Week 2 Solutions

Solutions for week 2 of Algorithms and Data Structures. Note that the students are asked to describe or explain their solution, not actually implement the algorithms for most of these questions. Pseudo code is still a good way to describe it, and is used extensively below.

1.2.6 - Green

```
def isCircularShift(s: String, t:String) -> Boolean:
    return s.length == t.length AND (s+s).contains(t)
```

1.3.7 - Green

```
#For linked list implementation
def peek():
    if not isEmpty():
        return first.item

#For array implementation
def peek():
    if not isEmpty():
        return a[N-1]
```

1.3.19 - Green

```
Node current = first
while current.next.next is not null:
    current = current.next

current.next = null
size--
```

1.3.20 - Green

```
def deleteKthElement(k: int):
    if size > k:
        current = first
        for i=2 to k-1:
            current = current.next
        current.next=current.next.next
        size--
```

1.3.21 - Green

```
def find(key: String, list:Linked List):
    current = list.first
    while current is not null:
        if current.item == key:
            return true
        current = current.next
    return false
```

1.3.22 - Green

It inserts node t immediately after node x.

1.3.23 - Green

When it comes time to update t.next, x.next is no longer the original node following x, but is instead t itself!

1.3.27 - Green

```
def max(node: first node in a linked list):
    if node is null:
        return 0
    max = node.item
    current = node
    while current.next is not null:
        current = current.next
        if current.item > max:
            max = current.item
    return max
```

1.3.28 - Yellow

```
def max(node: first node in a linked list):
    if node is null:
        return 0
    nextMax = max(node.next)
    if node.item > nextMax:
        return node.item
    else:
        return nextMax
```

1.3.24 - Yellow

```
def removeAfter(node: Linked-list node):
    if node and node.next are not null:
        node.next = node.next.next
```

1.3.25 - Yellow

```
def insertAfter(n1: Linked-list node, n2: Linked-list node):
    if n1 and n2 are not null:
        node2.next = node1.next
        node1.next = node2
```

1.3.31 - Red The primary difference is the additional pointer to the previous node in the list, which needs to be considered for all the operations. The entire doubly-linked list class may also have a pointer to the last element of the list, this is assumed in the pseudo code below and has to be considered for every method.

```
def insertAtBeginning(newNode: a DoubleNode to be inserted):
    if first is null:
        last = newNode
    else:
        first.previous = newNode
        newNode.next = first
    first = newNode

def insertAtEnd(newNode: a DoubleNode to be inserted):
    if last is null:
        first = newNode
    else:
        last.next = newNode
        newNode.previous = last
    last = newNode

def removeAtBeginning():
    if first is null:
        return
    first = first.next
    if first is null:
        last = null
    else:
        first.previous = null

def removeAtEnd():
    if last is null:
        return
    last = last.previous
    if last is null:
        first = null
    else:
        last.next = null
```

```
def insertBefore(node: node to insert before, newNode: node to insert):
    current = first
    while current is not null:
        if current == node:
            if current.previous is not null:
                current.previous.next = newNode

            newNode.previous = current.previous
            newNode.next = current
            current.previous = newNode

            if current = first:
                first = newNode

            return
        else:
            current = current.next

def insertAfter(node: node to insert after, newNode: node to insert):
    current = first
    while current is not null:
        if current == node:
            if current.next is not null:
                current.next.previous = newNode

            newNode.next = current.next
            newNode.previous = current
            current.next = newNode

            if current = last:
                last = newNode

            return
        else:
            current = current.next

def removeNode(node: node to remove):
    current = first
    while current is not null:
        if current == node:
            if current = first:
                removeAtBeginning()
                return
            if current = last:
                removeAtEnd()
                return

            current.next.previous = current.previous
            current.previous.next = current.next

            return
        else:
            current = current.next
```

1.3.48 - Red We consider the left side of the deque to be stack1 and the right side of the deque to be stack2. So if we push something to stack1, we call pushLeft() to the deque, and if we pop something from stack1 we call popLeft(). And likewise for stack2 with pushRight() and popRight(). To avoid an empty stack drawing from the bottom of the other stack, we keep track of the current number of elements in both stacks and only allow pop() if there are elements remaining in that particular stack.