A Quantitative Analysis of Heap Building

Trevor Hodde

trhodde@wpi.edu

CS Department, WPI

**Abstract.**

In this paper, I describe the implementation of two different methods for constructing a heap. These two methods are common, though they typically serve different purposes. The two methods are usually referred to as “max-heapify” and “max-heap-insert” (Cormen, et al.), however this paper will refer to them as “sift-down” and “sift-up” respectively. This paper explains my analysis and observations of these two methods and describes how the methods perform when they are used to build and sort heaps given sets of randomly generated integers.

1. **Introduction and Background**

Heapsort is typically a very fast algorithm with an average and worst case complexity of *O (n lg n)*. The classic implementation of heapsort uses a sift-down approach simply because sift-up is more expensive for heap building. This is because the number of comparisons and swaps that occur during a call to sift-up increases with the depth of the node in the heap, and there are many more “deep” nodes than “shallow” nodes in a heap. However, with the sift-down approach, the number of swaps per call decreases with the depth of the node. Therefore, roughly half the calls to sift-down will have at most one swap, and about one quarter of the calls have at most two swaps (Perlis).

Of course sift-up is still capable of building a heap, and it is a popular choice for heap repair. This paper describes the analysis and demonstration of sift-up as a heap building technique, but also proves why it may not be an ideal candidate for such a task.

The implementation of heapsort and the various techniques of heap building described by this paper were written in C on a Macbook Pro and compiled with gcc (though it should compile on any platform, as it does not require any platform specific libraries). A makefile has been provided to aid in the compilation process. The algorithm for building and sorting a heap is run a total of 8 times, 4 using the sift-up method and 4 using sift-down. For each iteration, the number of elements in the initial array increases by an order of magnitude, starting at 10^k where k is equal to 2 and ending at 10^k where k is equal to 5. The elements of the array are then randomized 10^5 times to ensure truly random permutations.

The metrics gathered from this evaluation include the number of comparisons made through the duration of the algorithm, as well as the time spent on building and sorting the heap. This data is used to compare the sift-up and sift-down implementations of heap construction and heap sorting. Runtime performance was measured using built-in C functions, namely the *time* header file. This is to ensure accurate measurements that are platform agnostic.

1. **Why Not Sloppy Sort?**

There is another technique for heap repair, which this paper does not analyze because it cannot be used to build a heap. This method is called *SloppyHeapSort* and it uses a “sloppy” sift-down approach for heap repair. This procedure is interesting because it reduces the overall number of comparisons by moving particular elements to the bottom of the heap and then calling sift-up on that element until it finds its proper location. The reason this cannot be used for heap construction is that it does not consider the children of the node in question, and it leaves a “hole” in the position from where the element was taken. It simply takes the element in question from its position within the heap, and moves it to the bottom. The algorithm then moves this element up until it finds its home.

1. **Analysis**
2. **Comparisons**

All the element comparisons done by this algorithm reside in the sift-up and sift-down functions. For sift-down, the worst case is when a node has two children; it needs to make an extra comparison to find the larger of the two children. This means each call of this function will make at most 2 comparisons. Additionally, since sift-down is recursive, the number of comparisons it makes on *n* (where *n = 10^k, 2 <= k <= 5*) items is *H (n) = H (m) + 2*, where *m* is the number of items in the recursive call. This leaves roughly *2n/3* nodes in the subheap of size *m* (which is passed in the recursive calls to sift-down). Therefore, if our recurrence is *H (n) = H (2n/3) + 2* and *H (1) = 0*, we can determine that the number of comparisons should be about *O (n lg n).* Figure 1 below shows that the number of comparisons for sift-down is very close to *O (n lg n).*



Figure 1

However, the number of comparisons required when using the sift-up function is far larger. When building the heap in this manner, the number of comparisons is of the same magnitude as the sift-down approach, but sorting the heap can make several orders of magnitude more comparisons. This is because when using the sift-up technique, the number of swaps and comparisons increase with the depth of the node being sifted. Since most nodes are located at the bottom of the heap, there could be a great deal of comparisons required to move up the heap, and a node could potentially be sifted all the way to the root, which would take as long as *Ω(n lg n).*



