Fundamental of Data Structures and Algorithms
Assignment 1

Do Huynh

January 26, 2024

1 Efficiency Analysis

** For options D, F, and G, assume that the “tail” node is the first element of the List ADT, then the head of the Linked list will be the last element of the List ADT*

1.1 Search for an element

- **Fast group**:
- **Slow group**: A, B, C, D, E, F, G
- **Reasoning**: Even though accessing an element in an Array can be done in constant time because of direct access to elements using indices, you currently don’t have that index number so you have to traverse through the array to find that element. Linked lists (B, C, D, E, F, G) have only one way to search for an element and it’s traversing the list, and the number of steps in the two cases is proportional to the size of the list.

1.2 Get the middle element, i.e., with the list size being n, return the [n/2]-th element in the list

- **Fast group**: A
- **Slow group**: B, C, D, E, F, G
- **Reasoning**: with an Array, you already have access to the “size” property of the list so you can figure out the index of the middle element by doing *size/2* and access that element in constant time $O(1)$. For the other linked list options, since you don’t have access to the “size” attribute, you have to travel through the whole list to figure out how many elements are in the list and then figure out where the middle element is. That makes the operation slow with the time taken to do all that is $O(n)$.

1.3 Insert a new element at the beginning of the list

- **Fast group**: B, C, E, F, G
- **Slow group**: A, D
- **Reasoning**: Even though you have direct access to the first element of the array, you still have to shift everything to the right to make room for the new element. Option D only offers the ”tail” and ”prev” so you have to travel through the list to get to the ”head” and perform the operation. Those options in the ”Fast” group all provide access to the ”head” of the list either by directly providing ”head” or by being circular with ”tail” points to the ”head”.

1.4 Insert a new element at the end of the list

- **Fast group**: A, C, D, E, F, G
- **Slow group**: B
- **Reasoning**: options in the “Fast” group all provide access to the end of the list. An array provides the end of the list which is n where n is the number of elements of the list. Other Linked List options in the group provide the “tail” node and you can make it point to the new element and make the new element the new “tail”. B is slow because you have to travel the whole list from head to tail to perform the operation.

1.5 Delete an element from the beginning of the list

- **Fast group**: B, C, D, E, F, G
- **Slow group**: A, D
- **Reasoning**: An array will be slow because the operation requires shifting all subsequent elements one position to the left, option D requires traveling the whole list to get to the ”head” to perform the operation. The worst-case running time of both of these cases is $O(n)$ while other linked list options take constant time $O(1)$ because they provide access to the ”head” and you only need to update the “head” of the linked lists to the new “next” node or point ”next” of ”tail” to the ”next” of that element if you are using on the circular linked lists.

1.6 Delete an element from the end of the list

- **Fast group**: A, D, E, G
- **Slow group**: B, C, F

- **Reasoning**: with an array, you can access the last element in constant time because you know the index of the last element is “size-1”. For the linked lists options, the key to take here is that you need to have access to the second last element of the list to make it the new tail. E and G are fast because you can make ”tail” points to ”tail”’s ”prev” and this new ”tail”’s ”next” points to null (for option E) or you can update the ”next” of this ”prev” to point to what ”tail”’s ”next” is currently pointing to and make this ”prev” the new tail (for option G). The options in the slow group don’t have direct access to the second last element and require you to travel through the list to figure it out.

2 Algorithm Explanation

In my solution, I use both Stack and Queue data structures to solve the problem. I implement the Buffer Line using stack and the Exit using Queue. I have a variable called “look_for” to keep track of what number should go to the Exit and the Buffer Line stack is used to store numbers that are not what I am looking for. My algorithm starts with a “for” loop on the input array, if the current element of the array is equal to my “look_for” variable, that element will be put into the Exit and the “look_for” will be increased by 1. If the current element of the array is NOT equal to my “look_for” variable, the program will have to check the Buffer Line stack by starting a “while” loop with the condition that the number at the top of the stack is equal to what “look_for” is and the Buffer Line is not empty, and if that is the case, that number will be popped out of the Buffer Line stack, enqueued to the Exit queue, and “look_for” is increased by 1, if there is no number in the Buffer Line stack that matches “look_for” then the program will push the current-considering element of the array to the top of the Buffer Line stack and move on to consider the next element of the array.

After the “for” loop finishes its job, the program will go to the Buffer Line stack to check for possible missing numbers. Same way as earlier, starting a “while” loop with the condition of the number at the top of the stack is equal to what “look_for” is and the Buffer Line is not empty, if there is a match, the number will be popped out of the Buffer Line and enqueued to the end of the Exit queue and if not, the program will move on to return the Exit queue size. If the size is 0 then it will return 0.