Assignment 2

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; it **cannot** use a counter.) What is the running time of this method?

FindMidle(l)

If l.head.next()==null then O(1)

Return null  O(1)

f:=l.head.next() O(1)

r:=l.tail.prev() O(1)

while f!=r ?/\ f.next() !=r do O(n)

f🡨f.next() O(n)

r🡨 r.prev() O(n)

return f.element() O(1)

Total: 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?

n:=maxSize

s1:=new Stack of n integers

s2:=new Stack of n integers

**enqueue (e)**

if s1:size()=n-1 then O(1)

throw FullQueueException O(1)

s1.push(e) O(1)

Total= O(1)

**dequeue ()**

output=: element same type of stack type O(1)

if s2.isEmpty() then O(1)

while !s1.isEmpty() do O(n)

s2.push(s1.pop()) O(n)

if !s2.isEmpty() then O(1)

output🡨 s2.pop() O(1)

while !s2.isEmpty() do O(n-1)

s1.push(s2.pop()) O(n-1)

if output!=null

return output

else

throw EmptyQueueException O(1)

Total= O(n)

C-2.3 Describe how to implement the stack ADT using two queues. What is the running time of the push() and pop() methods in this case?

n:=maxSize

q1:=new Queue of n integers

q2:=new Queue of n integers

push(e)

if q1.size()=n-1||q2.size()=n-1 then O(1)

throw FullStackException O(1)

if q1.isEmpty() O(1)

q2.enqueue(e) O(1)

else O(1)

q1.enqueue(e) O(1)

Total : O(1)

pop()

output:= element same type of queue type O(1)

if q1.isEmpty() || q2.isEmpty() O(1)

return EmptyStackException O(1)

if q1.isEmpty() O(1)

while q2.size()>1 O(n)

q1.enqueue(q2.dequeue()) O(n)

output🡪 q2.dequeue() O(1)

else

while q1.size()>1 O(n)

q2.enqueue(q1.dequeue()) O(n)

output 🡪 q1.dequeue() O(n)

if output!=null O(1)

return output O(1)

Total: O(n)

1. 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 running time over the other?

Algorithm removeDub(s)

For i:=0; i< s.size(); i++ O(n)

For j:=0; j< s.size(); j++ O(n2)

if s.elemAtRank(i) == s.elemAtRank(j) O(n2)

s.removeAtRank(j) O(n2)

Total: O(n2)

Algorithm removeDub(l)

For i:=0; i< l.size(); i++ O(n)

For j:=0; j< l.size(); j++ O(n2)

if l.get(i) == l.get(j) O(n2)

l.removeItemAtIndex(l,l. j) O(n)

Total: O(n2)

removeItemAtIndex (list,index)

if index == list.size()-1

list.pop()

else

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.

**insertFirst(e)**

**node := new Node with e Element**

node.next() 🡨 head

node.prev() 🡨 null

head.prev() 🡨 node

head 🡨 node

**insertLast(e)**

**node := new Node with e Element**

node.next() 🡨 null

node.prev() 🡨 tail

tail.next() 🡨 node

tail 🡨 node

**insertBefor(bNode, e)**

**node := new Node with e Element**

node.next() 🡨 bNode

node.prev() 🡨 bNode.prev()

bNode.prev().next() 🡨 node

bNode.prev() 🡨 node