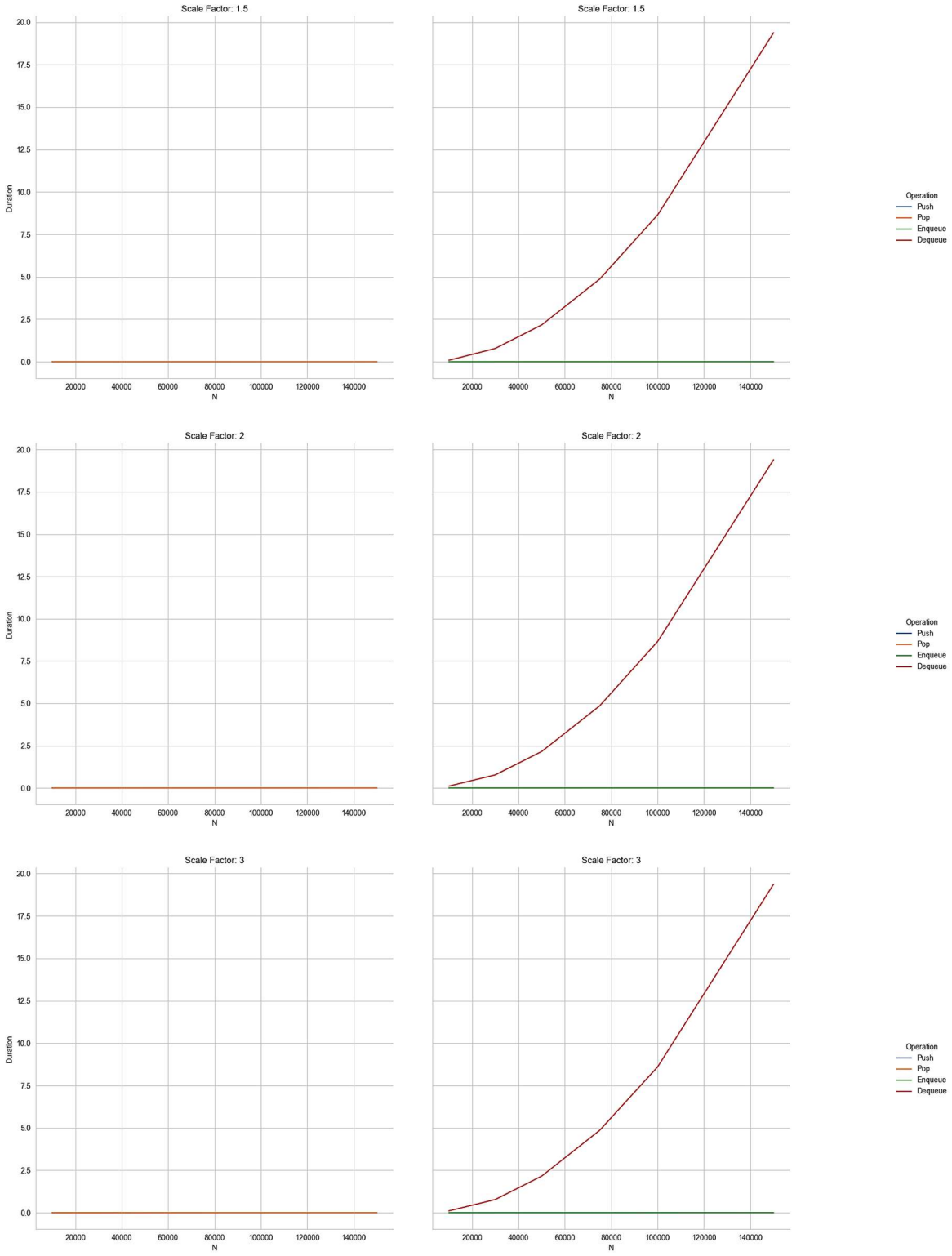
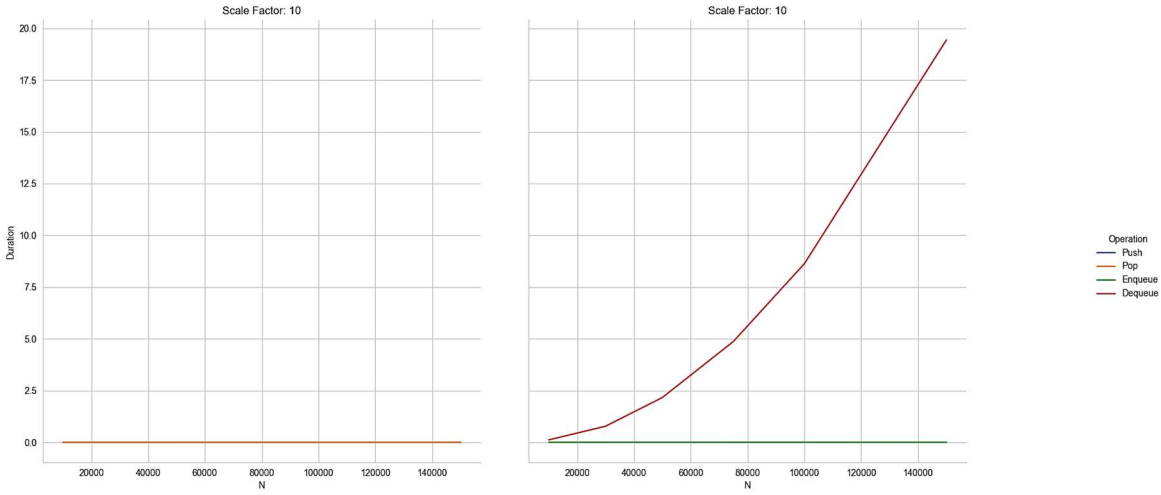
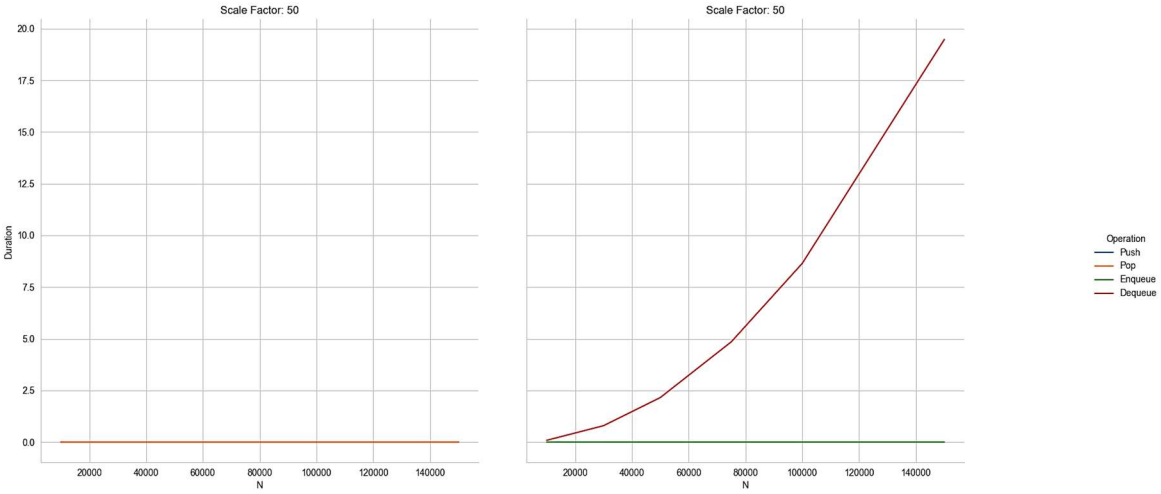
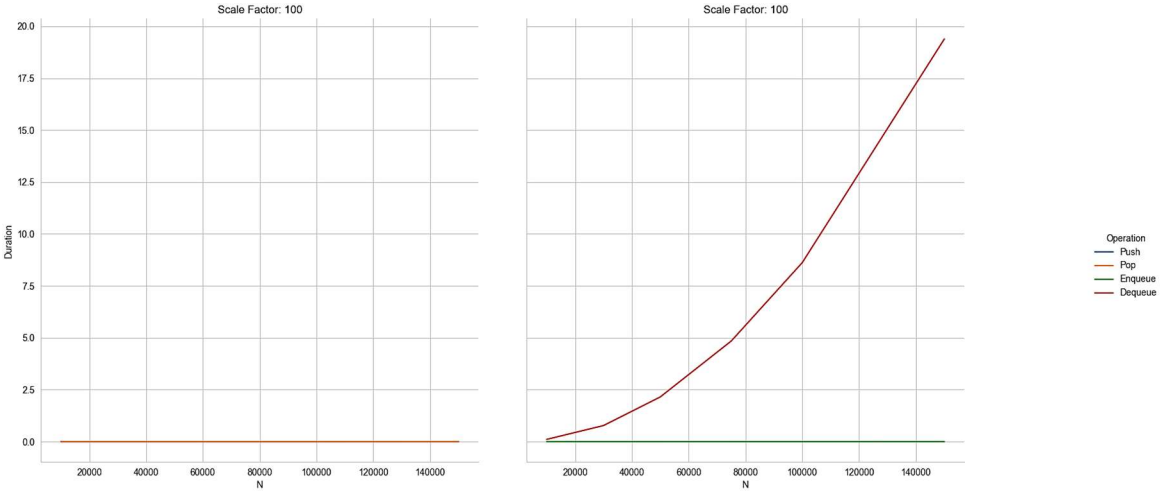
After an initial look at the data one thing in particular stood out, dequeue operations were noticeably slower than the other the other functions.

