EECS2101 - Assignment 1

Daniel Chahine - 219598994

January 26, 2024

	Fast	**Slow**
(a) Search for an element.	—	A B C D E F G
(b) Get the middle element.	A	B C D E F G
(c) Insert at the beginning of the list.	B C E F G	A D
(d) Insert at the end of the list.	C D E F G	A B
(e) Delete from the beginning of the list.	B C E F G	A D
(f) Delete from the end of the list.	A D E G	B C F

Table 1: Efficiency of different data structure with different operations

The above table uses the following definitions for the letters A, B, C, D, E, F, and G:
A. array (like the Java array)
B. singly-linked list, with “next” only, and with “head” only.
C. singly-linked list, with “next” only, and with “head” and “tail”.
D. reverse singly-linked list, with “prev” only, and with “tail” only.
E. doubly-linked list with “next” and “prev”, and with “head” and “tail”.
F. circular singly-linked list, with “next” only, and with “tail” only.
G. circular doubly-linked list, with “next” and “prev”, and with “tail” only.

1 Search for an element

1.1 Array - Slow

With the assumption that the array is made of random elements and not ordered, looking for an element in an array will be slow. We have to go through every element in the worst-case scenario when the element is the last one or not presented in the array. This process of traversing the array will have a time complexity of O(n). Hence proportional to the size of the array.

1.2 Singly-Linked list, with ”next” only, and with ”head” only - Slow

With an assumption similar to the array, looking for an item in a singly-linked list is proportional to the array. We start at the head and traverse the list until we reach the end (the node is null) or until we find the element. This process also has a time complexity of O(n) meaning it will be proportional to the size of the linked list.

1.3 Singly-linked list, with “next” only, and with “head” and “tail” - Slow

Having a reference to the tail will not make the singly-linked list any faster when searching for an element. The assumptions and the process of searching are similar to 1.2, traversing the list while checking for the item and if found, stop traversing. The process is proportional to the size of the list with a time complexity of O(n).

1.4 Reverse singly-linked list, with "prev" only, and with "tail" only - Slow

This is similar to the Singly-linked list with "next" while having reference to the "head" - 1.2, just the other way around. With a similar assumption, we start from the tail, and traverse the list backwards until we find the element or till the beginning of the list (node is null). The time complexity of O(n), therefore the number of steps is proportional to the size of the list.

1.5 Doubly-linked list with "next" and "prev", and with "head" and "tail" - Slow

Being able to traverse back and forth is not going to make the search for an element faster. Similar process to 1.2, 1.3, or 1.4. Start from one end and traverse until we find the element, or until the node is null. Slow because it will be proportional to the size of the list, O(n).

1.6 Circular singly-linked list, with "next" only, and with "tail" only - Slow

Similar process to a singly-linked list with reference to the head. The only difference is that we stop the loop when our node is equal to our starting node (either head or tail), or when we find the element. Since we have to traverse the whole list, it means that the number of steps taken by the search is proportional to the size of the list with O(n).

1.7 Circular doubly-linked list, with "next" and "prev", and with "tail" only - Slow

Similar to 1.6 and with the same assumptions (the list is random and not ordered), we have to traverse the whole list. In this case, we can choose any direction (next or prev) and any starting point (either head or tail). The number of steps taken by the search is also proportional to the size of the list, since in the worst case (if the element is not there) we have to traverse the whole list. Slow with time complexity of O(n).

2 Get the middle element

2.1 Array - Fast

Getting the middle element in an array can be done in a constant number of steps. Because the array data type is stored in a way where all elements are next to each other, indexing (getting an item through its index) an array requires constant steps since we're assuming we have the size of the array. We only have to obtain the element at index: $\lfloor size/2 \rfloor$. No need to traverse the whole array to get this element, since the value of the element is calculated by knowing the address of the first element + (index * size of one element) which gives us the address of the wanted element. The time complexity is O(1). Hence it is fast.

