DATA 311 Assignment 3 (Bonus Written Part)

Yunfan Yang 30067857

Searching on a linked list (10 points)

What is the biggest drawback in a linked list (SLL or DLL) that makes O(logn) search impossible on it?

For a search algorithm with O(logn) such as binary search, it usually divides the given array in half to subarrays and repeat this process until the value is found; by achieving this, the algorithm has to have the ability to find the middle point of the array repeatedly. Since linked lists always only track the head or both head and tail, and there is no direct way to access the middle point of the list. Therefore, the O(logn) search algorithms are impossible on a linked list.

A candidate modification to linked lists (10 points)

Consider a doubly linked list. We know that every node in a DLL contains a pointer to the next and previous nodes respectively. Now suppose every node has a next pointer to every other node that appears on the list after it, and a previous pointer to every other node that appears on the list prior to it. Suppose this list has n such nodes.

1. WHAT IS THE OVERALL OVERHEAD IN TERMS OF POINTER STORAGE PER NODE AND ON THE LIST AS A WHOLE?

   For each node on the list will now have $n-1$ pointers. For the whole list, the number of pointers will be $n(n-1) = n^2 - n$, the memory will have an increment for each new Node add to the linked list at a rate of 0.5 compared to the original linked list.

2. WHAT IS THE AVERAGE ASYMPTOTIC COST FOR INSERTING A NODE AT AN ARBITRARY LOCATION IN THIS LIST?

   The average asymptotic cost for inserting now will be O(1) since each node has the pointers of any other nodes, which the node can be directly inserted into an arbitrary location instead of tracing every node O(n).

3. DESCRIBE HOW YOU MIGHT PERFORM BINARY SEARCH ON THIS MODIFIED LIST.

   My implementation in the code file provided tries to be the same as possible with the original binary search (although there are other easier ways, I believe, to implement). The index in the algorithm will be the same as the original list; there will be a *importFromArray* function to import an original list to a linked list for testing, and the order of all the elements keeps the same. The search will be started from the middle point of the linked list (the middle point can be found by the element number of the successor list of the linked list head, and minus one since it needs to exclude itself when

dealing with the index, and floor divided by 2, and minus one because of the same reason), and it will pass the middle point Node instance to start the searching.

The followings are the steps:

a. The base case of the modified binary search is still “when the middle point value is the goal value”
b. Also if the Node’s value is None, or the current Node index is invalid (which is less than 0), or the searching range is invalid (which is less than 0), it means the goal value is not in the list.
c. If the goal value is less than the middle value, then divide the array into half by limiting the search range and calculate the new middle point in the search range from the current Node’s predecessor. Then it will pass the middle point Node instance to keep finding.
d. If the goal value is greater than the middle value, then divide the array and set the middle point as on the above step, but the middle point is from the current Node’s successor.
e. After dividing, it will recursively invoke *_binarysearch* function until the base case is satisfied.

The *index* variable has no direct effect on the algorithm itself but just to keep track of the current Node’s index in the list, we need this variable since Node instance does not have an index attribute.