CSCI 1933 Complexity Analysis

Fletcher Gornick

November 9, 2020

For the `ArrayList` complexities, i'm assuming that the array doesn't need to be resized.

I also have a variable in the LinkedList class that keeps track of the last node making the add function `O(1)`.

`boolean add(T element)` LinkedList - `O(1)` ArrayList - `O(1)`

For LinkedList, all the function does is change the last nodes pointer to a new field and updates isSorted. For ArrayList, the function adds into the lastOpen index and increments lastOpen and updates isSorted.

`boolean add(int index, T element)` LinkedList - `O(n)` ArrayList - `O(n)`

For LinkedList, the function needs to iterate through each node until it finds the correct index, then it moves around the pointers to place the object in correctly. For ArrayList, the function can go directly to the index and change it, but then it has to move everything after it over one position.

`void clear()` LinkedList - `O(1)` ArrayList - `O(1)`

For LinkedList, all you need to do is set the starting node's pointer to null. For ArrayList, you can just update the number of elements to zero so it overwrites the elements already in the list. In my program for ArrayList though, the function is `O(n)` because I update each value to null in the array.

`T get(int index)` LinkedList - `O(n)` ArrayList - `O(1)`

For LinkedList, the list has no concept of indexes, so I created a temporary variable to keep track of indexes, then returned the element's position when the indexes matched. For ArrayList, I was just able to return the index.

`int indexOf(T element)` LinkedList - `O(n)` ArrayList - `O(n)`

For both the LinkedList and ArrayList functions, it needed to iterate through and check if elements were equal, then just go onto the next so they're both `O(n)`. Technically, if the ArrayList was sorted, the complexity could have been reduced to `O(log n)` using a binary search tree, but I was lazy.

`boolean isEmpty()` LinkedList - `O(1)` ArrayList - `O(1)`

Since both Lists will always add to the earliest empty spots, all the functions need to do is check if the first element contains a value, making them both `O(1)`.

`int size()` LinkedList - `O(n)` ArrayList - `O(1)`

For LinkedList, I don't actually have an index position of the last node, so I actually need to iterate through the whole list and keep incrementing until I reach the last node then return that. For ArrayList, all I need to do is return the index of the first open spot on the array because that corresponds to it's size.

`void sort()` LinkedList - $O(n^2)$ ArrayList - $O(n^2)$

For LinkedList, I first move the smallest element to the front by iterating through with a while loop. Once that runs, I have another while loop that checks if it's sorted. If it's not sorted it'll go into another while loop that acts like an insertion sort. Once the sorted condition is met, the double while loop breaks and the function is done. Since there's a double while loop, the function is $O(n^2)$. For ArrayList, the function is just a regular insertion sort which is known to be $O(n^2)$.

`T remove (int index)` LinkedList - $O(n)$ ArrayList - $O(n)$

For LinkedList, the function needs to iterate through until the index is correct, then it changes the pointers to stop pointing to the index of removal. For ArrayList, getting to the index is $O(1)$, but then it's $O(n)$ to move all the following indexes back one.

`void greaterThan(T element)` LinkedList - $O(n)$ ArrayList - $O(n)$

For LinkedList, the function just changes the pointers around an element if it's less than or equal to, then goes onto the next until making it to the end of the list. For ArrayList, the function goes through and alters the list once it reaches an element that's less than or equal to. Worst case for an unsorted array of size k with every element less than is pretty bad though, because it would need to enter a for loop for every element, but it's still technically $O(n)$.

`void lessThan(T element)` LinkedList - $O(n)$ ArrayList - $O(n)$

Same reasoning for the greaterThan function but implemented slightly different if the list is sorted.

`void equalTo(T element)` LinkedList - $O(n)$ ArrayList - $O(n^2)$

For LinkedList, it updates the pointers every time an element is not equal to, or it just moves on if they are equal. The ArrayList function iterates through the list and runs the remove() function if they're not equal which technically makes it a double for loop, so I probably could have done something different but it just kinda felt right.

`String toString()` LinkedList - $O(n)$ ArrayList - $O(n)$

For both methods, the function iterates through the list and prints the value which is a pretty obvious $O(n)$.