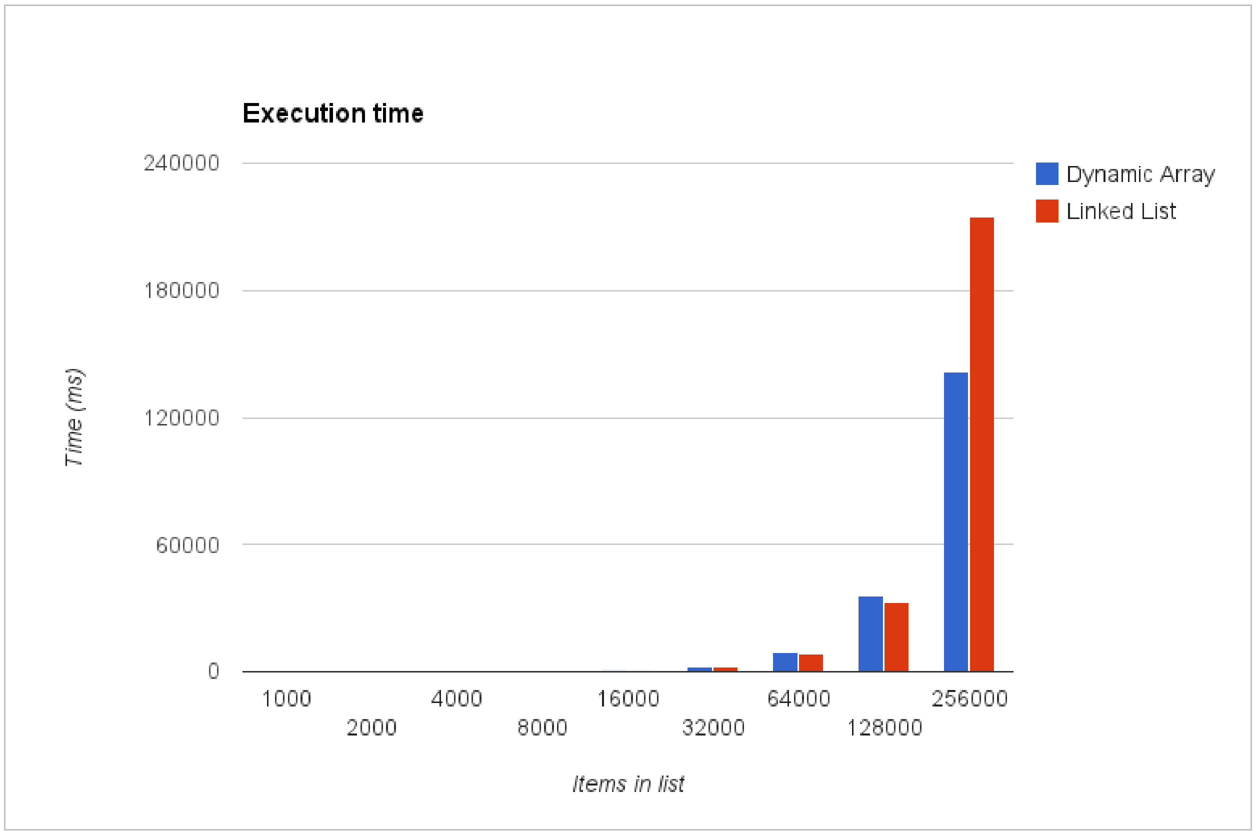
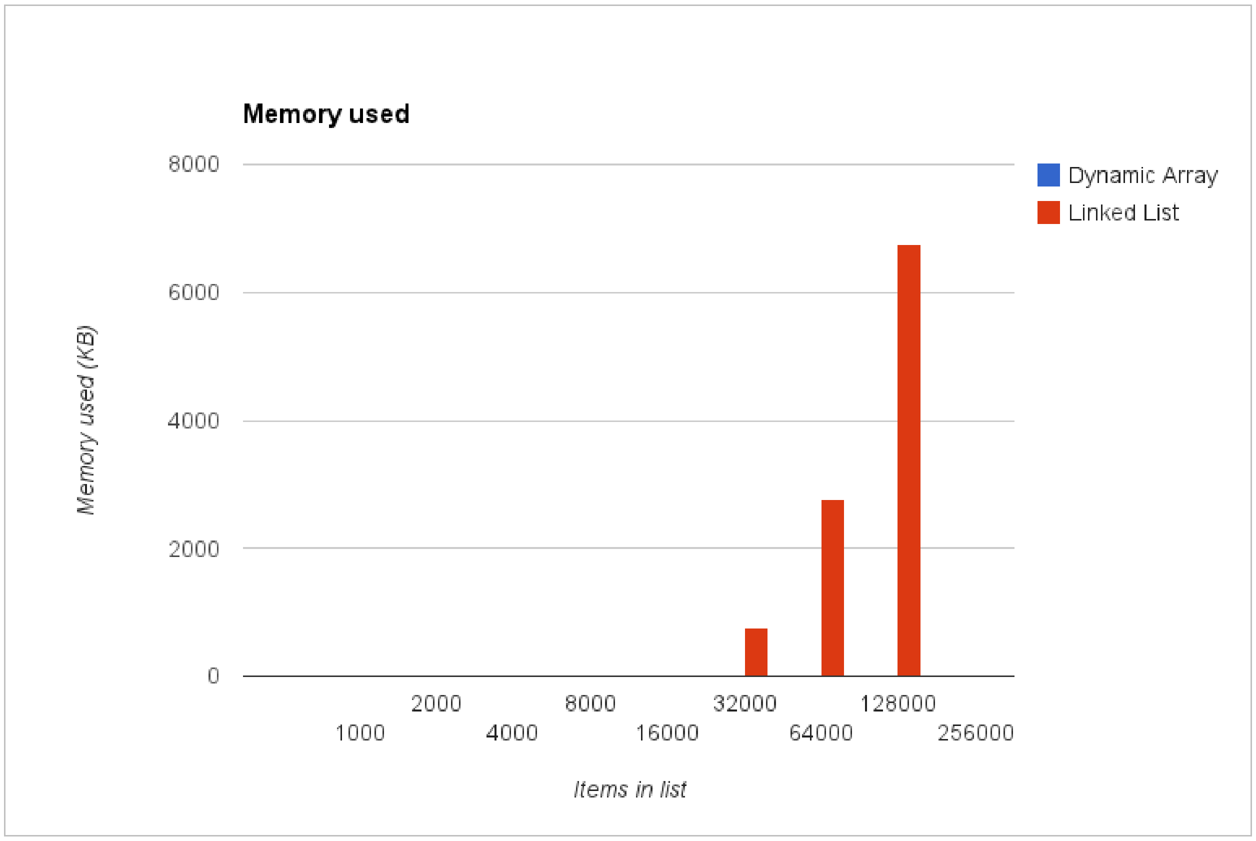
Assignment 3, Part 2 -- Schreiber, Matt

