Carsten Hood
CSCE 221H-200

# Program 1 Report
# Evaluation and Comparison of Stack Implementations

## Introduction

This report details the setup and results of efficiency tests of operations of two implementations of the Stack ADT: array_stack, which uses an array, and list_stack, which is implemented with a linked list. Each stack contains two functions for returning its number of contents, size() and my_size(). Both stacks implement the push function, which serves to add an element to the stack. This report focuses on comparing these size and push algorithms.

## Theoretical Analysis

The complexity of the size() functions is O(1). Both array_stack and list_stack also contain an alternative size function, my_size(). In the array-based implementation this algorithm copies all *n* elements into a temporary array and "pops" them off, incrementing a counter with each removal and returning the final value as the size. The complexity of the copy is O(n) and that of the sequence of removing *n* items is O(n), as each element must be "popped", so the overall operation is O(n) + O(n) = O(n). In the list-based stack, my_size() similarly increments a counter as it iterates through all of *n* contents and returns the final counter value. This algorithm also takes O(n), as it visits each node once.

The complexity of the Stack ADT push algorithm depends on its implementation. In an array-based stack it becomes necessary to expand the underlying array when the number of contents reaches the capacity. Resizing in array_stack is implemented in two ways: copying elements into a new array with a capacity increased by double or by a fixed increment, *k*. In the second method, the array must be expanded every *n/k* times. This may be more efficient when fewer items are pushed and expansion is rarely necessary; however, the amortized cost is O(n) because every *k* pushes all *n* elements must be copied. At higher ratios of *n* versus *k,* this method will become increasingly inefficient. Resizing the array by doubling requires resizing every $x = \log n$ pushes: whenever the stack size *n* reaches an exponent of 2, it expands again. Averaging the time of multiple push operations results in O(1) amortized time. While this method still involves copying *n* elements when capacity is reached, such resizing must occur only when *n* pushes are made after a prior resizing. It can be thought that each of these *n* push calls that do not require resizing save up time for the next expansion. This is superior for handling a variable number of push operations.

In the list-based stack, no resizing is necessary as elements are not stored contiguously; the push operation is always O(1).

The complexity of the pop and top functions is O(1). This holds true in both implementations. Access to the top of the stack to remove it or return its element is immediate and does not depend on how many elements are in the stack.

**Experimental Setup**

The experiments described in this report were performed on a Mac computer running OS X 10.9.1 with a 2.7 GHz Intel Core i7 processor and 8GB RAM memory. C++ code was compiled using Apple LLVM 5.0.

The standard library ctime was used to time the operations. Executions were repeated at least ten times as necessary and averaged. Over a million executions were averaged for the highest input sizes, and the exceptionally efficient size() function was repeated 64 times as much as the other algorithms to compensate for clock inaccuracy.

A vector of doubles was used as input for testing the running time of all functions. The vector was initialized prior to timing with a set capacity ranging from 2 to 1,048,576 and populated with random double values. To test the size and my_size algorithms, a stack was initialized and all values in the vector were added prior to timing. To test the push operations, all *n* elements in the vector were repeatedly pushed into the stack in each iteration.

The testing file was templated so that it could be used to test both array_stack and list_stack. The test performed by the test function for each call was determined by a string parameter, which_test, which was either “push”, “size”, or “my_size”. In this way all tested functions and both stack implementations concisely share timing code, and the differences in results can be relied on to depend entirely on the differences in the underlying implementation.

**Experimental Results**

Figure 1 plots stack size vs. time in seconds of the size and my_size functions of both the array-based and list-based implementations on a log-log plot. The standard size algorithm appears constant and similar between both implementations. It is clearly much more efficient than the versions of my_size, which both scale linearly at a similar rate. The list-based stack’s implementation of my_size is more efficient than the my_size of the array-based stack. This is more obvious at low values of *n* and may be partially due to the cost of the copy operation of the array-based implementation. For very small values of *n*, both versions of my_size near the speed of size; however, the time required by size remains relatively constant while my_size costs more time with every increase in stack size.

