assignment-2

Tuesday, 29 October 2024 7:41 PM

L

C-2.1 Describe, in pseudo-code, a link-hopping method for finding the middle node of a doubly linked list with header and trailer sentinels, and an odd number of real nodes between them. (Note: This method can only use link-hopping, i.e., the methods **before** and **after**; it **cannot** use a counter.) What is the running time of this method?

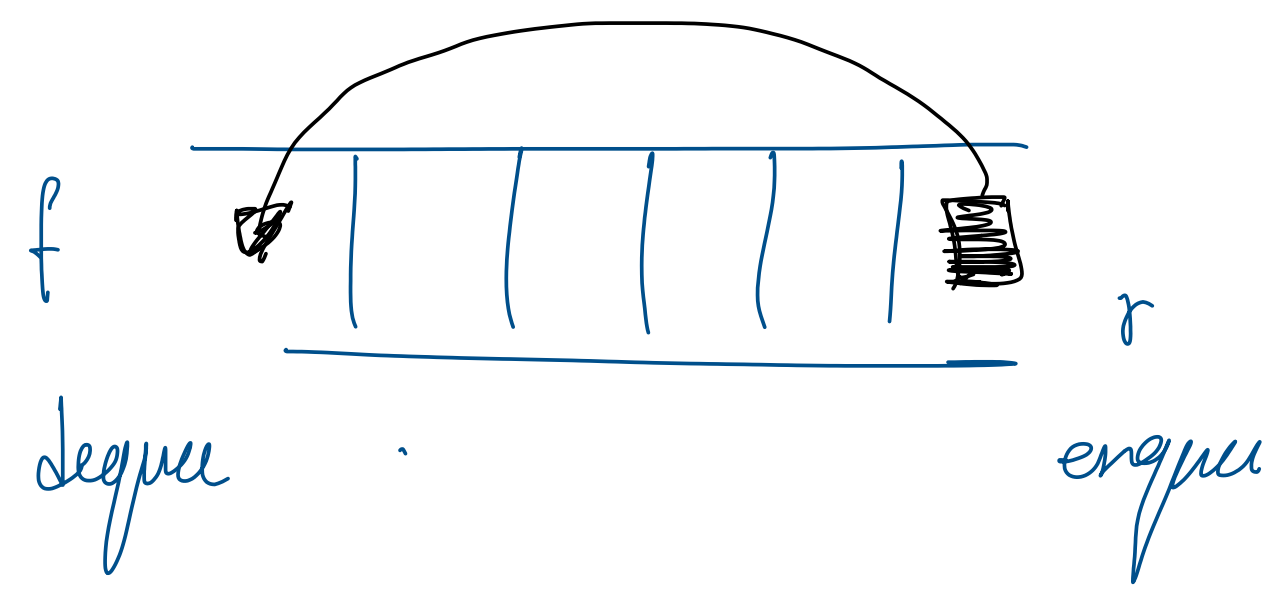
.

```
Algorithm FindMiddle( list )
    Input : list of odd number of nodes.
    output: middle position of list

    L := list.header.after()          O(1)
    R := list.trailer.before()        O(1)

    While L != R then                 O(n)
        L := L.after()                O(n)
        R := R before()               O(n)

    return L ;                        O(1)
```

Running Time = O(n)

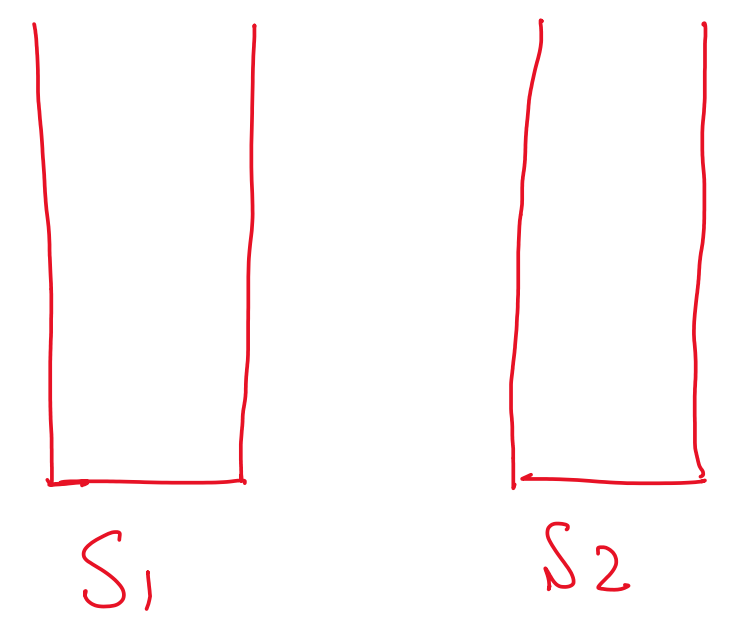
C-2.3 Describe, in pseudo-code, how to implement the stack ADT using one queue. What is the running time of the push() and pop() methods in this case?

```
q ← new Queue ()

Algorithm Push (element)
    q.enqueue (element)                    O(1)

    for i from 1 to q.size() -1            O(n)
        q.enquen (q. dequeue())            O(n)

Algorithm pop()

    if q.isEmpty () then                   O(1)
        throw EmptyStackException          O(1)
    return q.dequeue ()                    O(1)
```



Running Time O(n)

C-2.2 Describe, in pseudo-code, how to implement the queue ADT using two stacks. What is the running time of the enqueue() and dequeue() methods in this case?



```
S1 ← empty stack                        O(1)
S2 ← empty stack                        O(1)

Algorithm Enqueue (val)                 O(1)
    if S.Size() = n-1 then              O(1)
        throw FullQueueException
    S1. push( val)                      O(1)
```

