Liam Ormiston
2767240
4-12-18

Lab 09 Write up: Experimental Profiling

Organization of experimental profiling.

I broke down my experimental profiling into separate functions in order to help organize my tests. I found that it was easiest to create a function for each test and then one function that will orchestrate the tests. Therefore, for each data structure I had 4 functions. This helped my code feel more organized and modular.

Input data generating using random number generator.

For generating random numbers, I utilized the <cstdlib.h> to call rand() % (5*m)+1 in order to create a number that is between 1 and 5m. To ensure that I have a different seed for each time I call rand(), I used srand(time(NULL)). This creates a new random seed to each time that I call rand().

CPU time recording in C++.

In order to keep track of how much time a process takes, I needed to include <time.h> library. From here I can create a new clock_t to create a timer to keep track of the CPU clock cycles. Then when I need to start my newly created timer I run clock(), which grabs the CPU's clock time. Then when I want to stop the timer, I just grab the clock time again using clock(), but now I subtract the value I had earlier with the value I just got to see the overall time elapsed. So I had clock_t t;, t = clock(), t = t - clock();

Data recording and analysis.

For each test I needed to run it 5 times and take the summation of every run and divide by 5. This would give me the average of the test. I then did this for the 5 different given sizes. For the delete functions I deleted .001m entries of the array of size m for each deleteMin and deleteMax. These were the results of the program:

```
BST TESTS:
Build of 1000000 took 1.386188 seconds
Delete Max of 1000 took 0.000411 seconds
Delete Min of 1000 took 0.000422 seconds

Build of 2000000 took 3.197912 seconds
Delete Max of 2000 took 0.001058 seconds
Delete Min of 2000 took 0.000988 seconds
```

Liam Ormiston
2767240
4-12-18

```
Build of 3000000 took 5.246447 seconds
Delete Max of 3000 took 0.001459 seconds
Delete Min of 3000 took 0.001380 seconds

Build of 4000000 took 7.405666 seconds
Delete Max of 4000 took 0.002151 seconds
Delete Min of 4000 took 0.002596 seconds

Build of 5000000 took 9.457004 seconds
Delete Max of 5000 took 0.002623 seconds
Delete Min of 5000 took 0.002441 seconds
```

MAX HEAP TESTS:

```
Build of 1000000 took 0.060279 seconds
Delete Max of 1000 took 0.001583 seconds
Delete Min of 1000 took 2.708721 seconds

Build of 2000000 took 0.085231 seconds
Delete Max of 2000 took 0.003471 seconds
Delete Min of 2000 took 10.878606 seconds

Build of 3000000 took 0.113389 seconds
Delete Max of 3000 took 0.004959 seconds
Delete Min of 3000 took 24.400166 seconds

Build of 4000000 took 0.140835 seconds
Delete Max of 4000 took 0.006573 seconds
Delete Min of 4000 took 43.702106 seconds

Build of 5000000 took 0.169888 seconds
Delete Max of 5000 took 0.009497 seconds
Delete Min of 5000 took 68.433189 seconds
```

MIN HEAP TESTS:

```
Build of 1000000 took 0.055244 seconds
Delete Max of 1000 took 2.694381 seconds
Delete Min of 1000 took 0.000552 seconds
```

Liam Ormiston
2767240
4-12-18

```
Build of 2000000 took 0.083974 seconds
Delete Max of 2000 took 10.911826 seconds
Delete Min of 2000 took 0.001206 seconds

Build of 3000000 took 0.117823 seconds
Delete Max of 3000 took 24.597651 seconds
Delete Min of 3000 took 0.001981 seconds

Build of 4000000 took 0.139174 seconds
Delete Max of 4000 took 43.760902 seconds
Delete Min of 4000 took 0.002954 seconds

Build of 5000000 took 0.186208 seconds
Delete Max of 5000 took 75.193405 seconds
Delete Min of 5000 took 0.004130 seconds
```

Performance comparison, observations and summary.

Initial observations revealed that building the min and max heaps was very efficient using the bottom-up approach. This makes sense since inserting would be O(1) since it is first added to the back of the array. Then the build would be O(n) since we go through n/2 of the array to swap values. This would result in O(n) time complexity versus binary search tree which is O(n^2). This is because every insertion you have to compare to every value down the branch the node travels.

Another observation was that deleteMin and deleteMax for binary search tree were relatively equal compared to min and max heaps. This is because deleteMin for a max heap or deleteMax for a min heap is incredibly inefficient. It requires looking through all the children of the heap which would be n/2. This results in a time complexity of O(n) versus its deletion complement which would be O(logn) because the min/max is at the root and we just need to repair the tree from there.

Note: All the time complexities are worst-case.

Conclusion

Each data structure has its benefits and downsides. If one were to need a structure that was constantly being added to and only need to delete either the min or max value, then a min/max heap would be the most ideal. However, if one had a static data and needed to quickly delete both min and max value, then a binary search tree would be more ideal.