Assignment 406
CS 310

Cameron Moberg

April 2, 2017

The heap is a tree-based data structure, where the root is empty or contains a data element of either higher or lower value than both of its children. The former is called a *max heap* and the latter a *min heap*. In this analysis we will exclusively be referring to a *max heap*, and will be analyzing a method that the heap uses: `build_heap()`, and a method that is implemented using a heap: `heap_sort()`.

For both methods, the value for n is taken to be the number of nodes in the heap.

1 Build Heap

To analyze the `build_heap()` algorithm, we chose the `percolate_down()` method on line 160 as the basic operation because this happens most frequently, is the most expensive, and is the philosophical heart of this algorithm.

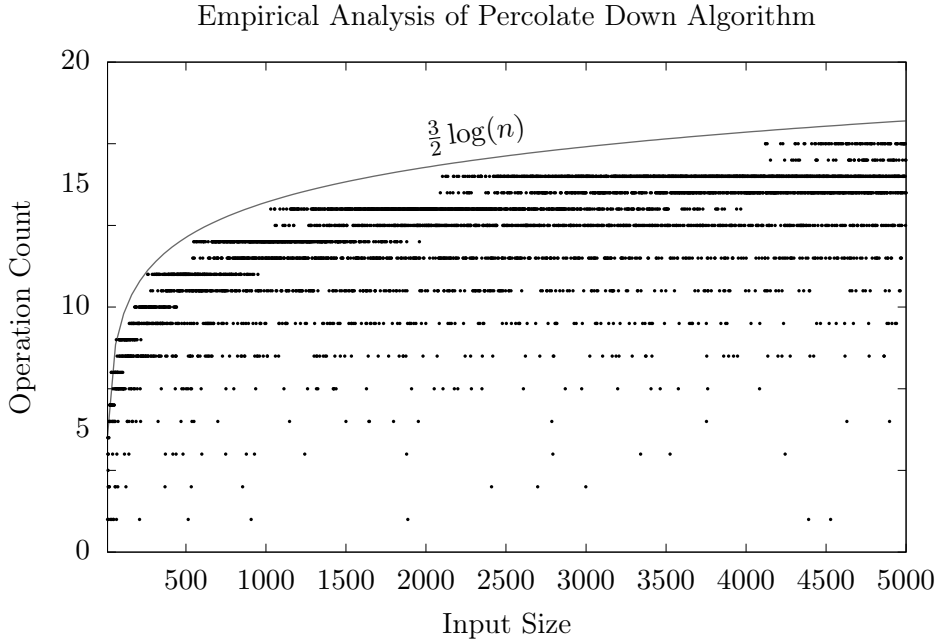
However, as it turns out `percolate_down()` is the only method called by `build_heap()`, and thus we must analyze it in order to find the time complexity of the `build_heap()` method.

1.1 Percolate Down

To analyze the `percolate_down()` algorithm, we chose both the comparison between child nodes on line 188, and comparison between root and the largest child node on line 195 as the basic operations. This is because they occur most frequently and are the philosophical heart of this algorithm. Regardless of the operation chosen, all of the local work inside of `percolate_down()` is linear, will only impact the constants and not the end analysis.

In this algorithm, we assume that the root node's children are heaps, and compare the root node with both of its children. If any of the children are bigger than the root node, we swap values with the largest child, and then recursively call `percolate_down()` on the node that we swapped the root with. We repeat this until the root node has no children or is greater than or equal in value to its children.

An empirical analysis of running `percolate_down()` for multiple values of n produces the results shown below. Standard function $f(n) = \log n$ with constant multipliers, has been added to illustrate the analysis.



