Assignment1

Student number: 490051481

August 17, 2024

Solution 1.

Let's understand the worst running time step-by-step.

Firstly, the complexity for line 2 is *O(1)*; from lecture, we know get the size of an array always take O(1) time;

Secondly, the complexity of line 3 is *O(n)*; since we need to initiate an array with size n - 16 which means the step is related to n, that is why I think the running time here is O(n);

Thirdly, the complexity of outer for-loop which locate in line 4 takes at most O(n-16)=*O(n)* steps; you can check by looking at the condition part for the outer for-loop, it says "we take i from 0 to n-16" which means it will loop for n-16 times;

Fourthly, the complexity of line 5 is constant which takes *O(1)* step; this is just equal to set(i,0) method for array which appear in lecture 2, and it just take O(1) step;

Fifthly, the inner loop takes O(16)= *O(1)* step; by looking at the inner for-loop condition, you may find that it will only loop 16 times no matter what n is.

Sexily, line7 takes *O(1)* step; similar to line 5, we just using index to access element in array A and array B and do a simple calculation; therefore, the time complexity here should be O(1)

Finally, let's combine previous steps. The time complexity equation should be: O(1) + O(n) + O(n)*(O(1)+O(16)*O(1)) = O(1)+O(n)+O(n)=O(n).

Solution 2.

(a)

I will extend the queue that we saw during the tutorial 2 Q2 which implement by linked list. There are 4 mainly differences between mine and the tutorial one. (1) each node in queue has a new variables which stores the sum of the values for previous node; Here previous nodes means predecessors. (2) There are 2 new global variables which is the sum of all dequeue values and the size of single linked list n, respectively; all these 3 variables will change dynamically. (3) There are 3 new pointer, I want to call the pointers which point to the last node in the linked list "lastP"; call the pointer which point to the $\lceil \frac{n}{2} \rceil^{th}$ node "middleP"; call the head pointer "firstP". (4) There is a new operation *DIFFERENCE()*.

Next, I will explain rules for my new data structure and operation

1. for the "sum" variable in each node, we simply do sum ← sum + e.value when we enqueue a new element e. The sum after the arrow represent e.last node's sum (I know single linked list does not have last() method, but I just use it to explain my idea) and it is 0 if it is the first node in single linked list. After we initialize it, it will not change (even we dequeued nodes). Lastly, although we cannot access previous node, we can do it by just accessing "lastP.sum" before updating "lastP".

2. for the pointer "middleP" which point to the $\lceil \frac{n}{2} \rceil^{th}$ node, the rule is: for both enqueue and dequeue operations, if "size" is even number, we do nothing; if "size" is odd number, middleP ← middleP.next; you can find the definition of "size" global variable below. As for pointer lastP and firstP, they are same to Tut2 Q2 (i.e. when Enqueue lastp ← e; when Dequeue firstp ← firstp.next).

3. for the "DequeueValue", we simply do DequeueValue ← DequeueValue + e.value when we dequeue node e (this node is always the first node of singly linked list, I call it e because it can help me to explain). It is 0 if there is no Dequeue operation happened.

4. for the "size" variable, we simply do size ← size + 1 when we enqueue a new element e. we do size ← size - 1 when we dequeue an element.

5. To implement the *DIFFERENCE()* operation. Firstly, we can use $\sum_{i=1}^{\lceil \frac{n}{2} \rceil} Q_i = Q_{\lceil \frac{n}{2} \rceil}.sum - DequeueValue$ to get sum of the "first half" nodes' value. We get $\lceil \frac{n}{2} \rceil^{th}$ node address from "middleP". We minus "DequeueValue" because "sum" variable is fixed which means it will not change when some nodes are dequeued; as for Enqueue operation, enqueued node is not predecessor of $\lceil \frac{n}{2} \rceil$ node, hence we need not to care about it. Secondly, we can use $\sum_{i=\lceil \frac{n}{2} \rceil+1}^{n} Q_i = Q_n.sum - Q_{\lceil \frac{n}{2} \rceil}.sum$ to get the "last half" nodes' value. It works because both Q_n and $Q_{\lceil \frac{n}{2} \rceil}$ include those dequeued nodes' value, so this part will cancel each other. In addition, we get address of node n from "lastP". Finally, we get $\sum_{i=1}^{\lceil \frac{n}{2} \rceil} Q_i - \sum_{i=\lceil \frac{n}{2} \rceil+1}^{n} Q_i = 2 * Q_{\lceil \frac{n}{2} \rceil}.sum - Q_n.sum - DequeueValue$

(b)

Next, I will explain why my data structure and operations satisfied the requirements. Before doing that, for convenience and clarity, I want to call the pointer which point to the last node in the linked list "lastP"; call the pointer which point to the $\lceil \frac{n}{2} \rceil^{th}$ node "middleP"; call the head pointer "firstP".

