Landon Crowther

U0926601

4/14/2017

CS 2420

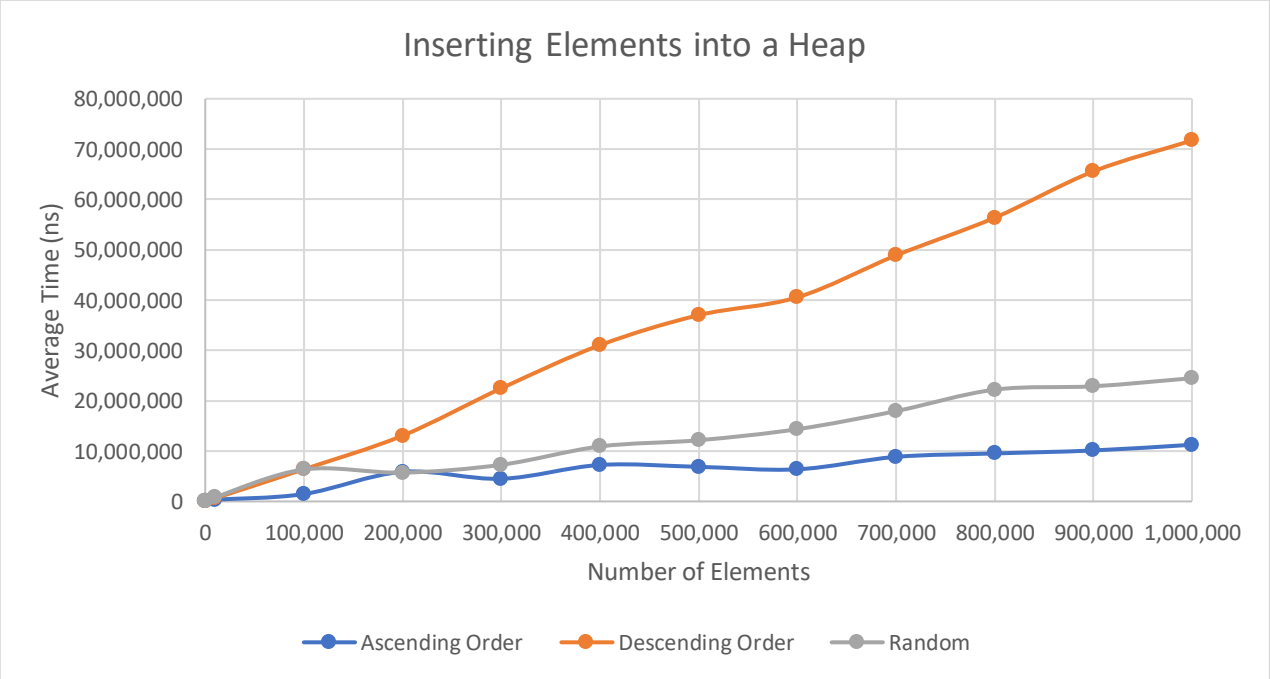
Partner: Michelle Simmons

The most important part of a heap is that it can store data in an array, but the data is used in a tree-like fashion. The minimum (or maximum, depending on if a comparator is provided) data is always stored at the top, and the maximum data is always stored at the bottom. The great thing about a heap is that it has unique properties, where ½ of the data is in the bottom row, ¾ of the data is in the next row up, 7/8 above that, and so on.

A heap's best implementation is a priority queue, where if a given set of data is placed in a priority queue, retrieving the top N data points is very easy and requires no sorting. This is due to the nature of how the heap is built, where the most "important" elements are always at the top. In the real world, priority can be seen in literal trees, as each branch has sub branches, but the "biggest" branch is almost always the trunk of the tree. In an abstract (computer) world, a priority queue can be useful in a situation where the top 10 scoring players (out of 1000 for example) are all that matters. In this case, the scores of each player would be placed in a priority queue, and the top 10 would be the first 10.

