EECS2101 - Assignment 1

Daniel Chahine - 219598994

February 2, 2024

Question (1)

	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.	A C D E F G	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.

Some Assumptions:

- All given values in our data structures are made of random elements and they are not ordered.
- When adding an item from an array that implements a list ADT, we assume that the size of the array will always be greater than the size of the list. Meaning there's always more space to add an element. This makes the process easier as we don't have to duplicate all the elements when adding at the end.
- When we're adding an element to a linked-list, we can call the element being added N.
- When we're traversing through a list, we're assuming that for every iteration we're doing node = node.next(or node.prev when traversing a reversed/doubly linked-list).
- The tail and the head are not normal nodes (they don't hold a value) they only have a reference to the next/prev element. To access the last element we can do tail.next (tail.prev in case it is a reverse linked-list). To access the first element, we can do head.next.

1 Search for an element

1.1 Array - Slow

Since we're searching for an element in any data structure we have to visit every element at least once. In the worst case, the element is the last one or not in the array. This process of traversing the array will have a time complexity of O(n) since we have to index every element in our array. Hence proportional to the size of the array.

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

To search for an element we start at the head and traverse the list until we reach the end (the node is null) or until we find the element. After every iteration we move to the next node (node = node.next) 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 allow us to search faster. The process of searching is 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 will not 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). Another approach could be starting from the head and tail at the same time and traversing the list from both sides. Even though this process might seem more efficient, the number of steps taken is still proportional to the size of the list. Hence slow.

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

Similar process to a singly-linked list with a reference to the head. The only difference is that we stop the loop when our node is equal to our starting node (loop until node = tail.next), or when we find the element. Since we still 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, we have to traverse the whole list. In this case, we can choose any direction (next or prev) and start from the 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 time complexity (constant number of steps) with the size of the array being given. We only have to obtain the element at index: $\lfloor ListSize/2 \rfloor$. 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 - Slow

Getting the middle element in a singly-linked list with a ”head” given would require a number of 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 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 many 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” - Slow

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 several 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 - Slow

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” - Slow

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 - Slow

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 - Slow

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

3.1 Array - Slow

When working with arrays, inserting at the beginning is considered slow as inserting at the beginning requires us to shift all of the elements of the array once to the right (for the first element to have an index of 0 - keep the right index for all the elements). Hence the number of steps needed to perform an insertion at the beginning will be proportional to the size of the list (slow).

3.2 Singly-Linked list, with ”next” only, and with ”head only - Fast

It’s much faster to insert at the beginning of a simple singly-linked list. The element we’re inserting will be pointing to the head.next. So when inserting the element, we just need to our new element point to the old first element and call our element the new head (head.next = N). 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” - Fast

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 old first element and make the head of the list our new node (head.next = N). We can see that we need a constant number of steps to perform this operation - Fast.

3.4 Reverse singly-linked list, with “prev” only, and with “tail” only - Slow

Because we only have the tail of the list, adding at the beginning of the list could take more time than other data structures. We need to traverse the whole list until we reach the first 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 size of the list, hence slow. After we reach the first node, we can make it point to our node and make our node point to null. This process will have a time complexity of O(n).

3.5 Doubly-linked list with “next” and “prev”, and with “head” and “tail” - Fast

To insert a new element at the beginning of our doubly-linked list, we should make this element point at the head.next, and make the head.next.prev point to our new element. After that, we can make our new node be the head of the list. As we can see, this operation requires a specific (constant) number of steps which would make it fast.

3.6 Circular singly-linked list, with “next” only, and with “tail” only - Fast

Inserting at the beginning of a circular singly-linked list could be fast. Since this list is circular, tail.next.next should point at the old first element. We make our new element point to the beginning of the list. After that, we make the tail.next point to our new element by doing tail.next.next = N. This means that the last element is pointing to the new first element now, completing our insertion at the beginning. The number of steps taken is constant, hence this is a fast operation.

3.7 Circular doubly-linked list, with “next” and “prev”, and with “tail” only - Fast

To add N at the beginning of our circular doubly-linked list (make it the new beginning), we can start by making N.prev equal to our tail.next, since the node before the first node should point to the tail.next. Then we should make N.next point to whatever the tail.next was pointing at by doing N.next = tail.next.next. Now we should make the node after N point to N as a previous node by doing N.next.prev = N. Finally, we just make our node tail.next point to N as its next node by doing tail.next.next = N. This could be seen as a lot of steps but it is still a constant number of steps, hence this operation is fast when working with a circular doubly-linked list.

4 Insert at the end of the list.

4.1 Array - Fast

As we’re implementing a list ADT from an array, we’re given the size of the array which means we’re assuming there’s free space in the list. We’re also assuming that we won’t add an element to the list when it’s full. With those assumptions, we can just index to the end of the list add the new element (array[size] = new element) and increase the number of items by 1. We can see that it takes a constant number of steps which makes the operation fast.