2.2 Singly-Linked list, with "next" only, and with "head only

Getting the middle element in a singly-linked list with a "head" given would require some steps that are proportional to the size of the list. To go to the middle item, we need to get the size of the list first, hence we need to traverse the list all the way to the end while keeping a counter variable. Once we reach the end (when our node is null) we have to start by traversing the list again till we reach a counter value of $\lfloor listSize/2 \rfloor$. This process requires a number of steps proportional to the size of the list with a time complexity of O(n). Hence this operation is slow.

2.3 Singly-linked list, with "next" only, and with "head" and "tail"

Having a head and a tail to a singly-linked won't differ much from having just a head (Process in 2.2). Obtaining the middle item requires us to traverse the whole singly-linked list from the head, get the

size in a counter variable, start from the head again until $\lfloor listSize/2 \rfloor$, and return the element. This operation will take a number of steps proportional to the size of the list. Hence it's slow with O(n).

2.4 Reverse singly-linked list, with "prev" only, and with "tail" only

This is similar to having a linked list with "next" while having a reference to the head (Process in 2.2). We just start from the tail and work our way backwards, while having a size counter, till we reach the beginning. Then we start again with another counter and traverse the list till the counter reaches $\lfloor listSize/2 \rfloor$. The number of steps taken by the operation is proportional to the size of the list. Hence this is a slow operation with a time complexity of O(n).

2.5 Doubly-linked list with "next" and "prev", and with "head" and "tail"

Being able to move back and forth in a linked list won't affect the time that it takes for us to get the middle element. Since we have no sense of size nor middle element in our linked list, we still need to traverse the list from either side while counting all elements, then traverse it again till we reach $\lfloor listSize/2 \rfloor$. This is a slow approach as the number of steps taken will be proportional to the size of the list.

2.6 Circular singly-linked list, with "next" only, and with "tail" only.

Having a tail pointing to the beginning of the list won't help much as we still have to traverse the list to find the middle element. Similar assumption and process to 2.2, 2.3, 2.4, and 2.5, we need to start at the tail and count the number of elements. Once we get the number of nodes that are in the list (list Size), we start again and go to $\lfloor listSize/2 \rfloor$. The process is slow as the number of steps is proportional to the size of the list.

2.7 Circular doubly-linked list, with "next" and "prev", and with "tail" only.

Having a circular doubly-linked list with next and prev pointers won't make a big difference from a doubly-linked list in terms of finding the middle element (similar process to 2.5). To find the middle element we need to traverse the list at least once, since we don't have the size of the list, to count how many elements there are. Then we start again and traverse the list until we reach the $\lfloor listSize/2 \rfloor$. The number of steps this process will take will be proportional to the size of the list. Hence this is a slow operation on this data structure.

3 Insert a new element at the beginning of the list

We're assuming that the head node will be similar to any other node, have a value in it and a pointer to the next node.

3.1 Array - slow

When working with arrays, inserting at the beginning could be considered slow. Because an array has a special property of having all of its elements in one chunk of memory, inserting at the beginning requires us to shift all of its elements once to the right (in order for the first element to have an index of 0). Hence the number of steps needed to perform an insertion at the beginning will be proportional the size of the list (slow).

3.2 Singly-Linked list, with "next" only, and with "head only

It's much faster to insert at the beginning of a simple singly-linked list. With the assumption that the head node could hold a value and a pointer to the next element, the element we're inserting will be pointing to the head. So when inserting the element, we just need to our new element point to the previous head and call our element the new head. We can see that the number of steps to perform this operation will be constant, which means that this operation will be fast given this data structure.

3.3 Singly-linked list, with "next" only, and with "head" and "tail"

Having a tail in the singly-linked list won't change the insertion at the beginning. We only need to make our current node point to the head and make the head of the list our node.

3.4 Reverse singly-linked list, with "prev" only, and with "tail" only

Because we only have the tail of the list, adding at the beginning of the list will take more time than other data structures. We need to traverse the whole list until we reach the last node (this could be done until the previous pointer is pointing to null). Just with that alone, we discover that the number of steps for this process will be proportional to the number of steps we have.