1. Worst case asymptotic running time analysis:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | isEmpty | size | Insert | findMin | deleteMin |
| BinaryHeap | O(1) | O(1) | O(log(n)) | O(1) | O(log(n)) |
| ThreeHeap | O(1) | O(1) | O(log(n)) | O(1) | O(log(n)) |
| MyPQ | O(1) | O(1) | O(log(n)) | O(1) | O(log(n)) |

1. Run-time values

Timing for Binary Heap (in nanoseconds)

|  |  |  |
| --- | --- | --- |
|  | Insert | delete |
| 10 | 923 | 2256 |
| 100 | 935 | 3056 |
| 500 | 648 | 3245 |
| 1000 | 824 | 2408 |
| 10,000 | 654 | 3062 |

Timing for Three Heap (in nanoseconds)

|  |  |  |
| --- | --- | --- |
|  | Insert | delete |
| 10 | 956 | 4023 |
| 100 | 623 | 3408 |
| 500 | 453 | 4520 |
| 1000 | 635 | 4301 |
| 10,000 | 635 | 3652 |

Timing for MyPQ (in nanoseconds)

|  |  |  |
| --- | --- | --- |
|  | Insert | delete |
| 10 | 853 | 4268 |
| 100 | 346 | 4654 |
| 500 | 836 | 3200 |
| 1000 | 990 | 4620 |
| 10,000 | 982 | 4561 |

1. a. The asymptotic analysis wasn’t very useful in predicting the run time for my implementations.

b. I think the way java compiles and runs programs cause serious variations in timing when the run time is on the order of a few thousand nanoseconds. Also, increase in size of n isn’t very significant due to the speed at which the computer is capable of navigating the array.

c. In theory, however, the binary tree should be the fastest. Operations of multiplication by 2 (simple bit shifting operation) makes array navigation the fastest in the binary tree. I think in all cases, the binary tree would out-perform the other two implementations.

1. To test my implementations, I built a tester program that would test sizes of 1 – 10. Inputting random values into my Priority Queue, and printing them back out, making sure they are in sorted order.
2. a.

|  |  |
| --- | --- |
|  | Child nodes at: |
| Binary | i\*2 and i\*2+1 |
| 3-Heap | i\*3-1 and i\*3 and i\*3+1 |
| 4-Heap | i\*4-2 and i\*4-1 and i\*4 and i\*4+1 |
| 5-Heap | i\*5-3 and i\*5-2 and i\*5-1 and i\*5 and i\*5+1 |

b.

The left-most child for any heap of order-d would be:

i\*d –d + 2 = child index, for parent index: “i”, and number of children “d”