**Performance Report**

**Sequential Numbers**

|  |  |  |
| --- | --- | --- |
|  | Data Size | Avg. Time in Milliseconds (PolyTree / JavaTree) |
| Entry 1 | 5000 | 114.6/0.6 |
| Entry 2 | 10000 | 474.2/11.8 |
| Entry 3 | 15000 | 1088.2/6.2 |
| Entry 4 | 20000 | 1971.2/31.4 |
| Entry 5 | 25000 | (StackOverFlow)/10.4 |

**Random Numbers**

|  |  |  |
| --- | --- | --- |
|  | Data Size | Avg. Time in Milliseconds (PolyTree / JavaTree) |
| Entry 1 | 5000 | 7.8/10.2 |
| Entry 2 | 10000 | 18/3 |
| Entry 3 | 15000 | 21/11.2 |
| Entry 4 | 20000 | 8/13.6 |
| Entry 5 | 25000 | 26/18.6 |

**Explanation**

First note that the each entry was collected by taking the average of 5 different runs of the class TreeSpeed.java with the data size indicated.

The very obvious trend is that the all the average run time statistically increased as the data size increased, with some notable outliers.

The performance of the Polymorphic Tree for sequential numbers was substantially worse. With it overflowing the stack at data size 25000. This is possibly due to the fact that all the data was sequentially added to the right side of the tree, starting with 1. Creating a tree with height equaling that of the data size. But on the other hand, Java tree should follow a similar approach as well. More research of the structure of Java tree needs to be done to accurately compare the two trees.