180012798 CS2001 07/11/18

W08 – Lists

Overview

- I completed all the Iterative Implementations of the List manipulator class
    - I provided time and space complexities as well.
- Extensions: I completed some of the recursive implementations (up to DeepCopy)

Design

Iterative Methods

ContainsDuplicates

When designing this method I wanted to minimise duplicated comparisons so I only compared each element to the elements after it in the list. This reduced the worst case number of comparisons from n^2 to n(n-2)/2. However this means that for very large lists the difference will be minimal.

ContainsCycles

The simplest solution was Floyd's algorithm. I created two nodes, one traversed the list twice as fast as the other. If the faster node ever meets the slower node then that must mean the list contains a cycle because there is no other way for the slower node to 'catch-up'.

IsCircular

This algorithm is the same as Floyd's algorithm but it has the additional constraint that the two nodes must meet at the head. This is works because if a list is circular then the slower node will always converge with the faster node after one cycle (two cycles for the faster node). If the two nodes meet anywhere other than the head this means there is a cycle but the list isn't circular.

Sort

While the Worst Time Complexity of bubble sort will always be $O(n^2)$, by checking if any elements are swapped before iterating through the list, the number of iterations for lists which are not in the worst ordering is reduced.

Implementation

Iterative Methods

All the iterative methods have constant space complexity as they often only store a set number of nodes and other variables, regardless of list size.

Size

Worst Time Complexity: O(n) Linear – The list has to be be iterated through once.

Worst Size Complexity: O(1) Constant.

Contains

Worst Time Complexity: O(n) Linear – As the linked list is unordered each element has to be examined in turn. Worst case occurs if the searched for element isn't in, or at the end of the list.

Worst Size Complexity: O(1) Constant.

Count

Worst Time Complexity: O(n) Linear – Every element in the list has to be examined to check if it matches the provided element. That means regardless of element being searched for the entire list has to be examined.

Worst Size Complexity: O(3) Constant.

ConvertToString

Worst Time Complexity: O(n) Linear – Linked list has to be iterated through once in order to get strings of each object.

Worst Size Complexity: O(2) Constant.

GetFromFront

To check if an index is outside the current ListNode, I could have used the size method to check at the beginning, however this would give a linear time even in in cases where the index is zero, because the size method has linear time. In my implementation, the InvalidIndexException is only thrown if the end of the list reached.

Worst Time Complexity: O(n) Linear – If index is more longer than the length of the ListNode or at the end then every element of the ListNode has to be iterated through before an exception is thrown.

Worst Size Complexity: O(1) Constant.

GetFromBack

I chose to use the position from the back to calculate the equivalent position from the front and use the GetFromFront method.

Worst Time Complexity: O(n) Linear – the size method is always called so the minimum time is linear. GetFromFront also has a linear time complexity, so at worst the entire list has to be iterated through twice.

Worst Size Complexity: O(1) Constant

DeepEquals

Worst Time Complexity: O(n) Linear – Each list is iterated through one element at a time, at the same time.

Worst Size Complexity: O(1) Constant

DeepCopy

Worst Time Complexity: O(n) Linear – The list is iterated through once in order to copy each element.

Worst Size Complexity: O(1) Constant

ContainsDuplicates

To minimise comparisons while iterating through the list I only compared an element to the elements that came after it. This leads to n(n-1)/2 comparisons being made instead of n^n comparisons.

Worst Time Complexity: $O(n^d)$ Quadratic – at most there can be n(n-1)/2 comparisons being made by the method. This simplifies down to n^2. This occurs if there isn't a duplicate or if only the last 2 elements in the linked list are duplicates.

Worst Size Complexity: O(1) Constant

Append

A node iterates through the linked list until it reaches the tail. It then points the tail to the head of the other list.

Worst Time Complexity: O(n) Linear – The entire linked list has to be iterated through in order to append one list onto the other.

Worst Size Complexity: O(1) Constant

Flatten

When implementing this method I initially called the append method in order to append each inner linked list to the new list. However this would mean iterating through each of the inner list each time a new list is appended, so instead I chose to store the tail node of the new linked list each time the new list was appended.

Worst Time Complexity: O(n) Linear – By storing the tail of the new list, each list added to the new list is only iterated through once.

Worst Size Complexity: O(n) Constant.

IsCircular

Floyd’s algorithm. One node traverses a list two nodes at a time, another node traverses one node at a time. If the two nodes meet at the head then the list is circular.

Worst Time Complexity: O(n) Linear.

Worst Size Complexity: O(1)

ContainsCycles

Worst Time Complexity: O(n) Linear.

Worst Size Complexity: O(1) Constant

Sort

Worst Time Complexity: O(n^2) Quadratic – Bubble sort has a worst case complexity of n^2 if the list is in the exact opposite order it is meant to be in (e.g. ascending instead of descending)

Worst Size Complexity: O(1) Constant.

Map

Worst Time Complexity: O(n) Linear – Each variable is iterated through once.

Worst Size Complexity: O(1) Constant

Reduce:

Worst Time Complexity: O(n) Linear – each node is iterated through once in order to add to the sum variable.

Worst Size Complexity: O(1) Constant

Recursive Implementations

Size

Worst Time Complexity: O(n) Linear

Worst Size Complexity: O(n) Linear

Contains

Worst Time Complexity: O(n) Linear

Worst Size Complexity: O(n) Linear

Count

Worst Time Complexity: O(n) Linear

Worst Size Complexity: O(n) Linear

ConvertToString

Worst Time Complexity: O(n) Linear

Worst Size Complexity: O(n) Linear

GetFromFront

Worst Time Complexity: O(n) Linear

Worst Size Complexity: O(n) Linear

GetFromBack

Worst Time Complexity: O(n) Linear

Worst Size Complexity: O(n) Linear

DeepEquals

Worst Time Complexity: O(n) Linear

Worst Size Complexity: O(n) Linear

DeepCopy

Worst Time Complexity: O(n) Linear

Worst Size Complexity: O(n) Linear

Testing

DeepEquals

I added test as to whether the function still worked if the elements were strings or if the elements were null.

ContainsDuplicates

I checked whether the function would be able to compare elements with null values.

Difficulties

For many of my methods I didn’t use tried not to use the head node, this was because I assumed that if I made another variable (e.g. currentNode) equal to it and then changed the node that currentNode was referencing, the reference would also change in head node. I now know that is not that case however.

Going through the JUnit tests, I couldn’t find many gaps or oversights in the testing. Each of the tests seemed to cover cases where one or both of the inputs where null, they covered a range of normal cases and checked that exceptions were called when they were meant to be.

Bibliography

https://codingfreak.blogspot.com/2012/09/detecting-loop-in-singly-linked-list_22.html – I used this website while researching an algorithm to detect cycles iteratively.