Figure 2 compares the running times of the push algorithms of the array-based and list-based stacks on a log-log plot. The number of consecutive push operations, *n*, is shown on the horizontal axis, and the total running time in seconds is shown on the vertical axis. In the array-based implementation, the underlying array is initialized to a capacity of 2 and resizes by doubling its capacity. As expected, both lines run roughly linearly; this holds

with the analysis of the push operations as being roughly constant, as a linear number of calls are being made to push. The constant nature of an average push call is represented in the following plot, Figure 3. Overall, though both algorithms scale at a similar rate, that of the array-based stack appears significantly more efficient for all input sizes. However, the array-based stack is slower at low values of *n*, when it must double its capacity more frequently.

Figure 3 shows the averaged cost of a single push operation for varying sizes on a log-log plot. The vertical axis is the result of dividing the time taken by multiple push operations by their number, *n*, and the horizontal axis is simply the number of operations, or the final size of the stack. The result is two roughly constant lines, representing the array-based and list-based stack implementations. This plot can be used to determine the Big-O constants *C* and *n0* for the push functions *f(n)* of both implementations, where $0 <= f(n) <= C*g(n)$ for all $n >= n0$. After an initial sharp bump at small sizes, the stack-based implementations settles into a constant path for all values of *n* greater than or equal to 32, bounded by the constant 2.983E-08. So $C = 2.983E-08$ and $n0 = 32$. The list-based stack is more consistent, so its maximum average time is used: $C = 1.278E-07$. This constant bounds the cost of a single push operation for all sizes *n,* so the constant $n0 = 0$. These constants are represented with arrows on the plot. Figure 3 primarily serves to demonstrate that the average cost of a push operation is O(1), no matter the size of the stack or number of consecutive push calls.

Figure 4 is a log-log plot showing the averaged running time of the array-based stack's push algorithm using four strategies of resizing: the incremental and doubling methods of resizing are represented twice each with different initial capacities, 2 and 1000, and the implementations that resize by a fixed amount *k* use $k = 10000$ and $k = 1000$ respectively.

For low stack sizes, the stack that initiates the underlying array to 2 and expands it by doubling is the slowest, having to increase its capacity at every power of 2. The efficiencies of the stacks where the underlying array is initialized to 1000 surpass it in speed and are equally efficient until *n* reaches 1000; then the stack using the doubling strategy maintains a constant path, while the stack increasing incrementally by 1000 scales linearly. Here, both implementations are surpassed in efficiency by the stacks initialized with a capacity of 2. This reveals that a higher starting capacity results in superior performance only for a short range of sizes. It also demonstrates that an incremental strategy may be faster for certain ranges; however, geometric expansion is more reliable for unknown ranges. The algorithm that initiates the array capacity to 2 and increases it incrementally by 10000 presents this more clearly: the averaged speed of its push operation is faster than all others between the stack sizes 1024 and 8192, which fall within the range of its first expansion. However, as *n* continues to increase, this implementation becomes increasingly inefficient. The array using an initial capacity of two and the doubling algorithm turns out to be the most efficient for large sizes, just as it was the least for small sizes. For this reason, this strategy of initializing to 2 and expanding by doubling is chosen to represent the array-based implementation in its comparison to the list-based implementation in Figure 2 and Figure 3.

When the push algorithms in Figure 4 are compared to the plot of the list-based push algorithm in Figure 3, all of the stack-based forms of push appear faster up to a certain size. At larger sizes the stack implementations using incremental expansion become less efficient that the list-based stack as their linear scale reveals itself. However, the stacks implementing the doubling strategy remain consistently much faster than the list-based implementation.

Valgrind was employed through a Unix server to ensure memory efficiency in the list-based and array-based stack implementations.

