Graph-based Analysis

# ABS Graphs

# ABQ Graphs

Text-based Analysis

The Array Based Stack (ABS) and Array Based Queue (ABQ) are two fascinatingly powerful data structures to use in C++. Dynamic, easy to use and relatively quick are three characteristics of the structure that give it a desirable base to use for any project involving millions to billions of elements. However, one may ask: really, how quick are they?

To answer that question, some counterquestions must be answered beforehand. What would the scale size of the array be? How many elements will be added? Which data structure will be used? One could use the graphs provided above to find a quick answer to the question above. Yet, the requirements of this assignment demand I provide a thorough explanation using text in compliment to the visualizations.

As one may assume, the more elements at play, the longer the process will take. However, this is not a pure linear relationship. The scale factor of the array will also play a role in the duration of the process. Surprisingly, a larger scale factor does not always translate into less time. This can be seen from the change in scale factor from 2 to 3 in the ABQ graphs for 1 billion elements, taking ~1.2 million clocks and ~1.5 million clocks, respectively. My hypothesis is that popping and dequeuing *even* scale factors take less time relative to their odd increments. Why? Good question.

The amount of resizes an array will perform seems to be dependent on both the number of elements and the scale factor, yet becomes less dependent on the number of elements when the scale factor is larger. For example, with a scale factor of 1.5, pushing 10 million elements into an ABS results in 39 resizes, and for 1 billion elements it takes 4 resizes. However, with a scale factor of 100, pushing either 10 million or 1 billion elements takes only 4 resizes. This is entirely based on the logic that with such a grand scale size, there is a plentiful amount of memory left in the array after every resize that it takes a larger proportion of the finite elements to fill the array before requiring another resize. Thus, taking such few resizes for the large span of numbers past a certain threshold.

This leads us to the question with no concrete answer. What is the optimal scale size?

It depends.

As stated earlier, the relationship between the number of elements and the size is not linear in most cases. Which tends to vary with the scale size. Some operations with small scale sizes perform quicker than large scale sizes with the same elements at play. Yet those same two scale sizes invert after some number of elements and the larger scale size performs quicker than the smaller one. Due to this, you never will be sure if you are using the *most optimal*scale size. Good luck sleeping tonight.

To conclude this analysis, we take a look at the performance differences between the ABS and ABQ data structures. They are closely related, but not identical. In most cases, the ABQ performs faster than the ABS. This is likely due to one, conclusive reason. As I was performing this high element pushing and enqueueing into both the ABS and the ABQ. I noticed the ABQ was taking far longer than the ABS. So I decided to create an optimized dequeuing system where the beginning of the array was only reset to zero when it was resized, otherwise, the first element in the ABQ – the one to be dequeued first – always had its index stored in as a private data member of the class. I believe it was this optimization that led to the enhanced performance of the ABQ relative to the ABS.