1. This data structure satisfies enqueue(), dequeue() methods; according to Tut2 Q2 "To enqueue an element we add it at the end of the list. To dequeue an element we remove it from the front of the list.", hence these operation takes O(1) time and they are correct;

2. This data structure satisfies first(), size(), isEmpty(); As for first() , we can do it by checking the pointer "firstP"; As for isEmpty(), we can check whether "firstP" is null; As for size(), we can simpley check the global variable *size*. Since we just access global variables, so these operation takes O(1) time and they are correct;

3. The complexity of calculating "sum" for each node is O(1); since we can do it by just accessing "lastP.sum" before updating "lastP", then we only need to do a simple addition and we get "sum" for the new enqueued node. It must be correct because it need not to consider dequeued nodes and we only do it after the Enqueue operation.

4. The complexity of calculating global variable "size" is O(1); since we only need to plus 1 or minus 1 when we do the Enqueue or Dequeue operation respectively. It must be correct because we only do it after the Enqueue or Dequeue operation.

5. The complexity of calculating global variable "DequeueValue" is O(1); since we only need to minus the value of dequeued element when we do the Dequeue operation, so it is correct and with O(1) complexity.

6. The complexity of calculating global variable "middleP" is O(1); since we just use if statement to judge the condition of "size" and assign "middleP.next" to "middleP"; it is correct because only when n changes from an even number to an odd number could lead to the change of $\lceil \frac{n}{2} \rceil$. For example, when n = 0, we have $\lceil \frac{n}{2} \rceil = 0$; and we can see, when n changes from 0 to 1, $\lceil \frac{n}{2} \rceil = 1$; but when n changes from 1 to 2, $\lceil \frac{n}{2} \rceil = 1$;

7. As for the *DIFFERENCE()* operation, the complexity is O(1) because it only ac-

cess few values and do the simple calculation. It is correct because of 3 reasons. (1) we can just use $Q_{\lceil \frac{n}{2} \rceil}.sum - DequeueValue$ to get sum of the "first half" nodes' value. We minus "DequeueValue" because "sum" variable is fixed which means it will not change when some nodes are dequeued; as for Enqueue operation, enqueued node is not predecessor of $\lceil \frac{n}{2} \rceil$ node, hence we need not to care about it. (2) we can use sum of all nodes' value (include dequeued nodes' value) to subtract sum of "first half" nodes value (include dequeued node's value) to get "last half" nodes' value since "dequeued node's value" parts will cancel each other. (3) now, we only need to make sure "middleP" is correct and I did this in part 6.

(c)
There are n nodes and each node takes 1 space. In addition, each node has a new variable, this variable will also take 1 space. In total, all nodes take O(2n)=$O(n)$ space. Furthermore, there are 3 pointers which takes 3 space (lastP, middleP, firstP). There are 2 global variables "size" and "DequeueValue" and they take O(2) space. Combine them, we get the data structure takes O(n) space. As for the running time of my operations, I did it in part (b).

Solution 3.

(a)

My algorithm is: choose robot 1 as starting point. Firstly, try to find all robots locate in one side of robot 1; then, find all robots locate in another side of robot 1. The step of finding one side robots is: (1)create an array to keep in track of the rest robot IDs; (2) set current node as robot 1, if any robot is neighbour of current node, we just record it as current node's son; (3) changing current node to that son node and remove neighbour robot's Id from array in (1). (4) repeating step (2) and (3). After finish above steps, we just finish one side of robot 1. Then, we just set current node back to robot 1 and repeat above steps (2) and (3) for another side of robot 1 except we store neighbout robot as the father of current node.

Algorithm 1 return the order of roberts

```
1: ArrayCollection ← [2, 3, ..., n]
2: OrderDoublyLinkedList ← [1]
3: Current ← address of node "1" in OrderDoublyLinkedList
4: Origin ← address of node "1" in OrderDoublyLinkedList
5: SuccessfullyFindNeighbourRobot ← True
6: while SuccessfullyFindNeighbourRobot do
7:     SuccessfullyFindNeighbourRobot ← False
8:     neighbourRobotID ← null
9:     for i in ArrayCollection do
10:        if adj(i,element(Current)) then
11:            create new node "i" and make sure its father is Current (node i.last = current)
12:            Current.next ← address of node "i"
13:            Current ← address of node "i"
14:            SuccessfullyFindNeighbourRobot ← True
15:            neighbourRobotID ← i
16:            break this for loop
17:        end if
18:    end for
19:    Remove neighbourRobotID from ArrayCollection
20: end while
21: Current ← Origin
22: SuccessfullyFindNeighbourRobot ← True
23: while SuccessfullyFindNeighbourRobot do
24:    SuccessfullyFindNeighbourRobot ← False
25:    neighbourRobotID ← null
26:    for i in ArrayCollection do
27:        if adj(i,element(Current)) then
28:            create new node "i" and make sure its son is Current (node i.next = current)
29:            Current.previous ← address of node "i"
30:            Current ← address of node "i"
31:            SuccessfullyFindNeighbourRobot ← True
32:            neighbourRobotID ← i
33:            break this for loop
34:        end if
35:    end for
36:    Remove neighbourRobotID from ArrayCollection
37: end while
38: return OrderDoublyLinkedList
```

