Performance of Stacks Using Various Growth Functions

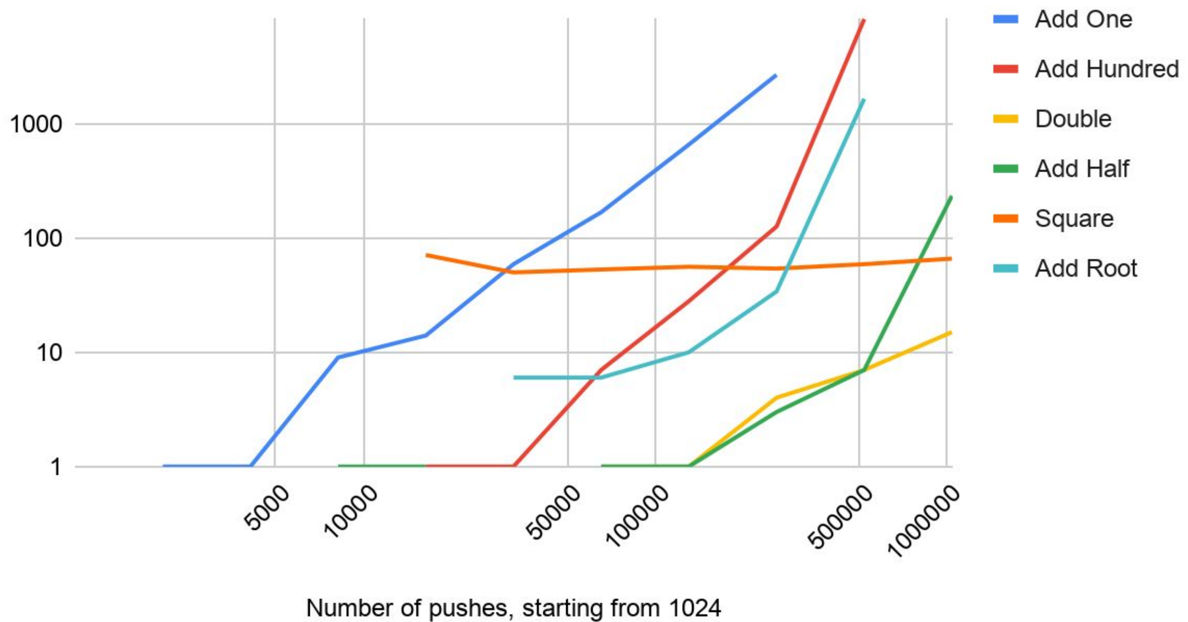
Aaron Paterson

Programs very often store a variable amount of data, which occupies a variable amount of memory. One such abstract data type is a stack, which stores and restores homogenous values in first-in first-out order. Allocating memory takes a long time, so allocating more memory than is immediately necessary can have big performance benefits if it avoids reallocating memory down the line. The trade off of implementing a stack like this is that it takes up more space, which leads to a question about its performance: how much memory ought to be allocated to a stack such that it takes up minimal time and space?

Assuming we want stack growth to be as lazy as possible, it should only happen when a value is pushed to a full stack. The only heuristic needed to grow a stack like this is its capacity. To test the performance of a stack's growth, the subject will be a function of its capacity, the independent variable will be the size to which it must grow, and the dependent variable will be the time it takes to do so. Subjects will include:

- Increasing the stack capacity by one
- Increasing the stack capacity by one hundred
- Doubling the stack capacity
- Increasing the stack capacity by half its current capacity
- Squaring the stack capacity
- Increasing the stack capacity by the square root of its current capacity

The machine of choice will be a laptop running windows LTSC with a 2.20 Ghz processor and 32GB of RAM. To improve scope, tests will be taken as stacks grow to increasing powers of two. To improve accuracy, tests will be performed ten times between the hours of 12 and 1 AM while the laptop is connected to a power source in a very hot and humid dormitory, and the results will be averaged. To improve running time, each function will stop being tested after growth lasts more than ten seconds.



As the first twelve growths lasted less than one millisecond, they are omitted from the graph. As expected, the common method of doubling stack growth was the fastest, although it allocates more memory than all but the square method. Interestingly, the square method seemed to make pushing slower, suggesting that array assignments are not performed in

constant time. Also interesting is that the add one performance is only a constant factor slower than double, suggesting that they are the same big O time complexity.

As memory is almost universally more affordable and efficient than processing power, doubling seems like the best trade off between time and space. If memory capacity was very limited and memory allocation was very fast, incrementing would be ideal. Either way, it seems that scaling the capacity by a constant factor allows one to account for this trade off in a more predictable and effective way than the more complicated methods.