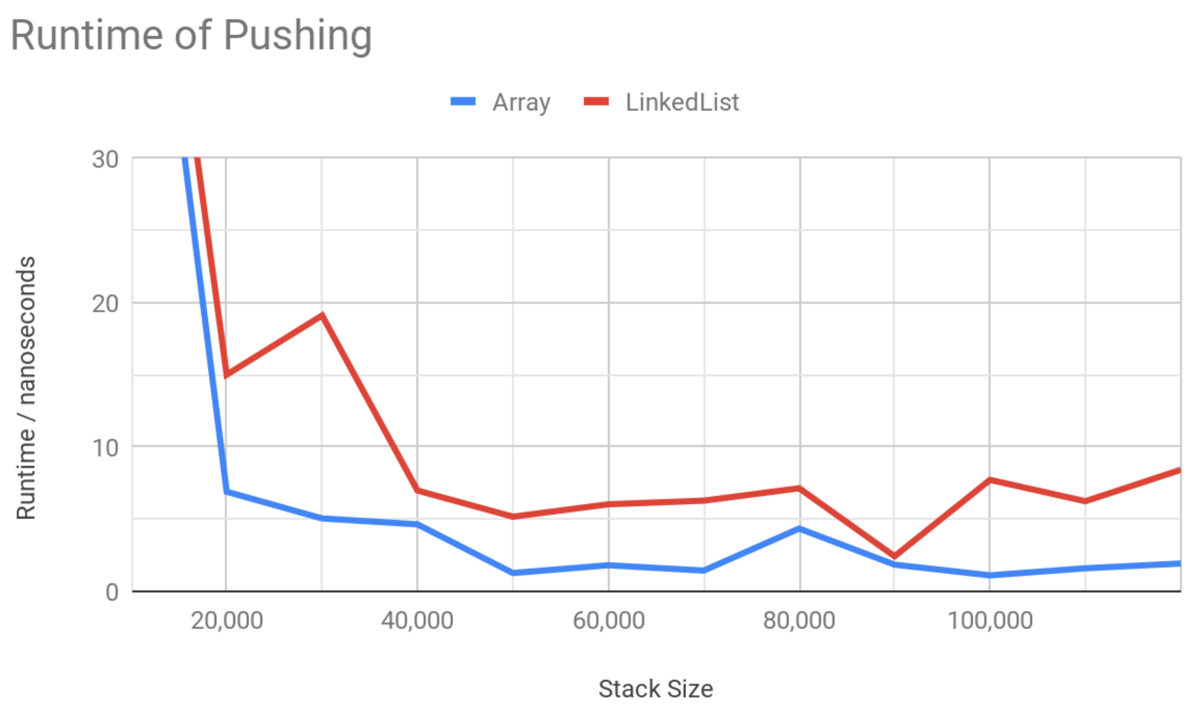
Assignment 6: LinkedList Stack

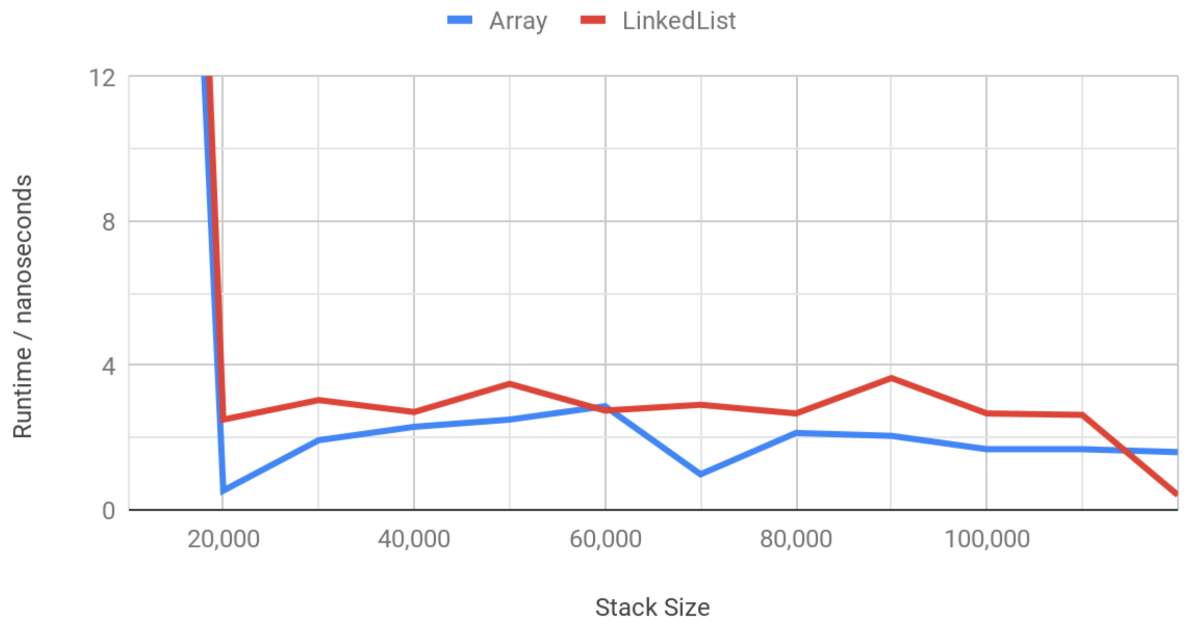
Analysis Document by Qianlang Chen (u1172983)

