Arrays and Singularly Linked Lists, Advantages and Disadvantages

Jade Koruna

CSCI317: Data Structures and Algorithms

Prof. Jamshidi

May 2nd, 2023

Abstract

Arrays are a common data structure that has several limitations that can make them difficult to use with certain types of algorithms. Singularly linked lists, due to their dynamic size and use of links, can provide a data structure different from arrays that address the limitations of them. An algorithm that finds the lowest value between two arrays and linked lists and appends them to the start of the second array or list demonstrates these advantages, being a more condensed code, is easier to use, and has a slightly better time complexity.

Introduction

An array is a data structure that is introduced early to most beginner programmers, as it is an easy to understand and utilize structure that allows storage of multiple different values of a data type under one structure. However, the array is not without its limitations, and these can often be difficult to overcome for use in certain algorithms. A singularly linked list is a data structure that is similar in use to an array on the surface, however it contains numerous differences that make it advantageous to use compared to an array in certain situations.

A singularly linked list is a data structure built off objects called nodes. Nodes are made up of two different components; the first component is the value stored within the node, and the second is a pointer to the next node. The pointer component is what connects nodes to different nodes. A singularly linked list is a string of multiple nodes, with each node pointing to the next node in the list, except for the final node as it points to null. An advantage to a singularly linked list over an array is that it does not have a fixed size. This makes appending new values to a singularly linked list a significantly less taxing process than appending a new value to an array. For an array, adding a new value to an array requires shifting most if not all the array to make room for a new value. For a singularly linked list, all one must do is change the pointer of one node to point to the next node, and the pointer of the new node to the next node and does not require shifting. A disadvantage of not having a limited size is that linked lists lack indexes, making it more difficult to go to a specific part of the list. The scripts used to set up the nodes and singularly linked list are those created in the references; however the singularly linked list class is slightly edited to have a function to return the starting node (Jamshidi, S).

Algorithm and Algorithmic Comparison

To demonstrate this advantage of a linked list over an array, I have created an algorithm with two different variations, one for arrays and one for linked lists. The algorithm takes in either two integer arrays or two integers linked lists and iterates through the first of the two given to find the lowest value. Once the lowest value of the first is found, it then compares the lowest value to the values of the second list. If the lowest value of the first list is lower than every value in the second list, the lowest value is appended to the beginning of the second list. Note that the code sent with the assignment contains extra code to print out the arrays and numbers, however because this code is not necessary to run the algorithm, that code will not be considered when running through the algorithm.

As an example, for the array version, say you have two arrays of equal length, with the first being [5, 9, 4, 10, 12] and the second being [6, 9, 10, 15, 8]. First, the algorithm iterates through the first array to find the lowest value, which in this case would be 4. The algorithm then iterates through the second array to see if the lowest value is lower than everything in the second array. In this example, 4 is lower than every value in the second array. As 4 is lower than everything in the second array, the algorithm iterates through the second array again, starting from the end. Every value in the second array is shifted to the right, overwriting the final value, to make the first entry in the array. The first entry in the array is then set to the lowest value, which would be 4.

While the beginning of the linked list algorithm is the exact same as the array algorithm, there is one key difference right at the beginning. If a linked list has a size of 0, the algorithm is unable to run, so if one list does have a size of 0 the algorithm automatically ends. Say we have two lists with the values (5, 9, 4, 10, 12) and (6, 9, 10, 15, 8), much like the first example. The algorithm iterates through the first list, finds the lowest value of 4, then iterates through the second list and finds that 4 is lower than every value in the second list. Instead of having to iterate through the second list again to shift everything and add the new value, a new node is created, that node's pointer is set to the original first node, and the new node is set as the starting node of the linked list. Cutting out a third iteration is possible due to linked list's having a dynamic size and easy addition of new values. These changes result in a few advantages for the linked list version of the algorithm. For one, having to immediately end the algorithm when a usable list is given results in a better best case time complexity for the linked list algorithm compared to the array one. Losing the third loop iteration condenses the code and makes it more digestible instead of a long string of for loops. It also allows for easier to use code, as just adding a node to the beginning of a list is much less complicated than shifting an entire array. The linked list algorithm, due to its advantages, ends up more condensed, easier to use, and has a slightly better time complexity.

Time Complexity

Due to the conditional check in the linked list example, the exact time complexities for each algorithm come out a bit differently. The best case for the array algorithm is if the first number in the first array is the lowest number, and if that number is greater than every value in the second array, which immediately ends the algorithm. Because there is a full for loop iteration through the first array no matter the case, and that for loop is based on the length of the first array, the best-case time complexity will always be linear, being Omega(n). For the linked list algorithm, because of the initial conditional check, if one of the lists has a size of 0, the algorithm immediately ends, making the best case scenario Omega(1). The worst case for both algorithms follow a similar pattern; the worst-case scenario is if the first array or list starts with the greatest number and each subsequent number is lower than it. Then the lowest number from the first list is lower than every value in the second list, thus being appended to the second list. Thus, the worst case runs the greatest number of commands in all for loops the maximum number of times. Because none of the for loops are nested in each other, the worst case for both algorithms always run as a factor of length, thus making the worst-case time complexity for both O(n). Despite the loss of the third loop in the linked list algorithm, the need for an extra command in the first two for loops to navigate through the linked list causes the worst-case time complexity to remain more or less identical between the two. The average time complexity for the array algorithm is Theta(n), as both worst case and best case are also linear time. The average time complexity for linked lists is also Theta(n); while best case time complexity is constant time, there's only three cases where it is best case, while there's an infinite number of cases where the size of both lists is above 0, which means at least one entire for loop will always run.

Conclusion

Linked lists contain numerous advantages over arrays, which shows within the created algorithm to find the two lowest numbers between either two arrays or lists and appending them to the beginning of one of the given lists. The dynamic size of linked lists allows for new values to be appended to the list without having to shift every value in the list, making it much easier to use and malleable. Linked lists also allow for slightly better best case time complexity being able to immediately end when a linked list is empty, while empty arrays will still run with the algorithm despite being useless. While the lack of indexes for lists causes loops iterating through a list to have a higher time complexity than arrays, the numerous other advantages outweigh this, resulting in an easier to use, better algorithm.

References

Jamshidi, S (2023), Singularly Linked Lists (Version 1.0)

Jamshidi, S (2023), Node (Version 1.0)