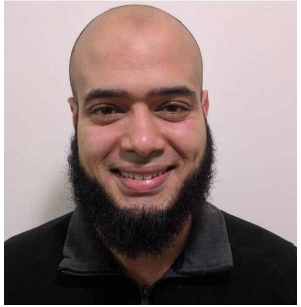
Data Structures

Drawings

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

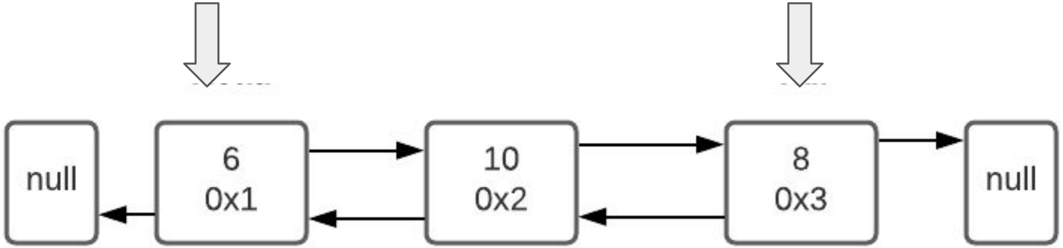


Problem #1: Find the middle

- Given a linked list, we would like to find its middle value
  - In odd length list, e.g. {1, 2, 3, 4, 5}, the middle value is 3
  - In even length list, e.g. {1, 2, 3, 4, 5, 6}, the middle values are {3, 4}. We need 2nd one {4}
- Provide 2 implementations, but consider:
  - You can’t iterate on the list more than once!
  - Don’t use the length variable!
- First: Use your doubly linked list
- Second: Solve it only with the next pointer. Don’t use the previous
  - ~5 lines of code.

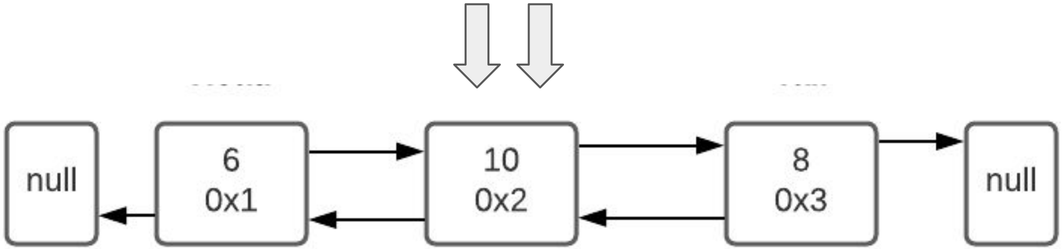
Problem #1: Find the middle using DLL

- Assume odd number of nodes
- Let's start from head and tail and move one step
- Then we must come to point where both are at the same: forward = backward



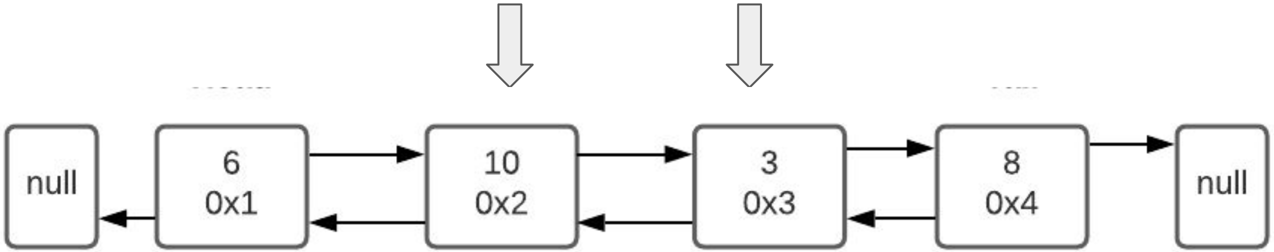
Problem #1: Find the middle using DLL

- Assume odd number of nodes
- Let’s start from head and tail and move one step
- Then we must come to point where both are at the same: forward = backward



