Mark Eatough
CS 2420 – 003

Analysis and Measurement Reflection Paper

When we first started working with linked lists, it seemed like the only difference between an array and a linked list was that a lot of operations were easier to code with a linked list. So I thought why would we ever use an array? From reading in the book, I learned that the array is more efficient in terms of space and time complexity, while the linked lists obviously offer much more flexibility.

Arrays take less space, but we can take more space than is necessary using an array that is too large, or if the array is too small our program cannot run to a successful completion. If we are sure how large our list needs to be an array will be more efficient, and may be the better choice. Linked lists on the other hand take more space but they are much more flexible. We do not need to know how many elements will be needed as the list capacity will adjust as elements are added or subtracted. If we are unsure of how many elements will be in the list, or if the number of elements will fluctuate a great deal, then the linked list will probably be the better option.

Getting an element in an array is more efficient as arrays are random access data structures, so we can move to any element in the array by calculating the address of the target cell, then doing a single memory access. The cost of inserting or removing an element from an array is O(n) if inserted or added at the beginning or middle of the list, as all of the elements will have to be shifted to make room for, or fill the gap of an element. This gives us a best case and worst case scenario for n. Inserting or deleting at the end of an array is O(1) because we simply need to increment or decrement the tail and insert or remove the element. By contrast a linked list is a sequential access data structure, so to get to a particular position we need to move from a starting position (typically the head), to the target position traversing one element at a time. Inserting at the head or tail of a linked list is O(1), as there is no traversing or shifting of the list necessary. We just allocate a new node, set its link and data fields, and thin insert at the head or the tail. There is never a need to shift any elements of a linked list as they are not adjacent in memory. Inserting in the interior of a list requires traversing which has a cost of O(n). The linked list iterator can traverse both directions, and start at the beginning or end of the list, so we never have to traverse more than n/2 links.

Any operation requiring a traversal will have an O(n) cost, but the real cost will be higher for a linked list as compared to an array because of the additional memory access. The traversal run time analysis results out of the tables in the book show that the time needed for the linked list implementation is consistently twice as much than that of an array implementation. So while both of the traversal operations are O(n), the sequential access implementation is much slower. So even though the worst case scenario for traversing is O(n/2) for a linked list, and O(n) for an array list, the time complexity of a linked list will be about the same as an array list at best.

In conclusion if the memory available for a program is limited, we have a good idea of how large the list will be then an array is by far the better choice. If memory is not as much of an issue and we do not know how large a list will be than a linked list will be the better choice. Other considerations would be if elements will have to be inserted at the beginning of the list frequently, as the linked list can do this function more efficiently. Inserting frequently or at the end of the list will have a similar time complexity for both an array and a linked list, but as discussed earlier in reality the array will be more efficient.