Corbin Charpentier

Lab 6 Report

**Introduction:**

The following report contains efficiency analysis of the Binary Search Tree and Binary Heap abstract data types. The report starts with implementation, a description of how the experiment was quantified. After that, the actual experimental results and brief analysis are displayed. Finally, the report concludes with a deeper analysis of the results and a few comments .

**Implementation:**

*Timing* – timing was implemented using the provided Timer class. A global instance of Timer was used record the time in seconds needed to execute the insertion and deletion operations for the Binary Search Tree and Binary Heap. More technically, the timer began (Timer::start()) ticking when the forloop for each operation began executing and stopped (Timer::stop()) when each for‐loop finished. The elapsed time, returned by Timer::stop(), was the total time required for each insertion and deletion operation, respectively.

*Comparison Counting* – this was implemented by adding a member variable (int comparisons)

to the Binary Search Tree and Binary Heap classes which keeps track of the number of comparisons .Whenever the member functions compare or iterate a loop, count was incremented by one.

*Random Data* – random values were attained by calling int\* generateRandomDataInRange(int n, int range), which took a size and a range, and returned a dynamically allocated array of size n, containing data from 0 to range.

**Experiment:**

The heap’s build method is obviously more efficient than the BST’s. This is because the heap first organizes the data into a full Binary Tree, then swaps nodes up the tree in O(n) time until the heap is structurally sound. The Build BST method must insert all nodes, potentially comparing each it with the number of nodes equal to the height of the tree. Thus it runs in O(nlg(n)) time.

Since the range, not the size, the input data changes, it’s no surprise that the two structures executed in time consistent with the input size. While the heap and the BST insert in O(lg(n)) time, the BST must follow pointers to the correct insertion point, an operation slower than the Heap’s random access array. This could partially explain the time difference, although the BST made additional comparisons proportional to the increase in time.

For the Heap and BST, linear complexity was expected for the search operation. Because the heap could linearly walk down an array, as opposed to recursively following pointers, it was able to finish faster than the BST.

For Remove, the Heap again beat out the BST for speed. This, however, doesn’t make much sense. Because the BST can find any value in lg(n) time, it should remove faster than the Heap, which must iterate over the array to find a particular value and restructure the tree (nlg(n)).

There must have been a problem with the BST::findMin() method, as it runs in constant O(1). In reality, findMIn should executed in O(lg(n)). The heap, as it should be, was much faster than the BST, as the min value is always at array[0].

Because of the problem with BST::deleteMin, the experiment should the BST executing faster and with less comparisons than the Heap. Both operations should executed in O(lg(n)) time.

The problem with findMIn appears to also be present in findMax. While BST::findMax() should be faster than Heap::findMax(), it should not be this much faster. The Heap’s versions runs in O(n) since it must check the entire array. The BST’s version runs faster, O(lg(n)), since it is organized in such a way that the max is always the farthest node on the right.

Again, the results were skewed by a bad BST::findMax function. The Heap’s data looks plausible, as the increase in data range increase the probability a searched value will not exist. We should also see this in BST::deleteMax(). The Heap executes deleteMax() in O(nlg(n)) and the BST in O(lg(n)).

**Conclusion:**

Aside from the BST::findMin()/findMax() debacle, the data structures operated as expected. Overall, it appears that the Binary Min Heap is a better choice for building and finding and deleting the min. However, in terms of Big-O, the BST is equal or better in all other operations. To find the max, the BST simply must traverse all the way to the right, whereas the heap must linearly search through the entire array (though staring from the back is probably a good idea). Further, the BST is structured in such a way that any find operation will executed in O(lgn), while the heap, unless it’s findMin(), requires n operations.