm.

An empirical analysis of running `heap_sort` for multiple values of n produces the results shown above, in Figure 4. The standard function $f(n) = n \log n$ with constant multipliers, has been added to illustrate the analysis.

An examination of the code itself explains the empirical results when we observe that this algorithm uses the method `delete_max()`, which deletes the largest valued node and *re-heaps* the heap using `percolate_down`. Since this method call is in a deterministic for loop and `delete_max`'s basic operation was shown to be $\Theta(\log n)$ time, we can extrapolate that `heap_sort` is of the form

$$T(n) \in \Theta(n \log n)$$

However, this analysis is completely dependent upon one thing: the input data. The best case scenario depends wholly upon the whether or not the data has duplicate values in it. If there are duplicate values in the data the best-case runtime turns into $\Omega(n)$, however, if duplicates do not appear in the data then the best-case runtime turns into $\Omega(n \log n)$ [SS93, pp. 84–86].

Shown below are possible conclusions based on input data.

Duplicates

$$T(n) \in O(n \log n)$$

$$T(n) \in \Omega(n)$$

Unique

$$T(n) \in \Theta(n \log n)$$

Our author states " [heapsort] falls in the same class as that of mergesort", which is not an outright conclusion, but points toward the assumption of $\Theta(n \log n)$. Our analysis agrees with our author's conclusion only if the data does not include duplicates. If the data does include duplicates the only thing affected is the algorithm's best-case, and thus changes our Big-Θ to Big-O.

References

[Knu73] Donald Knuth. *The Art of Computer Programming*. 1st ed. Vol. 3. Addison-Wesley, 1973. ISBN: 9780201896855.

[SS93] R. Schaffer and Robert Sedgewick. “The Analysis of Heapsort”. In: 15.1 (1993), pp. 76–100.

[Lev12] Anany Levitin. *Introduction to The Design and Analysis of Algorithms*. 3rd ed. Pearson, 2012. ISBN: 9780132316811.

[Bec17] Jon Beck. “PQs and Heaps”. Presentation. Mar. 2017.