**Summary**

This report compares and evaluates several algorithms of array-based and list-based implementations of the Stack ADT. Logarithmic plots demonstrate the running time of these stack operations in relation to size. Two differing functions intended to determine the number of elements in a stack, size() and my_size(), are analyzed and demonstrated to have similar complexities of O(1) and O(n) respectively in both implementations. Various implementations of the push algorithm are examined. Comparison reveals that the O(1) amortized complexity of the push algorithm of an array-based stack implemented with geometric expansion is preferable to the O(n) cost of an incremental implementation over a large or unknown span of input sizes. The array-based implementation of push is also demonstrated to be more efficient than the similarly O(1) list-based implementation for all sizes.

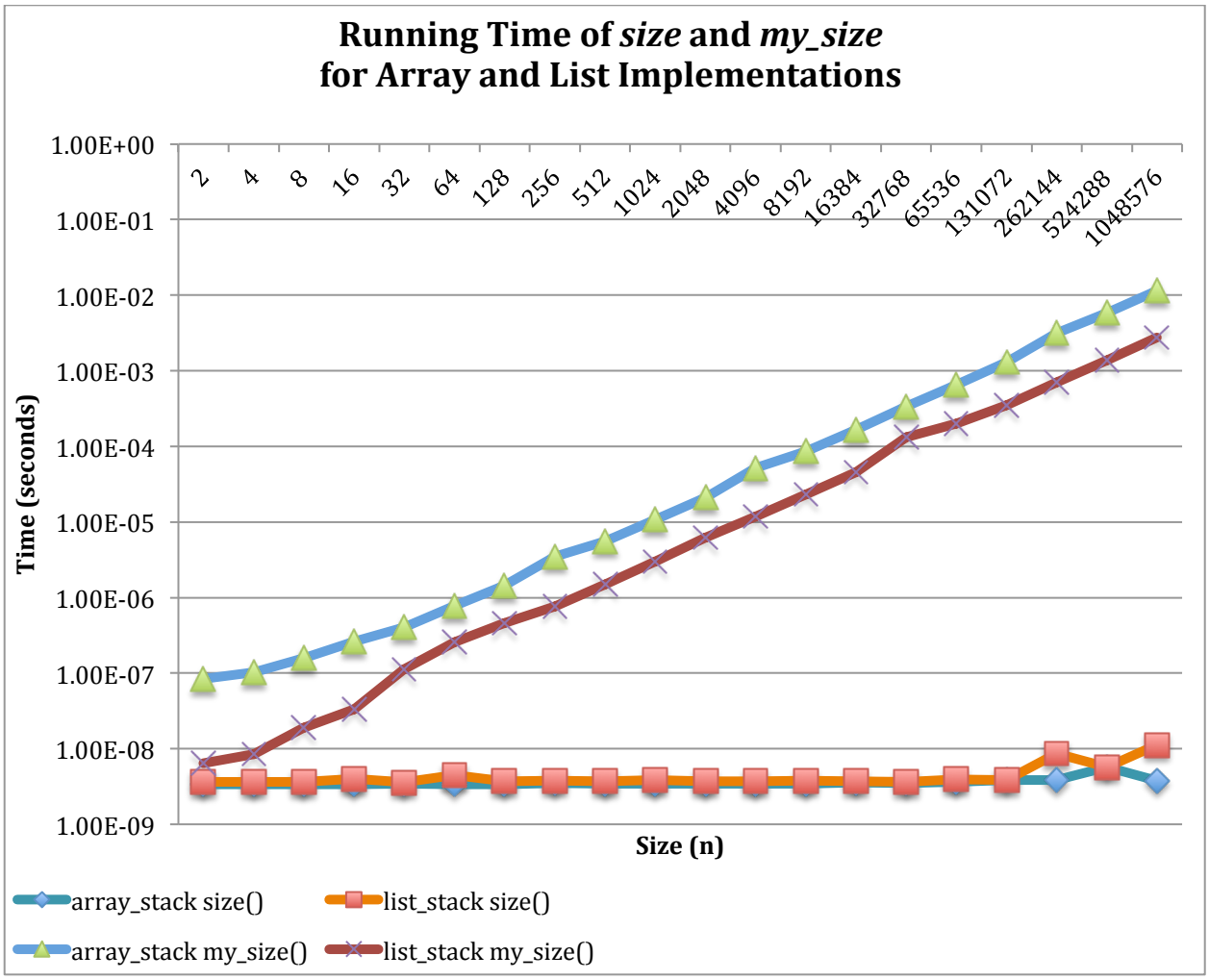


Figure 1: Execution time in seconds vs. stack size for the size() and my_size() functions of the array-based and list-based stack implementations on a log-log plot.

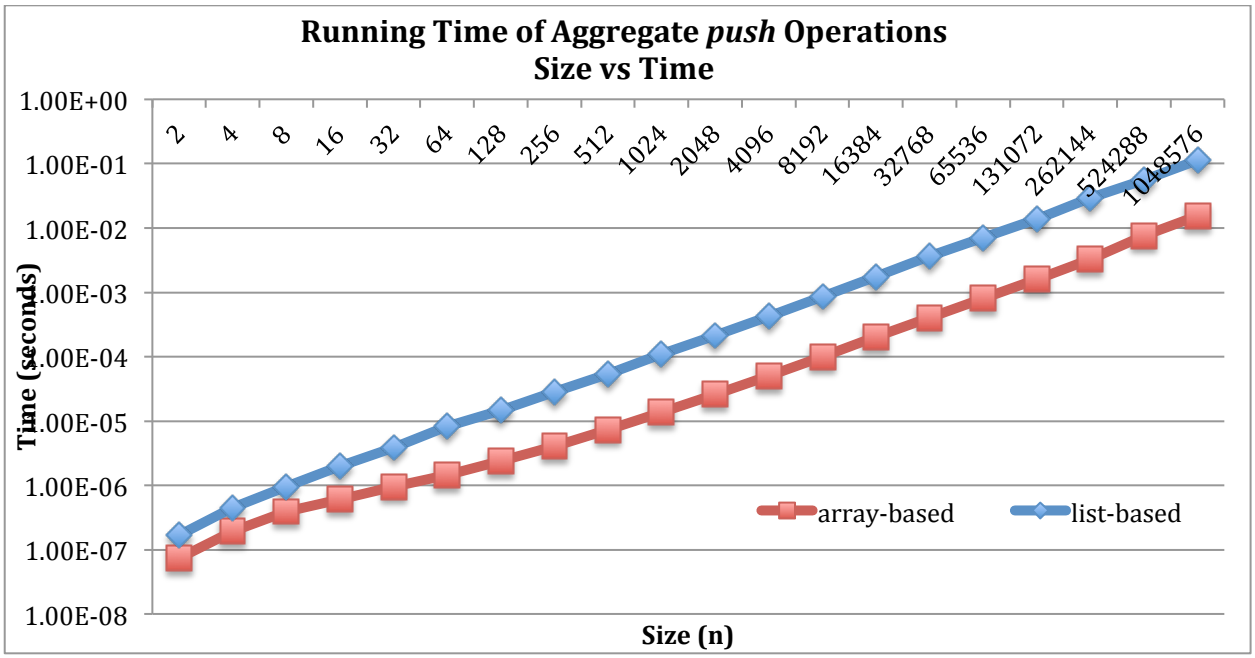


Figure 2: Execution time in seconds vs. number of push() functions of the array-based and list-based stack implementations on a log-log plot. In the array-based stack, the underlying array's capacity is initialized to 2 and dynamically resizes by doubling.