Problem #1: Find the middle using DLL

- For the even number of nodes, the 2 pointers will be neighbours!
- So overall. Move the head and tail copies till either
  - They are equal (odd)
  - They are neighbours (even)
- h != t && h->next != t

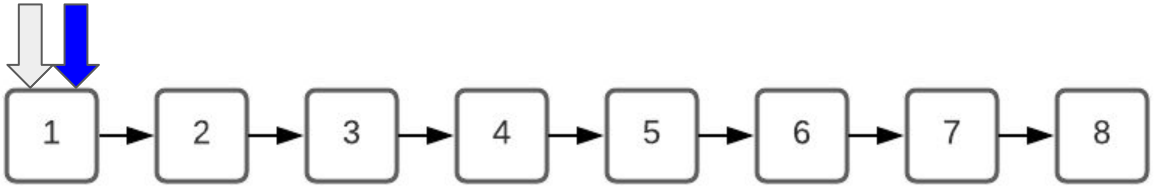


Problem #1: Find the middle using **SLL**

- Using next only is an interesting idea, but hard to get by yourself!
- Use 2 pointers:
  - The first (slow) moves normally step by step
  - The second (fast) jump 2 steps each time!
- If the list has e.g. 10 elements
  - When the slow in the middle (e.g. 5), the fast is at the double at the end! (10)
- From that we know we found the middle.

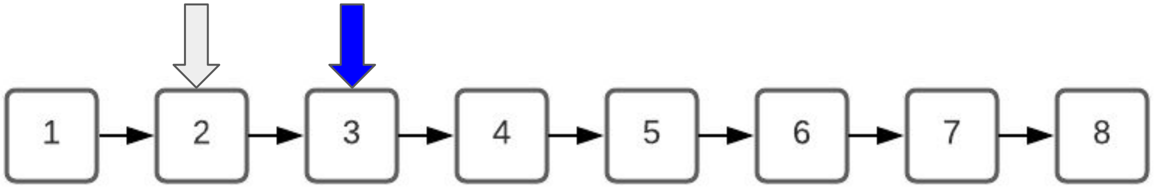
Problem #1: Find the middle using SLL

- Assume Blue is fast pointer
  - Slow moves: 1, 2, 3, 4, 5, 6, 7, 8
  - Fast moves: 1, 3, 5, 7, null
  - Once fast is done, slow MUST be at the middle as fast moved twice the slow



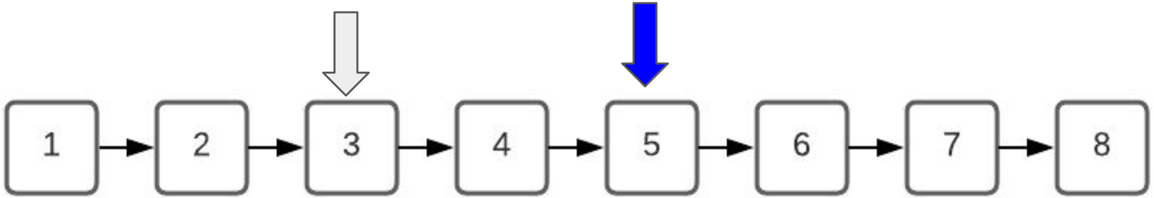
Problem #1: Find the middle using SLL

- Assume Blue is fast pointer
  - Slow moves: 1, 2, 3, 4, 5, 6, 7, 8
  - Fast moves: 1, 3, 5, 7, null
  - Once fast is done, slow MUST be at the middle as fast moved twice the slow



