Heap Analysis

Liam Adams

February 24 2019

insertValue Analysis

My heap is implemented in C++ with std::vector, which is like a resizeable array. The first step in insertValue is to insert the element at the end of the heap, which runs in linear time in the worst case because the end of the heap is usually not at the end of the array. To insert an element at the end of the heap all the elements that have been removed from the heap must be moved to the right. It then runs the minHeapify function on each element in the bottom half of the array. Each call to minHeapify takes $O(log_b n)$ time, so a loose upper bound is on insertValue is $O(n + nlog_b n) = O(nlog_b n)$.
However the time for minHeapify to run varies depending on the height of the element it was called upon in the tree. An n element heap has height $log_b n$ and at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of any height h. The time required by minHeapify is $O(h)$ and the cost at any level is $\lceil \frac{n}{2^{h+1}} \rceil O(h)$ so the runtime can be bounded by

$$O(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h})$$

. The sum is a decreasing geometric series so the running time of insertValue is $O(n)$. The constant factor associated with this function for a branching factor greater than 2 will probably be greater than a binary tree because there are more comparisons per level because there are more children.

removeMin Analysis

removeMin swaps the minimum element with the last element in the heap and decrements the heapSize counter, both operations take constant time. Then it run minHeapify on the root of the heap, which will take $O(log_b n)$ time. So the runtime of removeMin is $O(log_b n)$. As before, The constant factor associated with this function for a branching factor greater than 2 will probably be greater than a binary tree because there are more comparisons per level because there are more children.