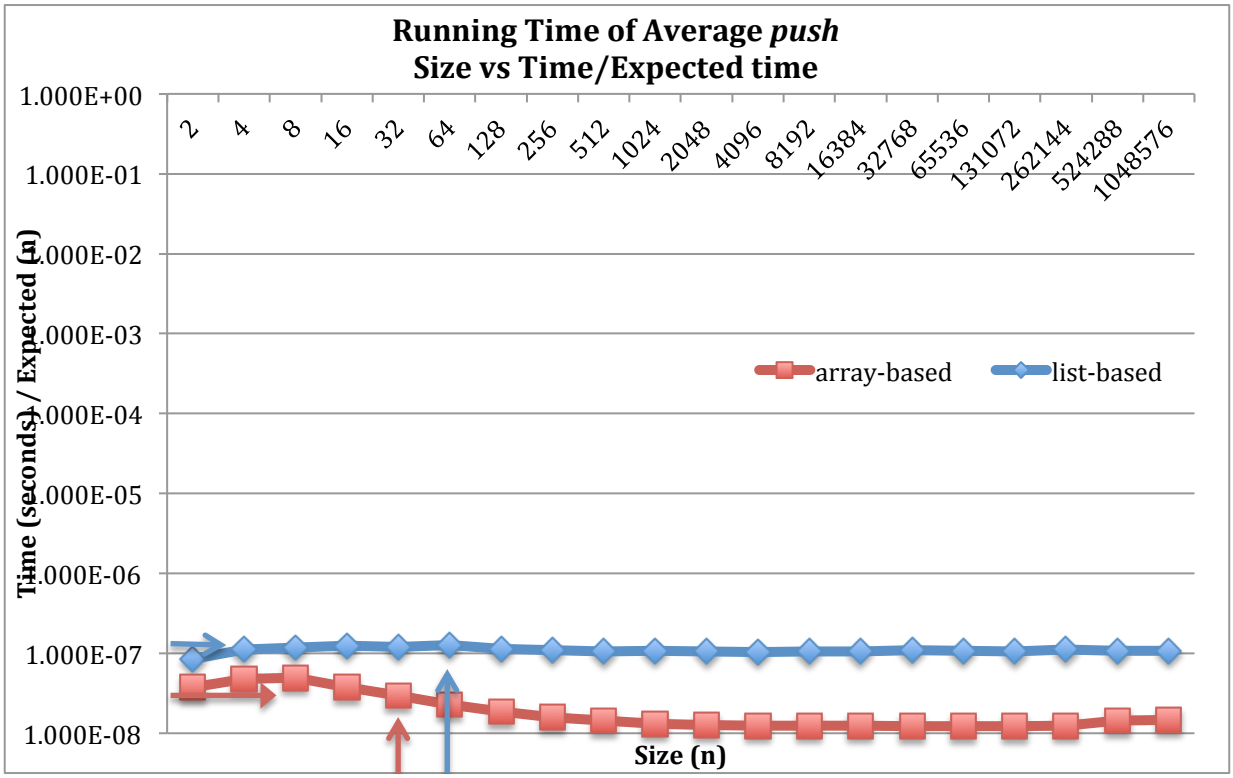


Figure 3: Execution time in seconds vs. stack size for the push() functions of the array-based and list-based stack implementations on a log-log plot. The time taken by *n* push operations is divided over *n* to produce an average time per push. Arrows represent the values of the chosen Big-O constants.