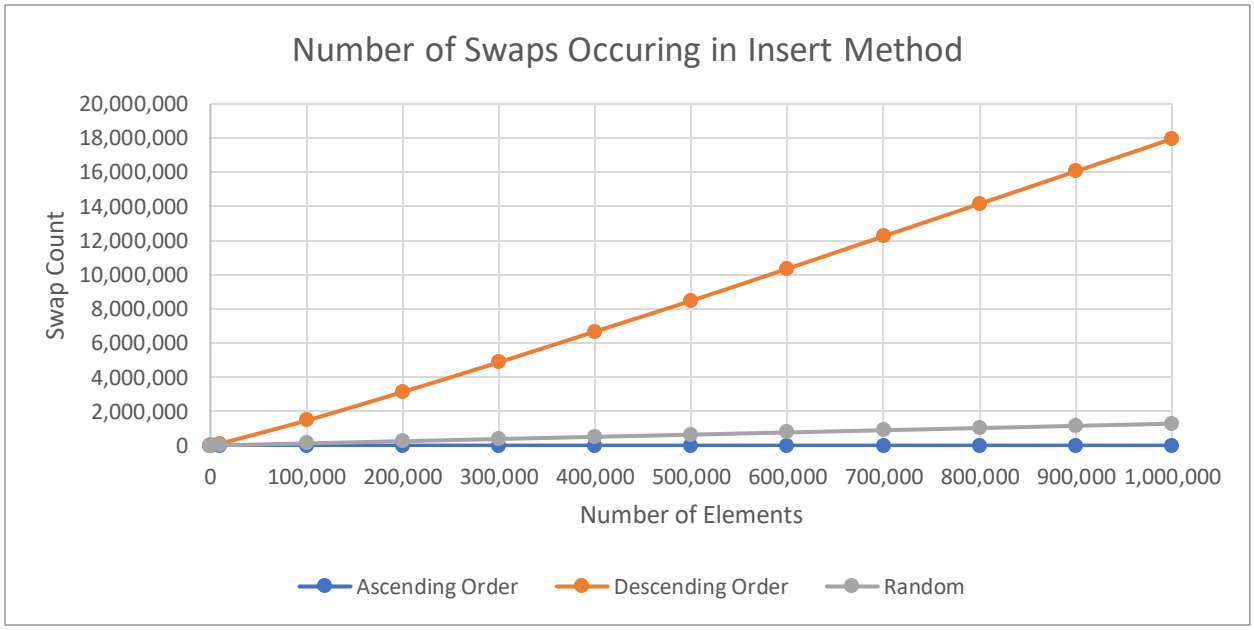
Timing and Analysis:

We did a variety of testing on our heap to determine its effectiveness. The first test we did was adding N elements that were in increasing order, decreasing order, and random order. Inserting into a heap should show relatively linear behavior. This is because every time an element is added, it is added to the end of the array storing the heap's data. Although inserting may be linear, once the element is inserted it must "bubble up" the heap until it is at the proper location. Adding elements to a minimum heap in ascending order would be the most efficient, and adding elements to a minimum heap in descending order would be the least efficient. This is because in ascending, none of the elements must change location, and in descending, *every* element will have to move to the top. Random data will be closer to "average case" which we would expect to fall between worst and best-case data. The following behavior was observed when adding random, ascending, and descending data. Note, for each size of elements, the addition was done 25 times and an average time was computed.



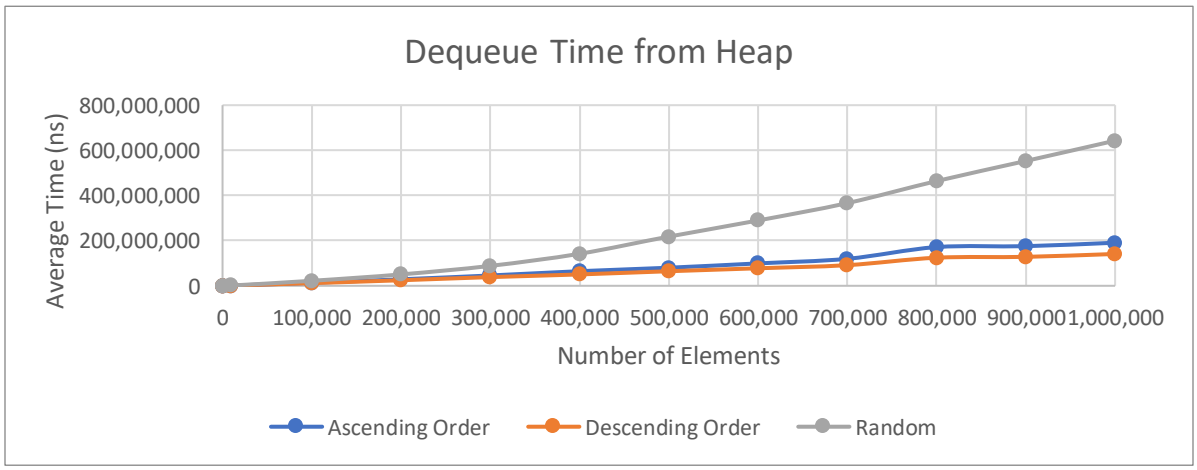
It is evident that overall, inserting follows a linear pattern. Like predicted, the most efficient insert was data that was in ascending order, while the most inefficient was data that was in descending order.