As in all previous assignments, I gladly had Brandon Walton as my programming partner in this assignment. Brandon did an excellent job of figuring out the logic behind most of the functions and wrote high-quality code that sped up our completion of the classes. However, I plan to switch my partner in the next programming assignment to get more experience of working with different people as well as helping them, and this would also apply to Brandon.

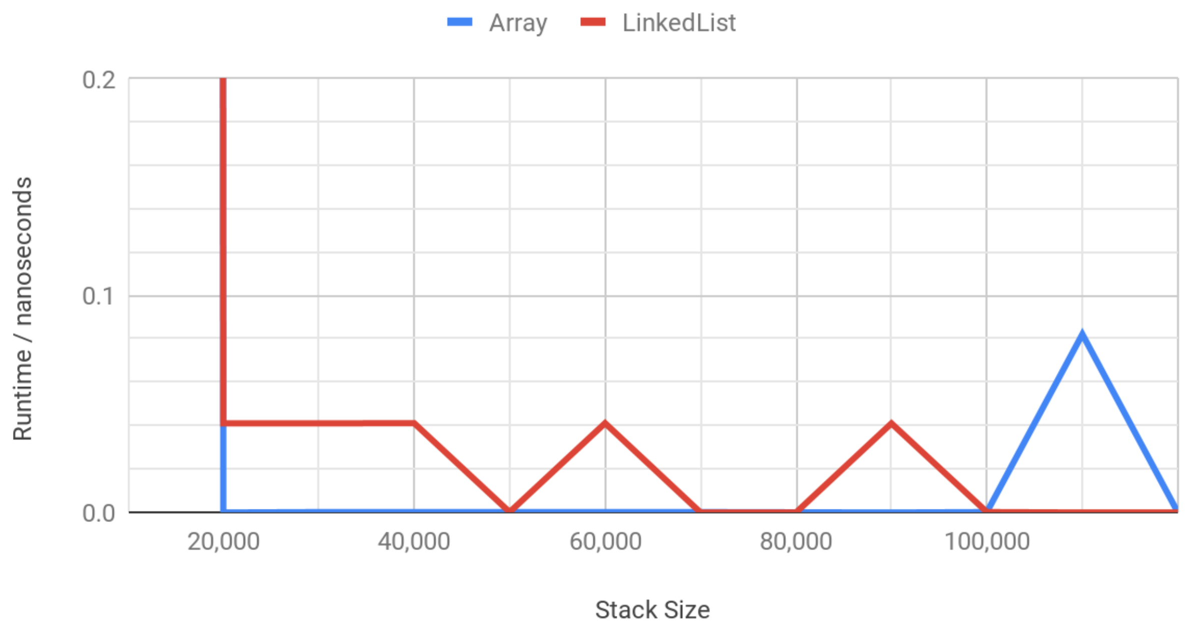
In this programming assignment, we encountered and used a new data type — stacks, and we saw that it could be implemented with either of the two data structures: arrays or linked lists. We compared the two choices by timing the most critical three functions of stacks: *push()*, *peek()*, and *pop()*, and here are the results:



Runtime of Popping



Runtime of Peeking



From our results, we can see that the runtime behaviors of the three methods of stacks implemented by either arrays or linked lists are all **constants** because the runtimes are so small (given that the data sizes are large enough) and do not seem to diverge. This conclusion makes sense because in all of our *push()*, *peek()*, and *pop()* methods, we were only interested in, and dealt with, the most recently added element. This also applies to arrays, because even though we had to increase the size of the arrays when they ran out of space for new elements, we did this so rarely that the effort became insignificant compared to the data size. However, in most of the times, the runtimes of stacks implemented by arrays still seem to be lower than those implemented by linked lists, so we used array-implemented-stacks in our *BalancedSymbolChecker* class. Though the differences are tiny, plus that the runtime behavior is constant, it might as well not matter.

An interesting question arose during our development of *BalancedSymbolChecker* class where we reported the line and column numbers to the caller when we found an unmatched symbol: how would we also keep track of the opening symbol of the unmatched one? My strategy to that question is, rather than only pushing the symbol-per-se onto the stack, **create a new class that helps record not only the symbol, but also the line and column numbers, and push that onto the stack** instead. That way when we find an unmatched symbol, we can peek the stack which gets an instance of the class we created, then we can quickly get the symbol and its line and column numbers.