Paul English

CS 2420-003

Robert Baird

March 6, 2013

Reflection - Efficiency Vs. Flexibility

When considering the choice of using a LinkedList, or an ArrayList it's important to consider the general trade-offs between flexibility and efficiency. In some ways and ArrayList can be flexible, though more often then not a LinkedList is more flexible. Both List implementations carry various differing efficiencies in both time and space.

1 Space Considerations

Since and ArrayList must always pre-instantiate a private array it's space complexity is always going to be at the capacity of the ArrayList, or $\theta(m)$, where m is the capacity of the list. This means that we must consider the size of the underlying array in implementation, and pay attention to operations that could resize the underlying array changing the space requirement for the list. We have to make sure we allocate arrays of adequate length, so we have the necessary capacity, however it's also possible to allocate an array that leaves us with excess unneeded space.

A LinkedList differs in that it doesn't have a constant space complexity based on the capacity. The space required for any linked list will be based on the number of nodes, or current size of the list at any point in time. It will also be affected by how the nodes keep track of successor and predecessor references, giving it a space complexity of $\theta(n * 2 * size\ of\ reference)$. This difference may be important in certain applications, where the element being stored by the list is very small, since the references to other nodes may be significant. This is less noticeable if you're storing very large elements in your list. Since the space capacity is dependent on the number of elements, or nodes, in the list a very long list may also consume more space than a similar ArrayList.

2 Time Complexity

An ArrayList is very fast for read operations, and appending to the end of the list. However any operations that modify the beginning or interior of the list will incur traversal costs, and list modification costs (having to move elements in the array). This means that modification at the front of an ArrayList is always $\theta(n)$ and modification elsewhere in the list is $O(n)$. Luckily all the other operations on a list are $O(1)$.

A LinkedList is slower for read operations since there is a cost incurred when following node references. However it's efficient when modifying the list since you don't need to move any elements around in memory, you're simply updating element references inside of nodes. Unfortunately any operations that occur inside the list (not at the head or tail) you incur traversal costs, which is $O(n)$ for both ArrayLists and LinkedLists. This is because a LinkedList is made up of nodes that may not be in order in memory, like an Array is.

3 Conclusion

By understanding the space and time efficiencies for our respective lists we can evaluate how their flexibility can relate to efficiency. In general, a LinkedList is more flexible than an ArrayList implementation, since it's easy to point references to other nodes, and it a bit trickier to manage the position of elements in an array. This leaves us with the $O(n)$ traversal cost for most of the operations in a LinkedList however harming our efficiency. When we look at the ArrayList we don't have to traverse the array to find an element, native arrays are randomly accessible since they are one block in memory. They aren't as flexible in general, due to the difficulties involved in add/remove, and managing the capacity.