The next test that was performed was determining how many "swaps" were performed. In our test, we only measured the swaps for inserting; therefore, only the "bubble up" method was called. In the "bubble up" method (called "swim" in our program), the value of the number that was just inserted is compared to its "parent." If the value is less than the parent, the swap method is called, and the recently inserted element switches places with the parent. This method is continuously called until the value of the parent is either less than or equivalent to the inserted value. Like insert, we predict that data in descending order would have the most swaps. This is because every element will have to be swapped at least once. Likewise, ascending order would have 0 swaps, because no elements need to be shifted in a minimum heap. Random data (or "average-case" would have a relatively small number of swaps, significantly smaller than descending. The following figure illustrates our results:



As predicted, the number of swaps is 0 for ascending data, relatively small for random data, and extremely high for descending data.

The next test that was performed was the dequeue method. In this method, the element at the top of the tree is removed. The following steps occur during this remove process: the top element is swapped with the very last element, the very last element (which was the top element) is assigned to null, the size is decremented, and the "bubble down" method is called on the new top element. In the "bubble down" (called sink in our implementation) method, the current element is compared to it's successors. In our minimum heap implementation, if the current element is greater than any of it's successors, it is swapped with the *lowest* successor. This ensures that the smallest element is always on top. Once swapped, the comparisons continue until the current element is less than its successors, or there are no more successors. The expected behavior of the dequeue method is O(logN). This is because on average, each element would have to shift down at most logN times until it is in the right place. In this method, we would expect the ascending and descending data to behave the same. That is because after the Heap is constructed (with ascending or descending data), they will essentially be the exact same heap. The following figure illustrates the dequeue results:



The above figure fits our predictions. Ascending and descending performed almost identically (linear), where random (average-case) shows logN behavior. This is what we would expect.

Lastly, although we have no data to confirm, we implemented the final two methods from this assignment. The first was build heap from array. This method takes in an array, and builds the heap from this array. Because the heap has tree like behavior, and half of the elements are at the "end" of the array, to get the heap into the proper heap orientation, we simply called the sink method on the top half of the data, and working upwards.

The last method we implemented was the heap sort method. Heap sort works by simply dequeuing each element of the Heap, and putting it at the end of the array. After calling heap sort on a minimum array, the user is left with an array in ascending order. Likewise, the opposite behavior would be expected for a maximum heap. Heap sort behaves O(NlogN), where N is the number of elements in the Heap. This is because N elements must be dequeued, and each dequeue has behavior of logN.

Software Engineering

This assignment probably took 7-9 hours. In my opinion, the hardest part was getting our sink method working properly. Initially I was trying to come up with some clever way to implement this, but I ended up having to create a large path of while loops and if statements. The next hardest part was ensuring and predicting the big O behavior.

The compare method is useful because it allows our heap to be used generically, as long as a compare method is provided. The user could make a custom object and sort them in a heap.

Like previously mentioned, the code is generic. Although the array is really an array of Type, to allow the compiler to process the code, we must "trick" the compiler by defining an array of Objects but casting to Type. The suppress warning is used to prevent Java from giving us a compiler warning that the casting took care of.

The Dot notation is used in the toString method, because it allows us to easily print out the string of our Heap in a GraphWiz format, anywhere in the code. GraphWiz allows the user to easily interpret and see what is going on in the Heap.

The test provided is useful, because it tests the Heap implementation very thoroughly, and randomly as well. The random aspect of this test ensures that it works for more than the user-defined preset testing data, covering a wide range of possibilities.