The C++ method used to produce this:

```
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, then we set the root node of the heap equal to it, and then `percolate_down()` was called with the root node index in order to simulate a "worst case scenario".

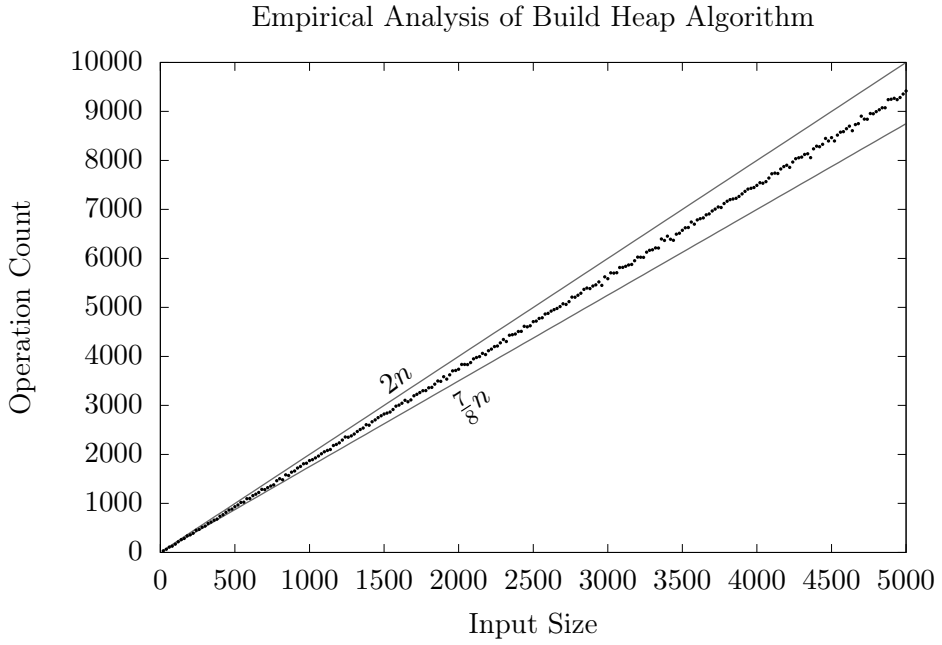
An examination of the `percolate_down()` method itself explains the empirical results when we observe that if the root node's value is the smallest of all nodes in the tree, it will have to end up as a leaf, at the bottom of the tree. This requires a comparison at every level of the tree, or, $\log n$ comparisons.

Since this algorithm can end early, 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 below. Standard function $f(n) = n$ with constant multipliers, has been added to



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 best we can do. Upon closer inspection we can tighten the bound to $O(n)$.

This is because of the fact that most of the work is comparisons are done on the lower levels, where there are less total comparisons than the higher levels. In a full binary tree there are $n = 2^{h+1} - 1$ nodes, where h is the height of the tree. The number of comparisons in `build_heap()` is shown below.

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	$2(\frac{n}{16} + \frac{n}{8} + \frac{n}{4})$
...	...	...

When `build_heap()` runs on a full binary tree, it starts on the level one above all of the leaves, as the leaves will have 0 comparisons, and iterates through calling `percolate_down()` on every node in reverse level order. The worst case amount of operations done is equivalent to the summation shown below.

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

The summation, although not done in this analysis, can be shown to be linear. Also, realize that the algorithm must access every element of the array at least once, which gives it $\Omega(n)$ time. Therefore,

we conclude that the algorithm is of the form

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

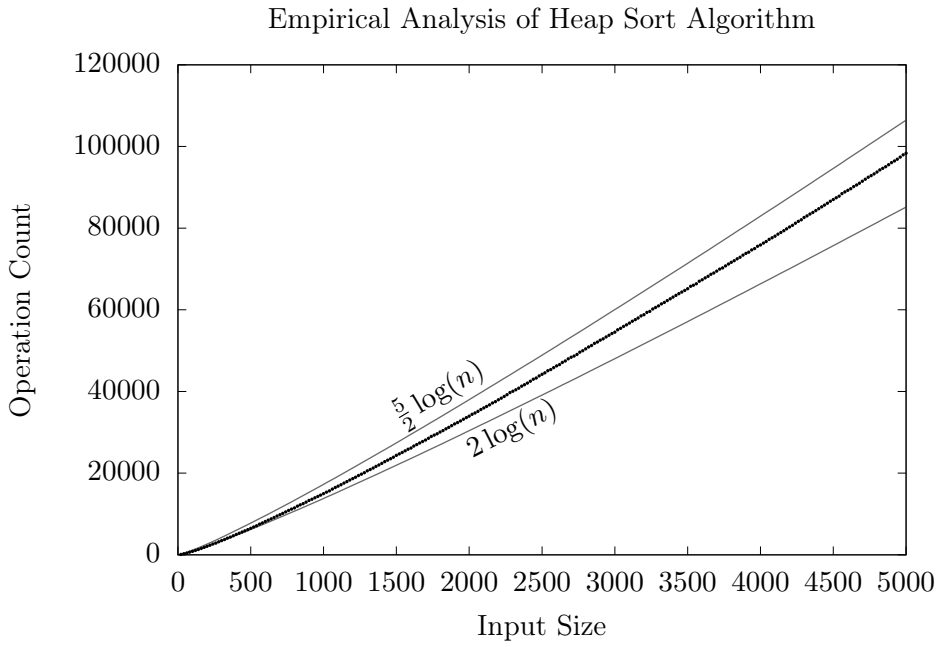
The author never explicitly stated what efficiency class the algorithm is in, but it can be assumed that they concluded that it is $O(n)$ [**levit**]. Our analysis does agree with the author's up to a point, but by observing that the algorithm must be $\Omega(n)$ we can tighten the bound to Big-Θ, instead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

2 Heap Sort

In general, a heap sort algorithm contains two major parts: building a heap from data, and repeatedly removing the root node and inserting it into the to-be-sorted array. As shown above, the first part can be done in linear time.

To analyze the `heap_sort()` algorithm, we chose `delete_max()` method on line 118 as the basic operation because this occurs most frequently and is the philosophical heart of this algorithm. We also chose this method because it has already been shown in this analysis that the first part of the sort, building the heap, is $\Theta(n)$ time, and only happens once.

An empirical analysis of running `heap_sort()` for multiple values of n produces the results shown below. 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 is calls the method `delete_max()`, which in turn has constant work, except for the call to `percolate_down()`, which has been shown above to be $O(\log n)$ time. If we go back and take a look at `heap_sort()`, there is a deterministic for loop that runs approximately $n/2$ times. When

we combine these two facts, we get that `heap_sort()` is of the form

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

We cannot tighten the bound to Big-Θ because of the fact that `percolate_down()` is not Big-Θ.