Jonathan Trost
COP3503
SUMMER 2021
LAB 3 EXTRA CREDIT

Table Of Contents:

Introduction:

The below is an analysis of an implementation of stack and Queue data structures implementing them with arrays. The testing performed involved filling the data structure up with N elements and then completely emptying the data structure. For each fill and empty of the data structure, the number of resizes was recorded and the time required. Each set of tests was performed for a different scale factor. The data structures are templates but all tests were performed with the 'int' data type. During this analysis, an observation was made of unnecessary operations in ABQ. The unnecessary operations were removed and ABQ original results are documented with ABQ improvements.

ABS Effects on Change in N:

Time goes up linearly with N, this is most clearly seen in Figure 5, there is a constant number of resizes of 9 for all Ns tested.

In figure 1, ABS scale factor 1.5, one can see that for large changing Ns and smaller scale factor the amount of time needed for large Ns rises quickly. This observation can most clearly be seen in Figure 3, points 2, 3, and 4 share the same number of resizes and the time grows linearly. The last point in Figure 3 requires 2 additional resizes and between point 4 and 5, the time required almost doubles in order to handle 50% more N.

On figure 2, we can see that for the same amount of resizes we see that the time grows more than linearly for larger Ns. Resizes at large N are much more costly than resizes at small N.

ABQ Effects on Change in N:

The same effects on change in N can be seen in the ABQ data structure the same as the ABS structure. The number of resizes for the ABQ and ABS are the same across the different figures. For the same scale factor and N, the ABS and ABQ graphs have the same inflection and proportional slopes to one another.

ABS Effects on change in Scale Factor:

A larger scale factor results in less resizes overall and a faster runtime of the data structure. At the smallest scale factor of 1.5 there were up to 92 resizes (Figure 1) compared to 9 resizes (Figure 5). For smaller scale factors, the time resizes require less time proportional to each previous resize. The effects for scale factor on time can most easily be seen when reviewing points 4 and 5 of Figure 1 and 3. For these 2 figures and points, there are 2 resizes between each. For Figure 1, from point 4 to 5 takes ~.5 secs, for Figure 4, from point 4 to 5 takes ~.7 secs.

A large scale Factor has a large effect on shortening the time needed for small N, this can be seen when comparing the time to the first point. At the smallest scale factor it is .190 secs and for scale factor 100 is .087. A smaller scale factor is proportionately faster for smaller changes in N, this would be due to there potentially being a great deal of wasted memory that is not used.

ABQ Effects of change in Scale Factor:

The same observations noted for ABS can be seen in the ABQ data.

The differences between ABS and ABQ:

The top difference is that ABQ takes longer overall. A review of the operations within push/enqueue and pop/dequeue. ABS's push has at least 3 operations. ABQ's enqueue has at least 5 operations. ABS pop has at least 5 operations and ABQ dequeue has at least 7 operations. ABS resize has at least 9 operations with a for loop that executes N times with 1 operation. ABQ resize has at least 8 operations and a for loop that executes N times with 2 operations. ABQ resize for loop operates 1 time for each element in the new array size, this results in extra unnecessary operations when resizing larger, this was corrected and discussed in "Improvement Analysis."

ABS pop/push combination has 8 operations and ABQ has 13 operations. This is equal to .615, looking at table 3, the first 3 scale factors are within +/-.10 of this ratio. As the scale factor gets larger the ABQ time required grows faster than ABS. This may be due to ABQ having 2 operations in each resize, the effect of this grows much more as the resize gets larger. The resize array for ABQ, when resizing larger, loops for as many times as there are elements in the larger array, the ABS array only loops as many times as the actual size (see improvement section).

Improvement Analysis:

ABQ on resize larger loops through as the full new larger array, this should be unnecessary. The ABS.h code was corrected to avoid unnecessary iterations and tested again on part 2 of the Zybooks Lab 3. The improved code is contained in ABQimproved.h and a chart of all its results on Figure 13. The results of this improvement are also seen in Table 4 and 5. The improvement brought the ratio of ABS to ABQ to ~.7 in line with expectations after observing the difference in the number of operations between push/pop and queue/dequeue. The ratio being ~.7 in testing, greater than the .615 is most likely due to ABQ resize having 2 operations in each resize and ABS resize has 1 operation. The original ABQ was able to perform its required job but did so in a less efficient manner. Further rigorous analysis of each operation could yield additional improvements.

Conclusions and Best Scale Factor:

The factors that have the largest impact on the data structures operating times are:

1. The number of resizes, minimizing resizes is one of the most important considerations in order to minimize the amount of time required.

2. Minimize the number of operations to improve performance. Subtle changes to the operations of the data structures could have large systemic benefits if even 1 or more operations is able to be eliminated. The ABQ improved shows what removing unnecessary operations can do for run time.

3. For this analysis a Larger scale factor results in better operation time. The major drawback of this method is you wind up with potentially 100x (or larger depending on scale factor) the amount of unused memory to used memory. Memory constraint is not an issue for the system the test was run on with the int data type, but more complex data types could begin to exhaust one's available memory.

4. Based on the results, If one was able to make an educated approximation of the size of N that would be needed, a scale factor that changes relative to the size of the data structure could possibly result in better performance and minimize unused memory. A large scale factor initially to fill up the array that levels off could potentially get most of the benefits with minimal drawbacks.