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

Adding a new element at the of singly-linked would be slow because we need to find the last element. Since we don’t have any sense of the end of the list, we need to traverse the whole list until we reach the last node (until node.next == null). When we have the end, we just make a pointer from the last node, to our new element. Also, we have to make our new element point to null to show that it is the last element. The number of steps taken for this process will be proportional to the size of the list as we have to go through every single element (node) till we reach the end. Hence this operation is slow.

4.3 Singly-linked list, with “next” only, and with “head” and “tail” - Fast

Having a reference to the tail makes our insertion much faster. All we need to do is to make the tail.next.next = N, make N.next = null, and tail.next = N (referring to our new node as the last element). We can see that the number of steps is constant (3 or 4 steps). Hence this operation is fast with O(1).

4.4 Reverse singly-linked list, with “prev” only, and with “tail” only - Fast

Adding an element at the end of a reversed linked list is really similar to adding it at the beginning of a singly-linked list. We start by doing N.prev = tail.next, basically saying that N points to the previous tail. Then we just make our new element the new tail (tail.next = N). This operation takes a constant number of steps, which means it is fast.

4.5 Doubly-linked list with “next” and “prev”, and with “head” and “tail” - Fast

When using a doubly linked-list to add an item at the end, we can use the reference of the tail and make it point to the new element (tail.next.next = N), make our element point to the tail previously (N.prev = tail.next). At the end, just make the new tail our element (tail.next = N). We can see that the number of operations is constant which makes it fast.

4.6 Circular singly-linked list, with “next” only, and with “tail” only - Fast

This process will be similar to adding to 4.3. We start by making our new element point to the beginning of the list since the last element should point to the start in a circular list (N.next = tail.next.next). We make the tail point to our new element (tail.next.next = N). Finally, our new element becomes the new tail (tail.next = N). A constant number of operations were required, hence we have a fast operation on this data structure.

4.7 Circular doubly-linked list, with “next” and “prev”, and with “tail” only - Fast

The process is similar to 4.6, the only difference is we have to keep track of the prev pointer as well. We start by making N point to the beginning since it will be the last item in a circular list (N.next = tail.next.next). We also have to make N point to the old tail as a previous node (N.prev = tail.next). We have to make the beginning point the new node as its previous node (N.next.prev = N). Finally, we just make the old tail point to N (tail.next.next = N) then call N the new tail (tail.next = N).

5 Delete from the beginning of the list

5.1 Array - Slow

Deleting from the beginning of the array will be time-consuming as it will require us to shift all of the other elements, one by one, to the left to replace the empty memory of the array at index 0. This could be done using a for loop with an index starting from 1 and ending at a list length - 1. For each index i, we can do the following operation, array[i-1] = index[i]. The number of steps required by the operation will depend on the size of the list which means that this operation is slow.

5.2 Singly-Linked list, with ”next” only, and with ”head only - Fast

Deleting from the beginning of this data structure could be done just by making setting the new head as the second node, what the old head was pointing to (head.next = head.next.next). Because this operation takes a constant number of steps we can say it’s a fast operation.

5.3 Singly-linked list, with “next” only, and with “head” and “tail” - Fast

Having a reference to the tail won’t help us when we’re removing from the beginning. Similar to 5.2, we just need to make the new head equal to the old head (head.next = head.next.next). A constant number of steps were taken which allows this operation to be fast.

5.4 Reverse singly-linked list, with “prev” only, and with “tail” only - Slow

To remove the first node, we need to know what is the first node. To do that in a reverse linked list, we have to start from the tail, traverse to the second last (until node.prev.prev = null), make it the last node by making it point to null (node.prev = null). Since we had to traverse the whole linked list, we needed to use a number of steps proportional to the size of the list. Hence, it is a slow operation.

5.5 Doubly-linked list with “next” and “prev”, and with “head” and “tail” - Fast

Removing the beginning of this data structure is done by making the second element point to null as a previous node (head.next.next.prev = null). Then make the second element the head (head.next = head.next.next). We can see that we used 2 or 3 steps to this operation which means that this operation is fast using this data structure.

5.6 Circular singly-linked list, with “next” only, and with “tail” only - Fast

Making our tail point to the second element (what the first element is pointing to) is enough to remove the first element in our list (tail.next.next = tail.next.next.next). Constant number of steps, (could be done in 1 or 2 steps) make this operation fast.

5.7 Circular doubly-linked list, with “next” and “prev”, and with “tail” only - Fast

The second element from a circular doubly linked list could be obtained by doing tail.next.next.next. To delete from the beginning of the list, we can just make the tail.next point to the second element as its next element (tail.next.next = tail.next.next.next), and then we just have to make the second element point to the tail.next (tail.next.next.prev = tail.next). A couple of steps are used to do this operation, which makes it a fast operation.