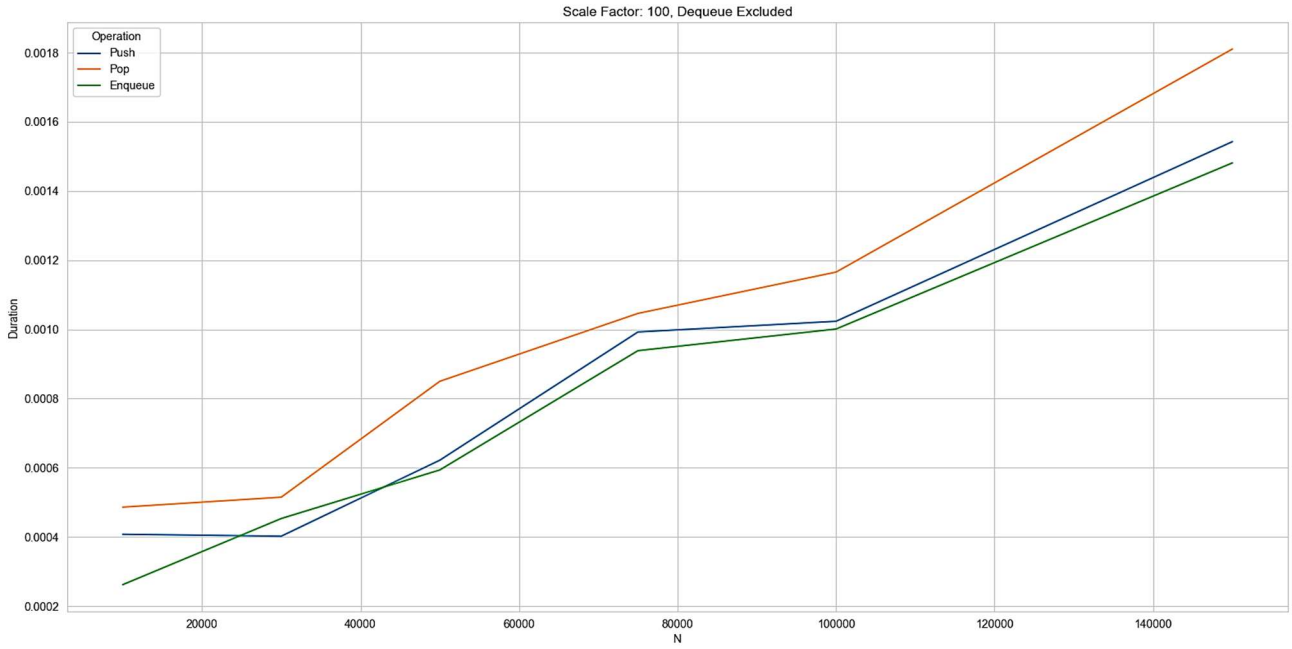



Scale Factor: 10
Duration
N
Operation
Push
Pop
Enqueue
Dequeue


Scale Factor: 50
Duration
N
Operation
Push
Pop
Enqueue
Dequeue


Scale Factor: 100
Duration
N
Operation
Push
Pop
Enqueue
Dequeue

To highlight the disparity, this is a graph of excluding dequeue at a scale factor of two.