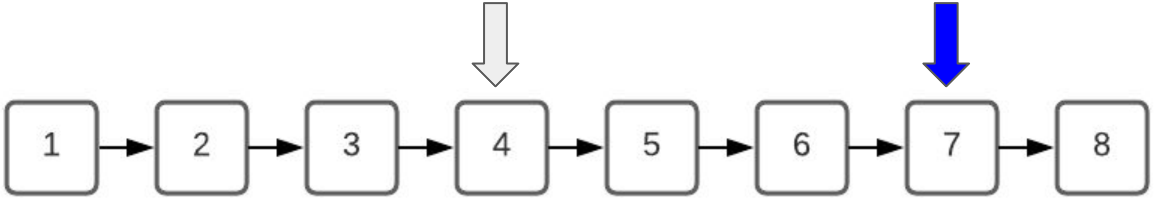
Problem #1: Find the middle using SLL

- Assume Blue is fast pointer
  - Slow moves: 1, 2, 3, 4, 5, 6, 7, 8
  - Fast moves: 1, 3, 5, 7, null
  - Once fast is done, slow MUST be at the middle as fast moved twice the slow



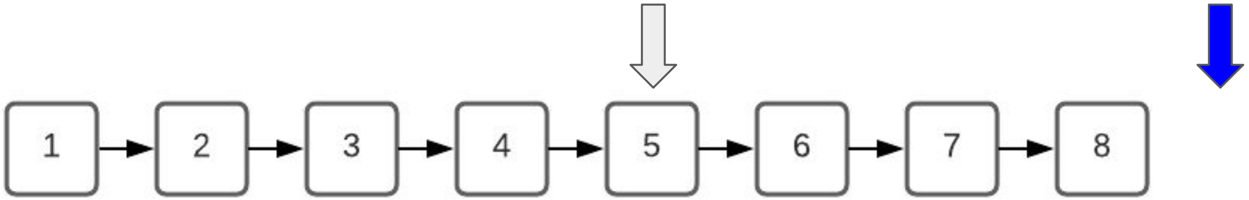
Problem #1: Find the middle using SLL

- Assume Blue is fast pointer
    - Slow moves: 1, 2, 3, 4, 5, 6, 7, 8
    - Fast moves: 1, 3, 5, 7, null
    - Once fast is done, slow MUST be at the middle as fast moved twice the slow



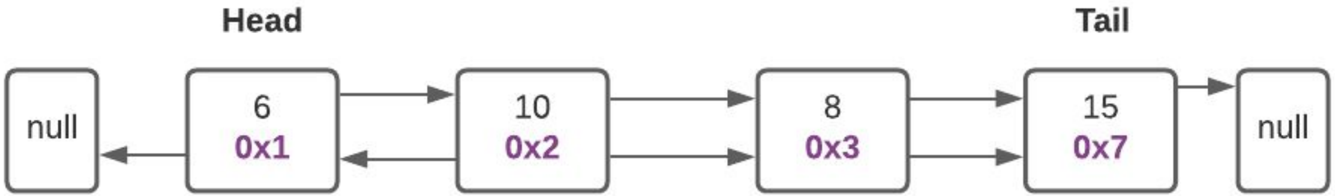
Problem #1: Find the middle using SLL

- As we need the 2nd middle, in both even/odd we see we are at right location
- *The idea is based on tortoise and hare algorithm [no need to check]*
  - You may meet later again