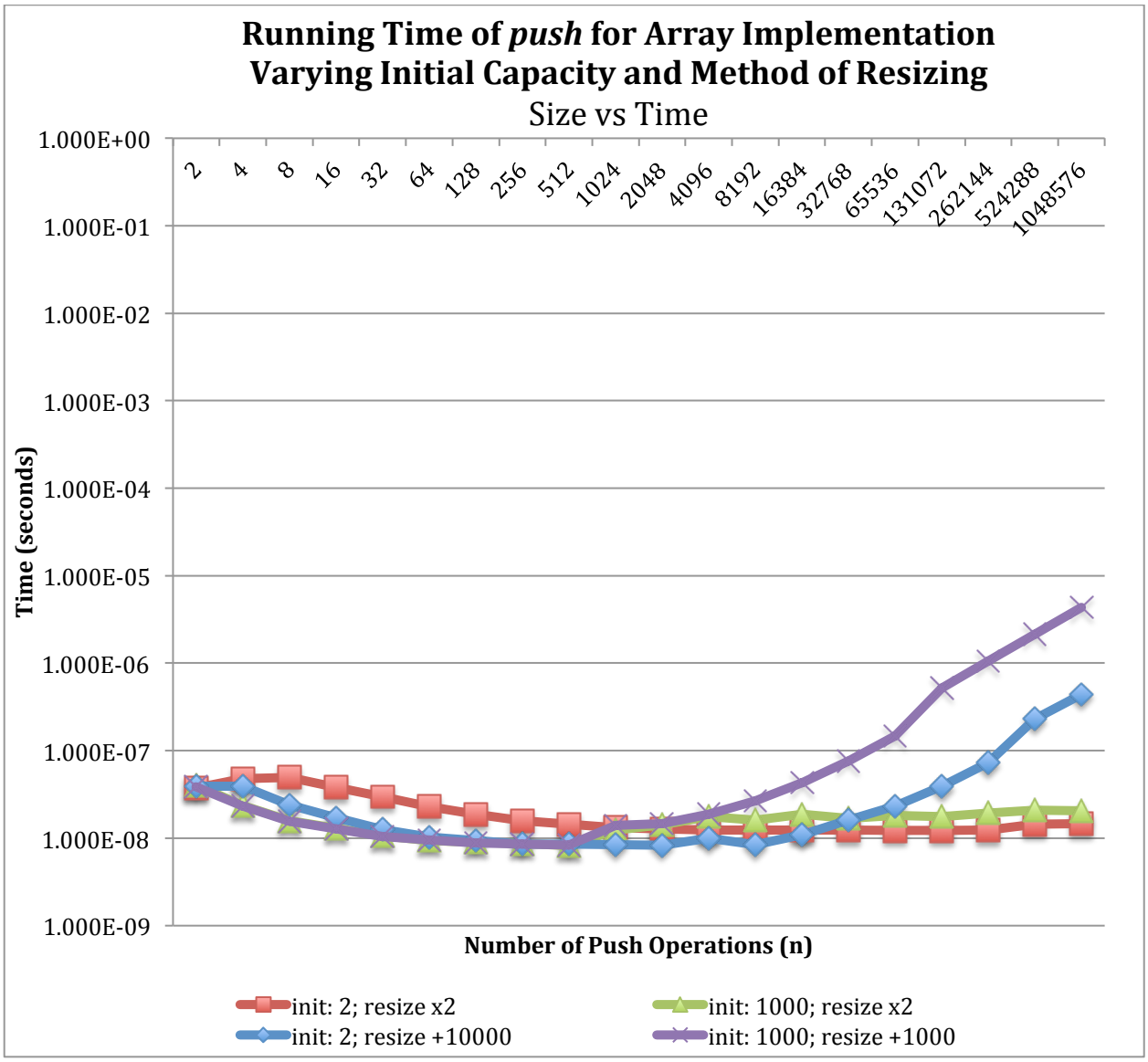


Figure 4: Average execution time in seconds vs. number of push operations for varying implementations of the push function of an array-based stack on a log-log plot. The initial capacity of the underlying array and the resizing strategy vary as follows:

1 Red: Array is initialized to 2 and resizes by doubling capacity.
2 Green: Array is initialized to 1000 and resizes by doubling capacity.
3 Blue: Array is initialized to 2 and resizes by increasing capacity by 10000.
4 Purple: Array is initialized to 1000 and resizes by increasing capacity by 1000.