All three of these operations run in an insert/delete loop which is O(N), meaning the underlying operation runs in constant time. **Unfortunately, the dequeue appears to operate at O(N)**. I suspect that this block of code is the culprit:

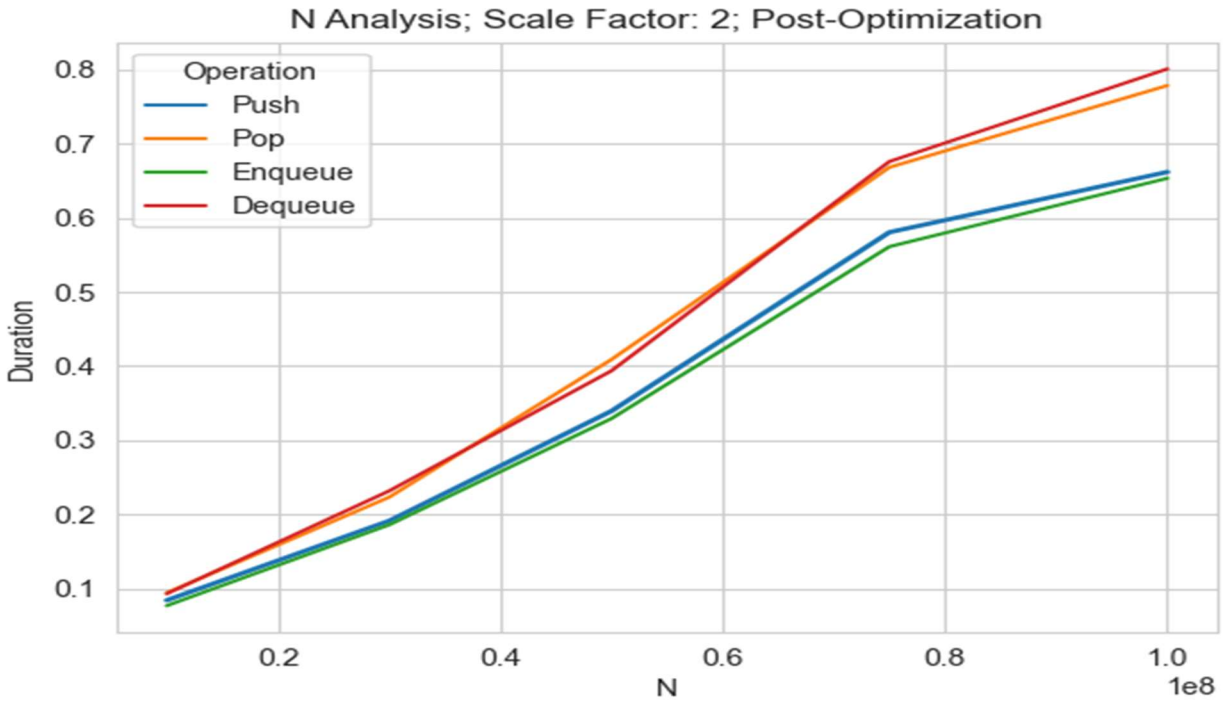
```
// Shift remaining items left
    if (m_size > 0) {
        for (unsigned int i {}; i < m_size; ++i) {
            m_data[i] = m_data[i + 1];
        }
    }
```

The goal of this code block is to keep the queue data in the range of 0->size-1 inside the underlying C-array. This was done under the assumption that dequeue and enqueue would be interleaved, which without this code would result in resizing operations being done on arrays which may still be mostly empty. Several options exist to fix this. One option would be to utilize a circular array as the underlying implementation. This would require a front and back marker that could loop around the end of the array. If maintaining a specific max capacity was not a requirement, utilizing a linked list as the data structure would allow O(1) run time for both insertion and removal of elements, though this would increase the memory requirements. For this specific requirement, where usage is limited to one insert loop followed by one removal loop, adding a front marker seems to be a good compromise.

