Searching on a linked list (10 points) :

For the singly linked list, we cannot traverse in the reverse direction and so using a doubly-linked list would make more sense since a pointer to the tail is definitely kept and the previous pointer of each node is kept as well. Linked Lists are inherently sequential access, therefore we have to traverse the list for accessing a Node which takes longer. Since any random element can not be accessed in constant time and assuming the list is sorted accessing an element in Linked-List is O(n). We cannot apply search methods like Binary Search on Linked-List because there is no random access even if we have a pointer to the middle element, by getting the middle element each iteration would mean you have to traverse the linked list till you reach the middle element. T = O(n/2) + O(1) the complexity of comparing being O(1) and the complexity of getting middle element being O(n/2) means that it is dependent on the size of the list. This is an order of O(n).

A candidate modification to linked lists (10 points):

1. Node has previous and next pointer to each node in the list.
   Number of Node in list = n
   Number of next pointer in Node = n - 1
   Number of previous pointers in Node = n - 1
   Storage per node = $(n-1)\,pointers$ **, Memory complexity = O(n)**
   Storage for whole list = $(n)nodes * n-1$ **, Memory complexity = O(n^2)**
2. Inserting a node at an arbitrary location means that the node that is inserted needs to point to every other node, the previous pointer pointing to those nodes before it and next pointers pointing to those nodes after it. Every other node also needs a pointer to be added pointing to the new node. So for space complexity, we have (2(n-1)) new pointers added to the list which is **Memory complexity of O(n).** For time complexity it would be $T = n^2 + n \;\; which\; is\; in\; order\; of\; O(n^2)$. As the new node needs n amount of pointer for its self, taking n-1 steps = $O(n)$, And every other node needs to point to the new node which takes n steps, but as well as move every other pointer to other nodes accordingly to point to the correct node, which is another n steps, thus taking n^2 steps.
3. Assuming the list is sorted and the size of the list is being tracked, we can move from either side of the list to the middle by $mid = size\,//2$ by using the next or previous pointer and mid calculations to get to the middle node. Then compare the key to the node value from the middle. If less then look at the first half of the list else look at the middle node to the last node. Recursively or iteratively repeat until one node is left over. If the node value is equal to the key return the node.
   Also since this is a multilinked list it can be treated as a skiplist and easily with the same steps jump to the certain nodes of the list base on interval pointers that are already set up.