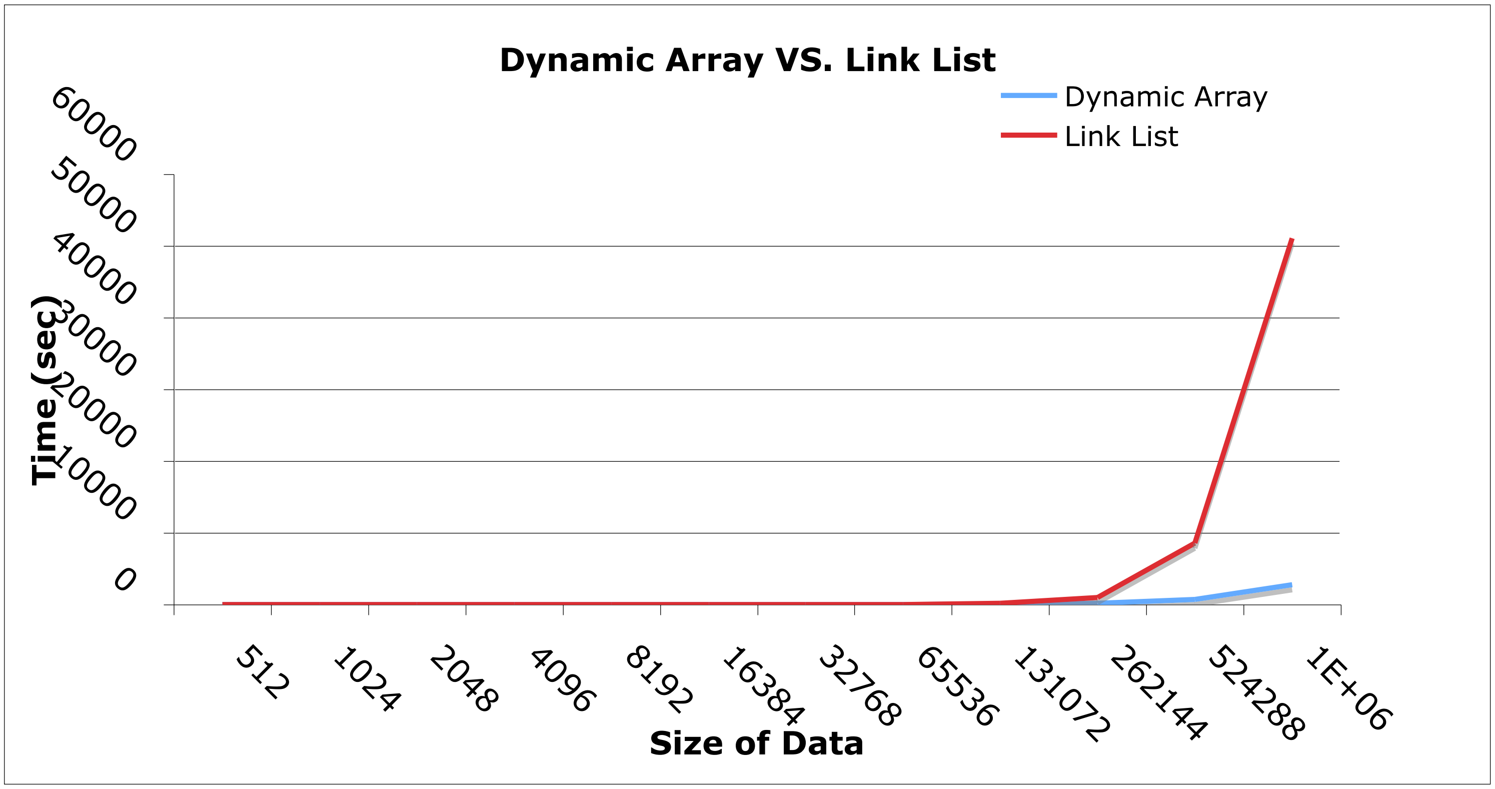
[redacted]



SIZE	**DYNAMIC ARRAY(secs)**	**LINK LIST (secs)**
512	0.00066	0.000507
1024	0.002686	0.002053
2048	0.0104	0.007961
4096	0.055505	0.041994
8192	0.190608	0.186145
16384	0.688898	0.896485
32768	2.821235	6.230206
65536	11.083738	33.361993
131072	44.790205	157.02231
262144	179.784678	997.616536
524288	693.360419	8583.660757
1048576	2786.83995	51096.48447

Link lists are great at inserting and deleting unlike arrays that have to shift all other elements over to fit into a particular location on the array. For the case where you are doing a lot of insertions and deletions, however, the link list seems to become increasingly more inefficient compared to the array as the size of elements gets larger. Why is this? As Bjarne Stroustrup puts it, "Compactness matters and predictable usage patterns matter enourmously." What he means here is that in an array memory is allocated contiguously unlike a link list where each linked node is scattered across random access memory. When the processor needs to read from or write to a location in main memory, it will first check if a copy of that data is in a cache. If the processor does find the data in the cache then it can immediately read from it. If it can't find that data there then it will proceed to look for the data in random access memory. This is called a cache miss and means the cache will need to allocate a new entry and copy in the new data before fulfilling the request for the contents of the cache from the processor. This is clearly a much slower process. With Link lists you maximize your cach misses because the memory of a list is not contiguous. A cache can only hold a relatively small amount of data compared to memory. So when a link list stores its data across the space of memory the cache has no way of collecting a big enough chuck of memory that would hold the entire link list forcing the processor to retrieve data from memory everytime it needed to retreive the next node from another location in memory.