6 Delete from the end of the list

6.1 Array - Fast

To delete from the end of the list, we would just decrease the size of the list by 1 which won’t allow us to access the item at the last index, hence deleting the last element. This works because our implementation checks for the index before accessing the item (if index <= size-1). The number of steps is constant which makes this operation fast.

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

When deleting the last item in a singly linked list with only reference to the head, we need to traverse the whole list until we reach the end of the list. Then we make the second last element point to till signifying that it is the last element. We just loop from the head until node.next.next = null, when we reach that point we just make node.next = null. We can see that the number of steps was proportional to the size of the list as we have to traverse more items when we have a longer list. Hence this operation is slow.

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

It might seem like it’s easier to remove the last item when we have a reference to the head and tail, but since we need a reference for the new tail, we still need to traverse the list from the beginning. To remove the last item we have to loop until we reach the end (node.next = tail.next). Once we’re pointing at the tail, we make the second last node point to null (node.next = null) and make the second last node the new tail (tail.next = node). We can see that the number of steps increases when the size of the list increases, as we have to traverse more items.

6.4 Reverse singly-linked list, with “prev” only, and with “tail” only - Fast

This is similar to 5.2. To remove the last element from the list, we just make the node before the tail our new tail (tail.prev = tail.prev.prev). The number of steps taken is constant, hence this is a fast operation with this data structure.

6.5 Doubly-linked list with “next” and “prev”, and with “head” and “tail” - Fast

To remove the last element, we just go to the second last node and make it the new tail (tail.next = tai.next.prev). We also have to make it point to null as its next node (tail.next.next = null). A constant number of steps are needed which makes this operation a fast operation.

6.6 Circular singly-linked list, with “next” only, and with “tail” only - Slow

To do this operation, the element before the tail should become the new tail. But since we don’t have a reference to the previous element, we have to traverse the whole list until we reach the element before the tail (node.next = tail.next). We just make this node point to the first node (node.next = tail.next.next), and then we just set this node as the new tail (tail.next = node). Since we had to traverse the list, the number of steps required is proportional to the size of the list which makes the operation a slow operation.

6.7 Circular doubly-linked list, with “next” and “prev”, and with “tail” only - Fast

The difference between this data structure and 6.6 is that we have a previous point which allows us to not traverse the list and just access the second last element with tail.next.prev. We make the first element point to the second last element as its previous element (tail.next.next.prev = tail.next.prev), make the second last element point to the first element as its next element (tail.next.prev.next = tail.next.next), then just call the second last element the new tail (tail.next = tail.next.prev). The process takes a constant number of steps and doesn’t rely on the length of the list which means that this process is fast.

Question (2)

Both Queues and Stack are implemented as new classes in the provided java file.
They are both an implementation that uses an array list to store the elements. The provided classes are not generic class, meaning they only work when the data type of the element is an Integer. When pushing an element, we push it to the end of the list no matter the type. The implemented isEmpty methods check if the list implemented is empty. The only difference between the stack and the queue is the pop/peek. We remove/look at the first element when dealing with queue, but remove/look at the last element when working with stacks.

The variables used in this solution are split into 3 parts. The start, the buffer, and the exit.

```java
Queue start = new Queue();
Stack buffer = new Stack();
Queue exit = new Queue();

for (Integer item: arr) {
    start.push(item);
}
```

The use of a Queue for start and exit is logical because we want to process the pucks in a first-in, first-out (FIFO) manner. The order in which the pucks are placed and removed from the start and exit areas should follow a sequential pattern, similar to how elements are processed in a queue.

The use of a Stack for the buffer is appropriate because the rules of the game only allow the pucks to turn left when moving into or out of the buffer line. A stack follows the last-in, first-out (LIFO) principle, which aligns with the left-turning movement allowed by the game. Pucks can be easily added and removed from the top of the stack, ensuring that the last puck placed in the buffer is the first to be moved out.

With the help of a counter that keeps track of what the value of the next element should be, a while loop is used to iterate until the start and buffer are empty.

```java
while(!start.isEmpty() || !buffer.isEmpty()){}
```

Through every iteration, we check if the beginning of the queue matches the counter and if so we move the item to the exit by popping it from the start and pushing it into the exit queue.

```java
if (start.peek()!=null && start.peek()==counter) {
    exit.push(start.pop());
    counter++;
}
```

Only if the start doesn't match the counter, we check if the buffer matches the counter.

```java
else if(buffer.peek()!=null && buffer.peek()==counter) {
    exit.push(buffer.pop());
    counter++;
}
```

If the buffer doesn't match the counter and there's an item in the start we move it to the buffer. Otherwise we just exit the loop because we can't move anything to the exit or the buffer.

```java
else if (start.peek()!=null){
    buffer.push(start.pop());
}else {
    break;
}
```

Once we finish with the loop return the size of the exit queue as it represents how many items that were ordered and exited. Hence completing the solve algorithm

```java
return exit.size();
```