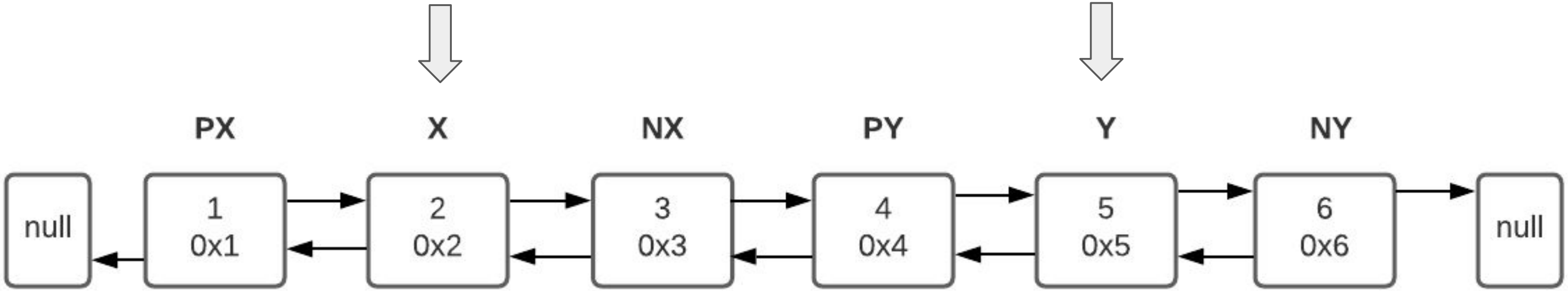
Problem #2: Swap forward with backward

- Given K, find the kth node from forward and backward
  - Swap them (**address** not values)
  - For example: for k = 1, we swap head (0x1) and tail (0x3)
  - For example: for k = 2, we swap nodes 0x2 and 0x3 ⇒ **(6/0x1), (8/0x3), (10/0x2), (15/0x7)**
  - Trick cases. Think and consider
- 2 implementations
  - Utilize the length variable in the list
  - Without the length variable totally



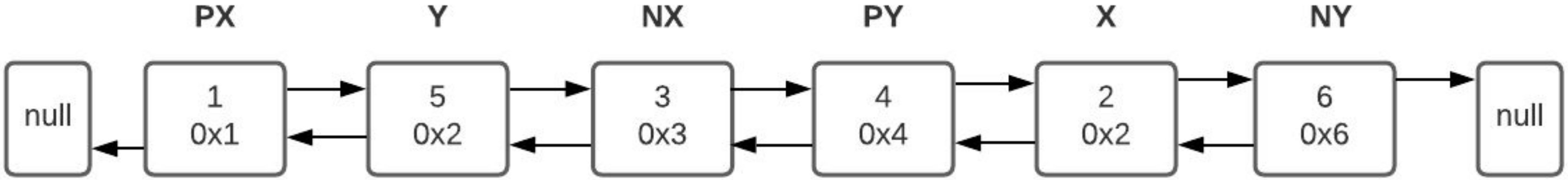
Problem #2: Swap forward with backward

- Assume k = 2. Let’s say nodes are X and Y
- Let’s say previous and next of X are PX and NX
  - Same for Y: PY and NY
- Take copies from all
- Draw the before and after. This helps trivially relink them!



Problem #2: Swap forward with backward

- Clearly we need to link the following pairs: (PX, Y), (Y, NX), (PY, X), (X, NY)
- Careful with cases:
  - Both nodes are the same. Happens only for odd length list
  - First node is after the last node. E.g. for K = 5
  - They are consecutive (neighbours). E.g. for K = 3
- They are all trivial to detect using length. Little more effort without it



Problem #3: Reverse list nodes

- Given a list, reverse all its nodes (addresses)
- E.g. {1, 2, 3, 4, 5} ⇒ {5, 4, 3, 2, 1}
- void reverse()

Problem #3: Reverse list nodes

- With SLL, reverse is easy. When comes to DLL, little more caution
- Move left to right. Reverse current 2 nodes then move
    - Assume list has nodes: A, B, C, D, E, F
    - Start from the begin with 2 consecutive nodes: A and B
    - Take copies of the next 2 nodes (B, C)
    - Link(B, A) now they are reversed
        - B, A   C, D, E, F
    - Move one step using the copies of B and C
    - In next step it will be
    - C, B, A   D, E, F
    - In end, handle proper head & tail

Problem #4: Merge lists

- Assume we have 2 sorted linked lists, of sizes n and m
- We would like to merge them together in O(n+m) but remain sorted
- void merge_2sorted_lists(LinkedList &other)
- E.g. list1 {10,20,30,40,50} and list2 {15,17,22,24,35}
  - ⇒ 10 15 17 20 22 24 30 35 40 50
- Consider the different cases!

Problem #4: Merge lists

- Start from a null pointer (we can do that in a DLL, but not in SLL)
- In each step, see which one has the smaller value? Pick from it
    - Keep doing as long as both sequence are not finished!
- After that, one of them remains. Just connect whole sequence with current one!
- Maintain correct data integrity

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."