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.

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. This means that

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