Running Time O(1)

```
Algorithm Dequeue () .
    if S2.isEmpty() then                        O(1)
        while (!S1.isEmpty()) Do                O(n)
            S2.push( S1.pop() )                 O(n)
    if !S2.isEmpty() then                       O(1)
        return S2.pop()                         O(1)
    else                                        O(1)
        throw EmptyStackException               O(1)
```

Running Time O(n)

A. Design a pseudo code algorithm to take a Sequence and remove all duplicate elements from the Sequence. Is the algorithm the same for both a List or a Sequence? Explain. Analyze your algorithm twice, once assuming it is a Sequence and once assuming it is a List. Which ADT is a better choice for this problem, i.e., does one version have a better big-O running time over the other?

```
Algorithm removeDuplicates (sequence)

    Seen = Set()                                        O(1)
    position = sequence.first()                         O(1)
    while position is not null:                         O(n)
        element = position.element ()                   O(n)
        if element is in seen:
            nextposition = sequence.after( position)    O(n)
            sequence.remove (position)                  O(n)
            position = nextposition                     O(n)

        else
            seen.add (element )                         O(n)
            position = sequence.after (position )       O(n)
```

Running Time O(n)

If Sequence

- . the operations first(), after(), remove() are O(1)
- . adding to seen is O(1)

total O()

↳ O(n) as we iterate over n elements

If List

- Operations first(), after(), remove() are also O(1)
- and list allows efficient insertion & deletion

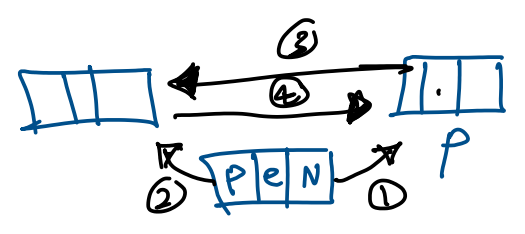
total O( )

↳ O (n)

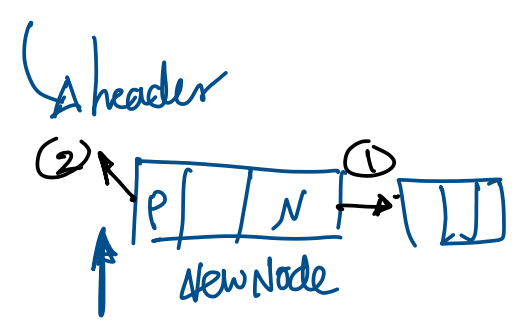
So Sequence & list Can achieve O(n) time Complexity

R-2.1 Describe, using pseudo-code, implementations of the methods insertBefore(*p*,*e*) , insertFirst(*e*), and insertLast(*e*) of the List ADT, assuming the list is implemented using a doubly-linked list.

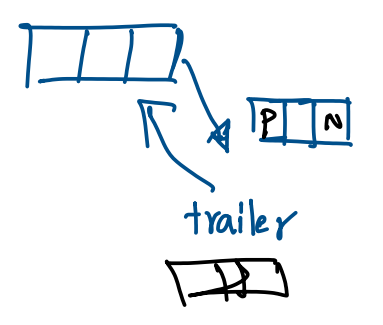
```
Algorithm InsertBefore (P, e)
                           position  element

    newNode = Node (e)
    newNode.next = P;
    newNode. prev = P.prev.

    P.prev.next = newNode.
    P.prev = newNode.

    return newNode.
```



```
Algorithm InsertFirst (e)
    newNode = Node(e)
    newNode.next = header.next
    newNode.prev = header
    header.next.prev = newNode
    header.next = newNode
    return newNode.
```

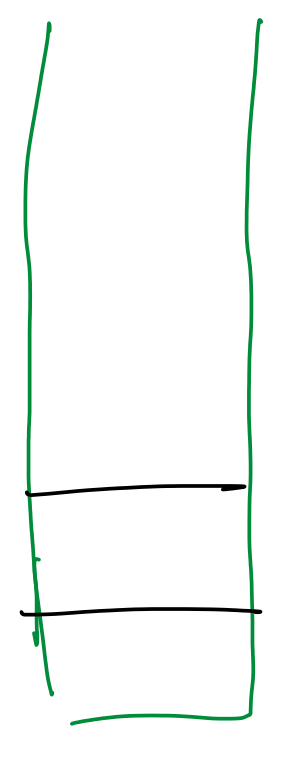


```
Algorithm InsertLast (e)
    newNode = Node (e)
    newNode.next = trailer
    trailer.prev.next = newNode.
    trailer.next = newNode
    return newNode;
```



A. Write a pseudo-code function, **isBalanced(arr)**, that takes an array **arr** of characters and determines whether or not the array contains balanced square brackets [ ], parenthesis ( ), and braces { }. For example, if **arr** contains [(){8}](), then **isBalanced** would return true, but if **arr** contains [( ) { 8 }] ( )], then false would be returned since there is an extra] at the end. All other characters should be ignored, e.g., like 8 in the examples.

```
Algorithm isBalanced (arr)
    stack = new stack()
    matchingPair = { ']':'[', ')':'(', '}':'{' }

    for each char in arr:
        if char is one of '[', '{', '(':              // if opening parenthesis
            stack.push (char)
        else if char is one of ']', '}', ')':
            if stack.isEmpty()                          || if empty
                return false
            top = stack.pop()
            if top != matchingPair[char]
                return false

    return stack.isEmpty()
```



stack