Figure 2

1. **Time**

As Figure 1 above shows, the heap sort procedure runs quickly, and increases very slightly as the number of comparisons increases with the magnitude of the number of nodes in the heap. As the number of nodes increases from 100 to 100,000, the time spent building and sorting the heap only increases by a few hundredths of a second. This is because of how quickly a heap can be built and sorted. The build heap procedure runs in *O (n)* time and the actual sort runs in *O (n lg n)* time. This means that in the worst case, sift-down heap sort will run in *O (n lg n)* time. The biggest advantage of using heap sort in practice is its favorable, worst case runtime, so overall, these numbers show how a sift-down heap sort can handle large sets of data reasonably quickly.

When analyzing the results of running sift-up heap sort shown in Figure 2, there is a similar trend; however it is immediately clear that this implementation is not as advantageous. While it is certainly possible to implement heap sort in this manner, the time spent building and sorting the heap is far greater in all cases than the times recorded from the sift-down implementation. This is particularly true in the case where k=5 (or n = 100,000). The number of seconds recorded for each order of magnitude of *n* increases proportionately, until n = 100,000. At this point, the runtime of this algorithm jumps from less than 1 second to about 75 seconds. With multiple runs of this algorithm, the behavior is consistent, so this outlier cannot simply be written off as erroneous data collection. The number of comparisons from k = 4 to k = 5 more than doubles, but the time should be expected to increase proportionately as it had for the rest of the data. Since these results appear to be consistent with each run of this algorithm, we can assume that this is the main factor that persuades people to choose the sift-down implementation of heap sort over sift-up.

1. **Conclusion and Further Research**

The implementations of the algorithm used to build and sort a heap for this analysis were very similar to those described in *Introduction to Algorithms* and the notes given in class. This leads me to believe with confidence that my implementation of both sift-up and sift-down are correct. With that said, the results speak for themselves, with the exception of the outlier in the sift-up *build\_heap* function when *n* starts to get large. The rest of the data makes sense and generally follows a predictable pattern across multiple runs of the program. The time and comparisons tend to increase proportionately with each order of magnitude of *n* and the sift-down implementation performs much better than the sift-up implementation, which was expected.

The only mild overhead that could be considered to reduce the runtime of these algorithms is to remove the comparison counter. I built a special function called *cmp(int, int)* which simply compares two values and increments the *comparison\_counter* variable. This is a very short function, however as *n* starts to get large (n >= 100,000), the number of comparisons starts to increase greatly. As the sift-up approach shows, the comparisons can get into the upper millions, or even billions. This means that the *cmp* function is being called billions of times, and the overhead of calling a function so frequently might add up to something quite substantial. Perhaps this can explain the sift-up data, since its comparison count is much greater than that of sift-down, however further analysis would have to be done to prove this. If research were to be continued on this topic, I would certainly be interested to see if this is in fact a bottleneck, and also how removing this function and replacing it with a more trivial approach would affect the results. If this did prove to be the cause of the anomaly, I would like to prove it by analyzing values of *k* greater than 5. If the algorithm continued to perform proportionately for sift-up as it did for sift-down, then we have found and corrected the bottleneck.

One other bottleneck worth noting (but one that does not directly affect the performance of the algorithm) is the time spent randomizing the array prior to building the heap. This time is recorded separately as to avoid skewing the data, however, running the program to collect data can take 20 to 30 minutes. Most of that time is spent randomizing the array used to build the heap, but it was worth pointing out in case further analysis is desired.

1. **References**

Cormen, Thomas H. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009. Print.

Perlis, Don. 10/23/2003. http://www.cs.umd.edu/class/fall2006/cmsc351/notes/heapsort/.

Muhamma, Rashid. Heap Sort. http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/heapSo rt.htm.