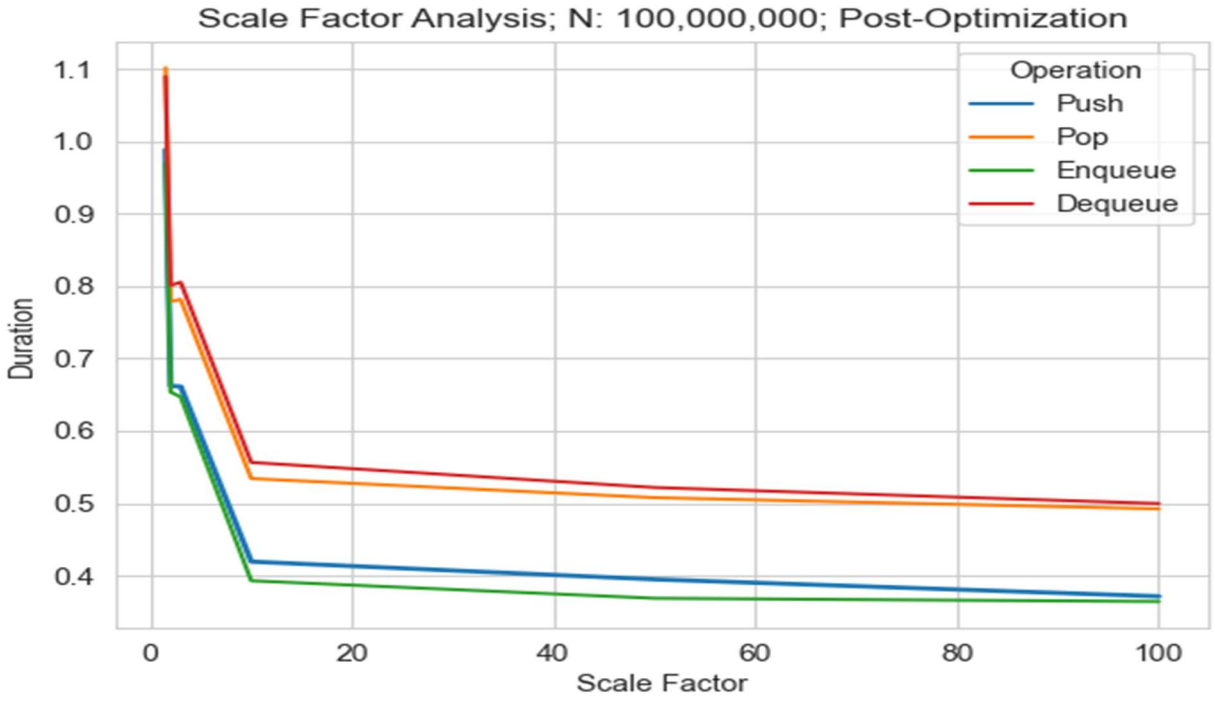
Implementing this change required changes to the member functions for increasing and decreasing the size of the underlying array, specifically the portion that creating a deep copy of the original data.

```
    for (unsigned int i{}; i < m_capacity; ++i) {
        new_data[i] = m_data[head + i];
    }
```