Now, I will explain my pseudo-code. In *line*1, I initialized an array, this array store all robots' ID except robot 1. In *line*2, I initialized a doubly linked list which only include one node and that node has robot 1 ID. In *line*3 and *line*4, I store the address of the 1st node in doubly linked list. In *line*5, I initialized the while loop condition, its initial value means nothing, but after while loop begin, it represents whether we successfully find a neighbour for *Current* (see *line*7 and *line*14). I use the first while-loop, which begin at *line*6, to check whether *Current* has any neighbour, it will terminate when there is no neighbour for *Current* robot (see *line*7 and *line*14); therefore, I will only illustrate the case when we successfully find a neighbour. In *line*8, I initialized a variable which I will use it to store the ID for the neighbour robot later. The inner for-loop, which begin at *line*9, just go through all robots' ID in *ArrayCollection* and check whether there is a neighbour for *Current* robot. If we successfully find one, then firstly we just create a new node to store the neighbour robot's ID (*line*11). Secondly, storing new node address in *Current.next* (*line*12). Thirdly, changing *Current* address to new node (*line*13), otherwise, we cannot find neighbour for our new node. Fourthly, changing *SuccessfullyFindNeighbourRobot* to True, otherwise, the while loop will terminate. Fifthly, storing the neighbour robot's ID (*line*16), since we need to delete it from *ArrayCollection* when we exist this inner loop. Finally, we just break the inner for-loop and go back to the while-loop (*line*16). After we exist the for-loop, we need to do one more thing before entering next while-loop, that thing is remove *neighbour* ID from *ArrayCollection*. We need to do this because we do not want to get a wrong answer when we try to find *neighbour.next* robot's neighbour (since it may return *neighbour* robot as *neighbour.next* robot's neighbour)(*neighbour* means the neighbour we found in the for-loop). After the first while-loop, we finish finding all robots in the "right side" of robot 1 (since we begin from robot 1, see *line*2). Now, it is time to reset *Current* to the address of node which represent robot 1 (see *line*21). Then we do the similar things again for "another side" of robot 1(see *line*29).

(b)

My algorithm is correct because it consider about all cases.

Case1: start from robot-1, and robot-1 is the most left robot in the x-axis; In this case, my algorithm will try to find a neighbour robot of robot-1. Since robot-1 only have right neighbour, so my algorithm will firstly check all robots in the right side of robot-1 and connect them in order. Then, it will check the left side of robot-1; however, there is no robot in the left side of robot-1 (in pseudo-code, *ArrayCollection* is empty when we do the *line*23 while-loop). Hence, it will finally return a doubly linked list with robots sort in order.

Case2: start from robot-1, and robot-1 is in the middle of x-axis (i.e. there are robots in both side of it); In this case, my algorithm will try to find a neighbour robot of robot-1. In this case, the neighbour could locate in the right side of robot-1 or in the left side of robot-1. Assume, the neighbour locate in the right side, my algorithm will firstly check all robots in the right side of robot-1 and connect them in order. Then, it will check all robots in the left side of robot-1 and connect them in order. Hence, it will finally return a doubly linked list with robots sort in order. The process is similar when we firstly find a neighbour locate in the left side.

Case3: start from robot-1, and robot-1 is the most right robot in the x-axis; prove of the correctness is similar to case 1

(c)

*line*1 takes $O(n-1)$ because it initialized an array with n-1 element; *line*2,*line*3,*line*4,*line*5 take $O(1)$ because they are just simple assign operation. The worst running time of while-loop in *line*6 is $O(n-1)$, because it is decided by *SuccessfullyFindNeighbourRobot* which take at most n-1 time to "become False" (actually, it does not change, it just stay as False when we traverse all element in *ArrayCollection*). *line*7, *line*8 take $O(1)$. The worst running time of for-loop in *line*9 is O(n-1), since the size of *ArrayCollection* is n-1 (the last robot ID in *ArrayCollection* is the neighbour). All operations from *line*10 to *line*18 just take $O(1)$ times (these operation are just simple assign operation). According to lecture 2, remove an element from array with length n take $O(n)$, so we have*line*19 takes $O(n-1)$. *line*21 and *line*22 take $O(1)$. For *line*1 to *line*22, we can combine those together which give $O(n-1) + O(1) + O(n-1) * [O(1) + O(n-1) * O(1) + O(n-1)] + O(1) = O(n^2)$ complexity. Because we do the similar operations from *line*23 to *line*38, therefore, the complexity of that part is also $O(n^2)$. Finally, the complexity of all algorithm is $O(n^2) + O(n^2) = O(n^2)$.