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?**

|  |  |
| --- | --- |
| Algorithm findMiddleNode(L)  if L.isEmpty() then return null  headP = L.head()  tailP = L.tail()  while headP != tailP do  headP = L.after(headP)  tailP = L.before(tailP)  return headP | O(1)  O(1)  O(1)  O(n/2)  O(n/2)  O(n/2)  O(1)  Also running time is O(n/2) |

**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?**

|  |  |
| --- | --- |
| Stack stack1, stack2  Algorithm *enqueue*(*o*)  if stack1.*size*() = *N* then  throw *FullQueueException*  else  *stack1.push(o)* | O(1)  O(1)  O(1)  Total running time: O(1) |
| Algorithm *dequeue*()  if *isEmpty*() then  throw *EmptyQueueException*  else  if stack2.isEmpty() then   while !stack1.isEmpty() do  stack2.push(stack1.pop())  return stack2.pop() | O(1)  O(1)  O(n)  O(n)  Total running time: 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?**

|  |  |
| --- | --- |
| Queue queue1, queue2  Algorithm *push*(*o*)  if queue1.*size*() = *N* then  throw *FullStackException*  else  *queue1.enqueue(o)* | O(1)  O(1)  O(1)  Total running time: O(1) |
| Algorithm *pop*()  if *isEmpty*() then  throw *EmptyStackException*  else  if queue1.isEmpty()  return null    while queue1.size() > 1 do  queue2.enqueue(queue1.dequeue())   popped = queue1.dequeue()  while !queue2.isEmpty() do  queue1.enqueue(queue2.dequeue())  return popped | O(1)  O(1)  O(n)  O(n)  O(1)  O(n)  O(n)  Total running time: O(n) |

1. **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 *removeDuplicateSequence*(S)  If S.*isEmpty*() then  throw *EmptySequenceException*  p := S.first()  while !S.isLast(p) do  a := S.after(p)  if p.element() == a.element() then  remove(p)  p := S.after(a)  return S | O(1)  O(1)  O(n)  O(n)  O(n)  O(n)  O(n)  O(1)  Total running time: O(n) |
| Algorithm *removeDuplicateList* (L)  if L.*isEmpty*() then  throw *EmptyListException*  L = L.sort()  return *removeDuplicateSequence(L)* | O(1)  O(n)  O(n)  Total running time: O(2n) |

Explaination: Because list is not order, but sequence is ordered. Sequence is better than List.

1. **Describe a recursive algorithm for enumerating all subsets of the numbers {1,2,…,n}, i.e., the powerset of the elements of a Sequence; the result should be a Sequence containing Sequences. What is the running time of your method?**

|  |  |
| --- | --- |
| Algorithm enumerateSubsets(n)  If n = 0 then  return [[]]  else  subsets = enumerateSubsets(n-1)  newSubsets = []  for subset in subsets do  newSubsets.append(subset)  newSubsets.append(subset)  return newSubsets | O(n)  O(n)  O(n)  Total running time: O(2n) |

**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.**

|  |  |
| --- | --- |
| insertBefore(p e)  if p.isEmpty() then  throw InvalidOperationException  if !p.prev then  p.prev = e  e.next = p  else  q := p.prev  q.next := e  e.prev := q  e.next := p  p.prev := e  insertFirst(e)  insertbefore(first(), e)  insertLast(e)  if p.isEmpty() then  throw InvalidOperationException  p = last()   p.next = e  e.prev = p | O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1) |

**Optional if you have time:**

**C-2-5 Describe the structure and pseudo-code for an array-based implementation of the vector ADT that achieves O(1) time for insertions and removals at rank 0, as well as insertions and removals at the end of the vector. Your implementation should also provide for a constant-time elemAtRank method.**