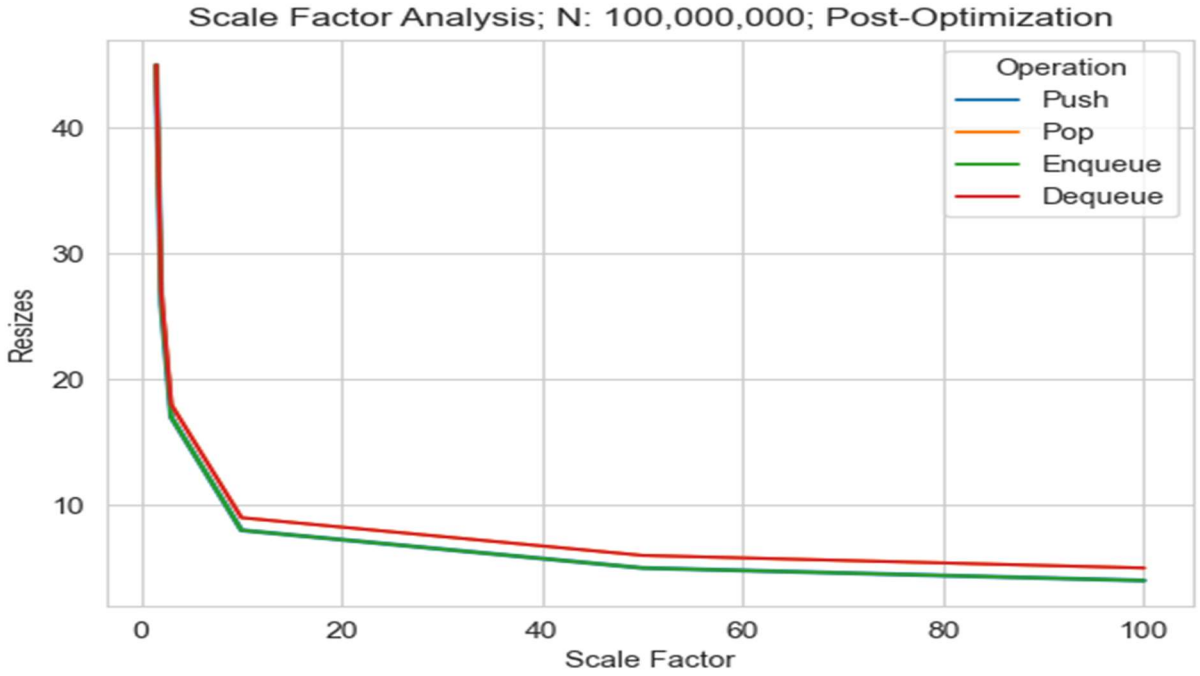
Peek and dequeue functions now return the data at the head index location. Additionally, dequeue increments this index to point at the new start of the data. Finally, the decrease capacity function resets this index to 0 after creating the deep copy of the original data. After these edits, the same plot at scale factor two looks like this:



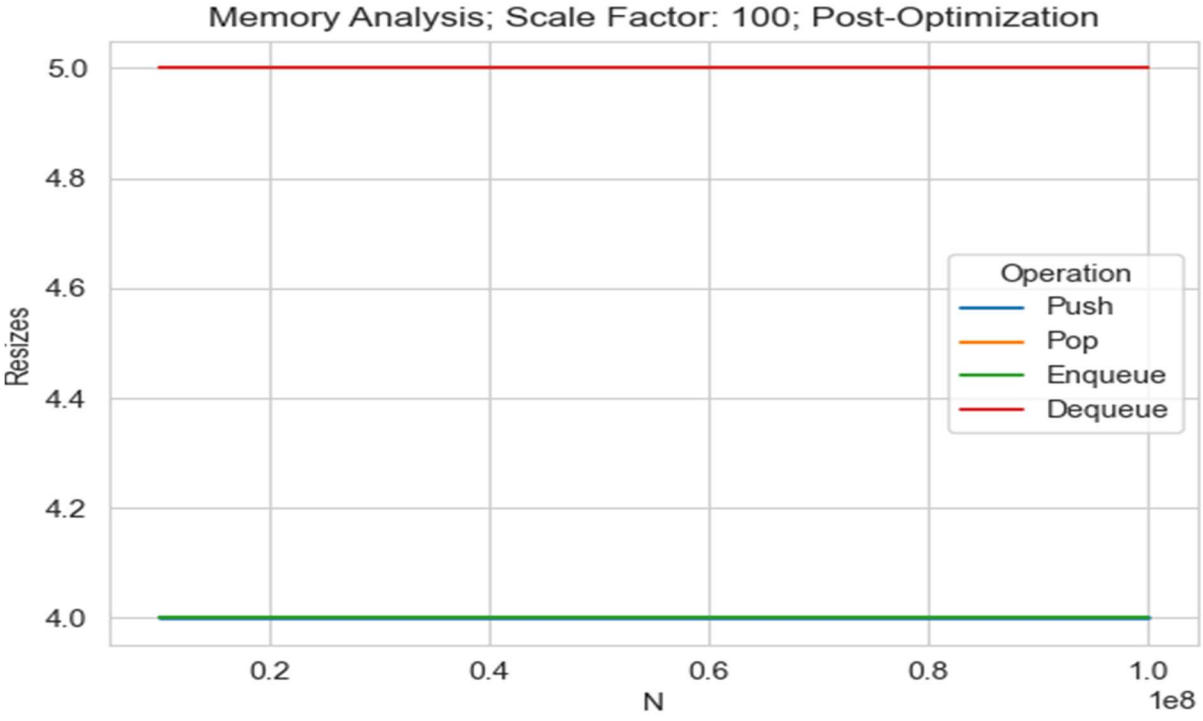
Now we can begin to answer the analysis questions. Based on the plot above, these data structures **scale linearly with N, with the removal operations (Pop and Dequeue) having a slightly higher constant factor than the insertion operations**. Looking at the effects of the scaling factor:



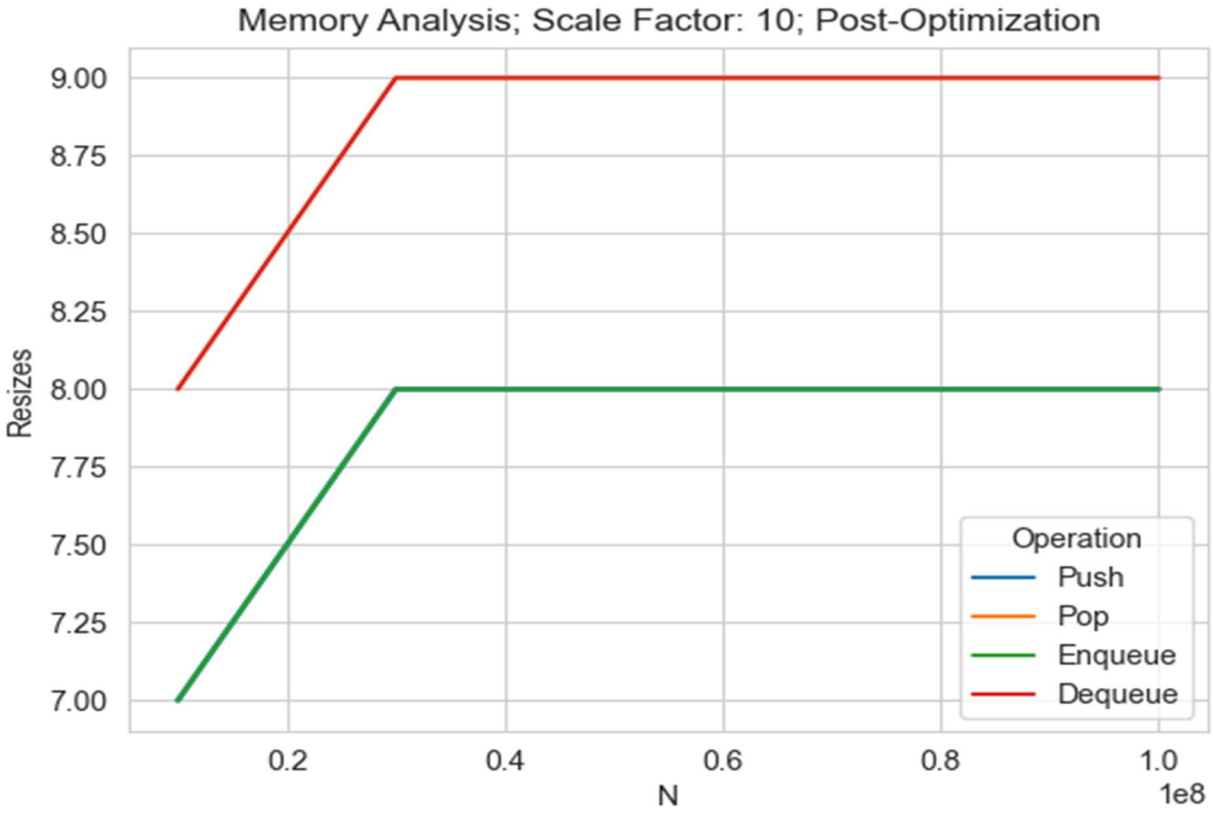
From this, we can see that **duration drops rapidly with an increase in the Scale Factor until the factor is 10**. From there, further increases continue to decrease the duration, but a much slower rate. This lines up with a look at the number of resizes performed:

