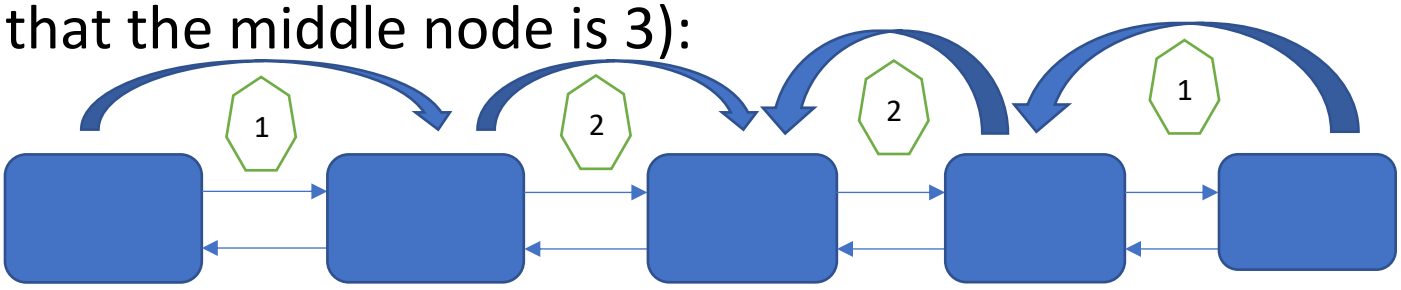
Name: Georges Hatem

# CS590 Homework 2: Creativity Exercises

Due Date: February 6, 2022

**Problem 2.5.11:**

Since we have an odd number between them, we must do the link hopping method from the head and from the tail until we get the same node from both sides. If this was even number between them, this would not have worked (as you can see in the illustrated example below: suppose we have 5 nodes, then 5/2 = 2, which means that the middle node is at 2. Same thing if we have 7 nodes, then 7/2 = 3, which means that the middle node is 3):




As you can see from above, both of them goes through 2 changes when n = 5 and then they become equal to each other from both sides at the end of the 2nd time.

**From above and my analysis, we can write the following pseudocode:**

**Algorithm findMiddle(L):**
**Input: List L with odd size**
**Output: Middle Node Position**

$\boldsymbol{p \leftarrow L.first()}$
$\boldsymbol{q \leftarrow L.last()}$
$\boldsymbol{while\ p\ != q\ do}$
  $\boldsymbol{p \leftarrow L.next(p)}$
  $\boldsymbol{q \leftarrow L.prev(q)}$
**return p**

**The running time of this Algorithm is O(n) for the following analysis:**

1) Assigning p to L.first() is one primitive operations
2) Assigning q to L.last() is one primitive operations

3) The while loop runs n/2 times since we are looking for the middle node position. So, p and q equalities inside of the while loop each run n/2 times
4) P gets compared to q ((n/2) + 1) times. The while loop runs n/2 times. However, p gets compared one more time when p = q but this does not enter the loop. So, the loop runs n/2 times. However, p gets compared to q ((n/2) + 1) times
5) Return p is one primitive operations

Based on the analysis above, we have the following:

$$(1) + (1) + (1) + \left(2 * \left(\frac{n}{2}\right)\right) + \left(\left(\frac{n}{2}\right) + 1\right) = 3 + n + \left(\frac{n}{2}\right) + (1) = \frac{3n + 8}{2}$$

**Based on the above calculation, you can see that the running time of this Algorithm is O(n)**

**Problem 2.5.20:**

Assuming that the depth (d) is a variable associated with each node, we can compute node's depth as follows:

**Algorithm depthsComputations(v, d):**

**v.depth ← d**

**if(v has no child)**
**return 0**

**else**

**for each child v' do**

**depthsComputations(v', d+1)**

**return (v.depth)**

**To compute the depth of each node, simply call depthsComputations(root, 0). This algorithm visits each node only once and each visit takes constant time. Therefore, this Algorithm running time is O(n).**