Items in list		Time (ms)	Memory used (KB)		Time (ms)	Memory used (KB)
		Dynamic Array	Dynamic Array		Linked List	Linked List
1000		0	0		0	0
2000		10	0		10	0
4000		30	0		30	0
8000		130	0		120	0
16000		550	0		510	0
32000		2220	0		2050	0
64000		8890	0		8240	764
128000		35580	0		33020	2764
256000		141750	0		214780	6764





1. **Which of the implementations uses more memory, and why?** The linked list uses more memory because the struct representing each element of the contains a pointer to the previous link and to the next link in addition to the variable for storing the link's value. By contrast, each element in the dynamic array takes only as much space as is needed to hold the element's value.
2. **Which of the implementations is the fastest, and why?** The dynamic array is the fastest, due at least in part to the reasons I mentioned in my posts in the Piazza thread entitled "Confused about results from Assignment 3 Part 2", which I copy here:

"To expand on what Dr. Ehsan wrote - in a dynamically-allocated array, all elements are contiguous in memory. In other words, the block of memory occupied by the element at index 0 is directly "next" to the block occupied by the element at index 1, the block occupied by the element at index 1 is directly "next" to the block occupied by the element at index 2, and so on. Traversing the array is therefore as easy as doing pointer arithmetic, which is to say adding a constant value (which depends on the size of the data type stored in the array) to the memory address in question. This is an operation with extremely little overhead. By contrast, traversing a linked list means reading into memory the value of "next" from each DLink, then moving to what could be a very distant location in memory to access the next link, then repeating this operation over and over again.

So, traversing a dynamically-allocated array is less costly in two ways (1) the computational intensiveness of determining where the next element is, and (2) the hardware latency associated with accessing the next memory location…

I don't know very much about memory in general, or about how C/C++ deal with memory in particular, but I think the answer lies in Dr. Ehsan's comments and something that Bjarne Stroustrup says in the video that Cameron posted. It appears that "spatial data caching" may operate in such a way that access to relatively continguous blocks of memory is faster than access to discontiguous elements. The CPU cache has much lower access latency than (most? all?) other forms of volatile memory, and is certainly much faster than disk access. CPU caches are optimized for various sort of locality of reference -- including spatial locality, which basically means that when the contents of a particular piece of memory is read into the cache, the contents of areas of memory "nearby" get read into the cache as well. Here we get to Stroustups's mention of "cache misses": with linked lists, there is an increased likelihood that the next element in the list has not been read into the cache, meaning first that when the CPU looks for the next link in the cache, it won't find it (that's the cache miss), and then the CPU has to access the pertinent memory location. Both the data read miss and the memory access take time, so, given that they are quite a lot more likely to happen when dealing with linked lists, traversing the linked list is much more costly."

3. **Would I expect anything to change if the loop performed remove() instead of contains()? If so, what?** Yes -- the linked list would perform better than the dynamic array, because removing a link involves operations which are all O(1) complexity (and is therefore itself O(1) complexity), namely changing the values of certain pointers and calling free() on the removed link, whereas removing a value from a dynamic array is complexity O(n) since it involves copying an average of n/2 elements to overwrite the removed value.