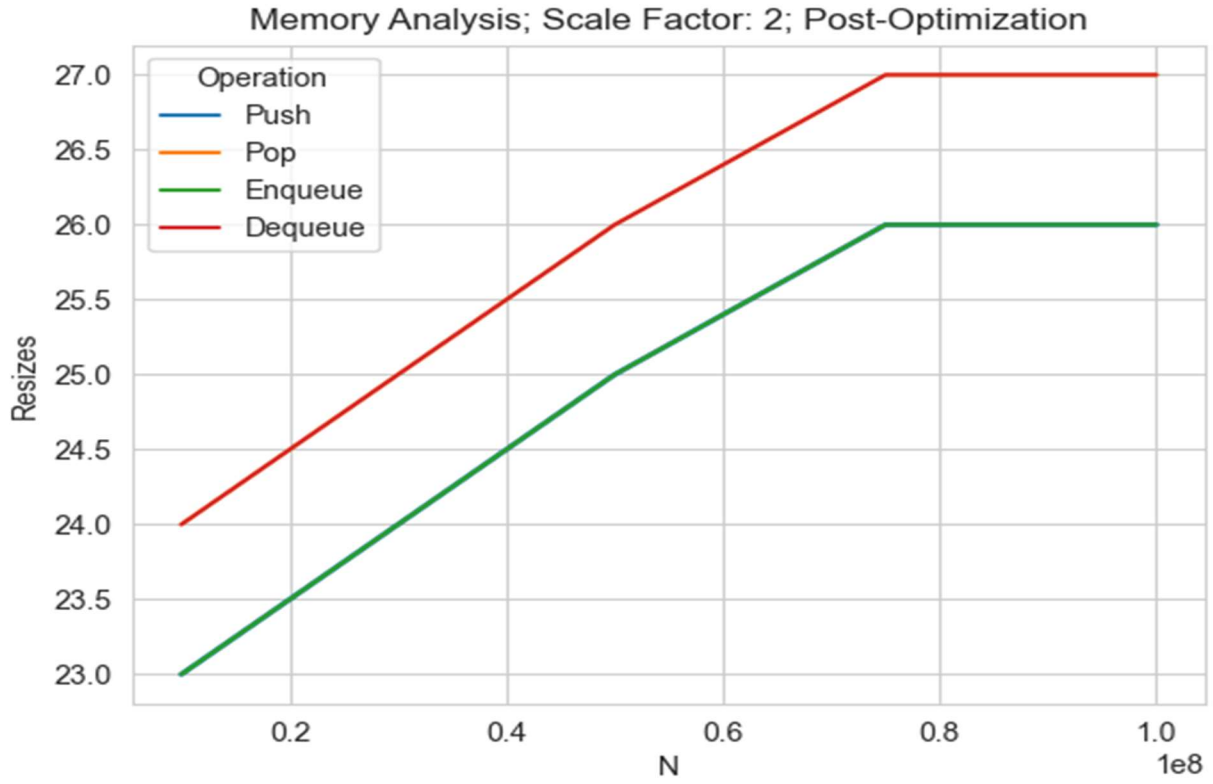


This chart shows the same pattern as before, a rapid drop until the Scale Factor is 10, then a slower decrease as the Scale Factor continues to increase. From this we can conclude that **duration is inversely proportional to the Scale Factor**. Small scale factors result in many resize operations, though the downside is that a high Scale Factor many involve the program reserving far more memory than is required. This can be visualized in the following chart:



When the Scale Factor is 100, the same number of resizes are performed with an N of 10,000,000 as with 100,000,000. This indicates that **high Scale Factors result in the program possibly reserving far more memory than is required**. Looking at this same chart at a Scale Factor of 10 and two yield this:





A Scale Factor of two yields a far more gradual increase in memory used. Based on these factors, the best Scale Factor would depend on the specific use case: the size of N, as well as run time and memory usage requirements. In general, it appears that a good starting point is **somewhere between two and ten**. For the final question, the charts show **no discernable difference in performance between the stack and queue implementations**.

NOTES:

Testing on the initial implementation was done with smaller N values due to the slow dequeue operation. Once fixed, the N values prescribed in the assignment were used. Data analysis and plotting were done with python, pandas, and the seaborn library.