Heap Analysis

Liam Adams

February 24 2019

insertValue Analysis

My heap is implemented in C++ with std::vector, which is a resizeable array. The first step in insertValue is to insert the element at the end of the heap, which runs in constant time in the average case, it can run in linear time in the worst case if the array needs to be resized. It then runs the minHeapify function on each element in the bottom half of the array. Each of the $\frac{n}{2}$ calls to minHeapify takes $O(log_b n)$ time, and it makes at most b comparisons in every call. So a loose upper bound is on insertValue is $O(c + \frac{bn}{2} log_b n) = O(nblog_b n)$. 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 $\lceil \frac{bn}{2^{h+1}} \rceil O(h)$ so the runtime can be bounded by

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

. The sum is a decreasing geometric series so the running time of insertValue is bounded by the infinite decreasing geometric series, which is $\frac{\frac{1}{2}}{(1-\frac{1}{2})^2}$. Therefore $f(n) = O(nb)$. The constant factor associated with this function for a branching factor greater than 2 will probably be less than a binary tree because the tree will be shorter, so there will be more overhead with all of the recursive calls.

removeMin Analysis

removeMin gets the first element of the heap, then swaps the first and last element, then removes the last element. All of those operations take constant time. Then it runs minHeapify on the root of the heap, which will take $O(blog_b n)$ time. So the runtime of removeMin is $O(blog_b n)$. As before, the constant factor associated with this function for a branching factor greater than 2 will probably be less than a binary tree because the tree will be shorter, so there will be more overhead with all of the recursive calls.

input-4 Test

With a branching factor of 2, input-4.txt ran for 80 minutes needing 898588120 comparisons
With a branching factor of 64, input-4.txt took over 2 hours to run before I halted execution. The branching factor of 2 was much more efficient than branching factor of 64. I think this is because the b term in insertValue and removeMin is much larger than the performance gain you get from having a shorter tree.