Carsten Hood

CSCE 221H-200

**Lab Report**

**Performance Evaluation of STL Sort**

**Introduction**

This report details the setup and results of scalability 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 function of the Stack ADT is O(1). Both array\_stack and list\_stack 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 operation is O(n) + O(n) = O(n). In the list-based stack, my\_size() increments a counter as it iterates through all of *n* contents, returning the final counter value. This algorithm also takes O(n), visiting each node once.

The complexity of the Stack ADT push function is constant. However, in the 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 value, *k*. The second method may be more efficient when fewer items are pushed and resizing is rarely necessary; however, the cost is O(n) because every *k* pushes all *n* elements must be copied. At high ratios of *n* versus *k,* this method will be highly inefficient. Resizing the array by doubling it takes O(1) amortized time and is superior for handling an unknown or extensive number of push operations. While this method still involves copying *n* elements when capacity is reached, such a resizing must occur only when *n* pushes are made after a prior resizing. It can be thought that each of these fast *n* push calls that do not require resizing save up time for the next resizing. In the list-based stack, no resizing is necessary as elements are not stored contiguously; therefore the push operation is always O(1). However, at large values of *n* the list elements may waste memory space in-between their scattered locations.

The complexity of the pop and top functions of the Stack ADT 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 quick size() function was repeated 2^6 times as much as the others to compensate for the clock accuracy.

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 size() and my\_size() a stack was initialized and all values in the vector were pushed in prior to timing. To test the push() operations, all the elements in the vector were repeatedly pushed into the stack for every iteration.

The test function 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, that 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 counted to depend entirely on the differences in the underlying implementation.

**Experimental Results**

Figure 1 shows the running time of the array-based stack’s push() function using both methods of resizing, as well as the list-based stack’s push() function. Because the underlying array is initialized to 1000, both array-based methods of resizing share a line until *n* = 1000 is reached; even then the divergence is hardy noticeable until *n* reaches 8192. This demonstrates that resizing by a fixed amount *k*, in this case 1000, is not significantly inferior to doubling capacity up to a certain point. However, after this point the line representing the fixed resizing method rises sharply, so that the time taken by its final push operation is almost 5 seconds, when the doubling method takes close to a tenth.

Figure 2 shows Time/Expected Time, or Time/n\*log(n), vs. size on a log-x plot. This plot reveals the constants C and n0 from the equation: 0 <= f(n) <= c\*g(n) for all n >= n0. From the plot, C is 5.005E-09 and n0 is 16. After this pair (n0, C), all points fall beneath the line y = C. The red and blue arrows mark C and n0.

Figure 3 shows the line represented by C\*Expected Time, or 5.005E-09 \* (n\*log(n)), in red, over the running time of STL Accumulate, in blue. Where the lines intersect at n = 16 represents n0.

**Summary**

This report shows an experimental performance evaluation of the STL Sort algorithm. Its O(n\*log(n)) complexity is demonstrated through experimental tests, with asymptotic constants and n0 shown.

Figure 1: 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. In the array-based stack, the underlying array’s capacity is initialized to 2 and dynamically resizes by doubling.

Figure 2: 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 1: Execution time in seconds vs stack size 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 method of resizing vary as follows:

1 Red: Array initialized to 2 and resizes by doubling capacity.

2 Green: Array initialized to 1000 and resizes by doubling capacity.

3 Blue: Array initialized to 2 and resizes by increasing capacity by 10000.

4 Purple: Array initialized to 1000 and resizes by increasing capacity by 1000.