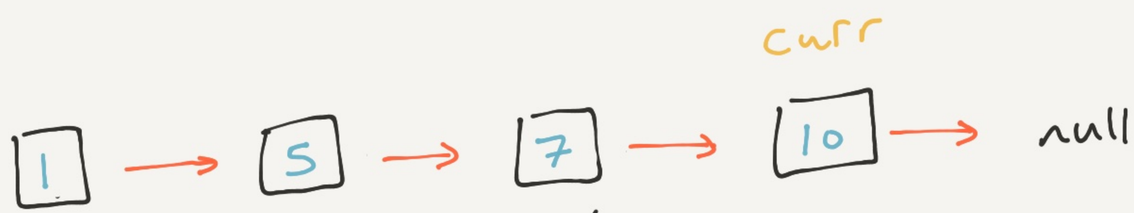
Linked Lists

Print Forward

```
ListNode {
    int data
    ListNode next
}
```



curr.next != null

```
while (curr != null) {
    print (curr.data)
    curr = curr.next
}
```

Time: O(n)

Space: O(1)

* Be carefull about using curr.next != null as your terminating condition otherwise you may run into off by one errors!

Linked Lists

Print Backward

```
ListNode {
    int data
    ListNode next
}
```


```
function (node)
```

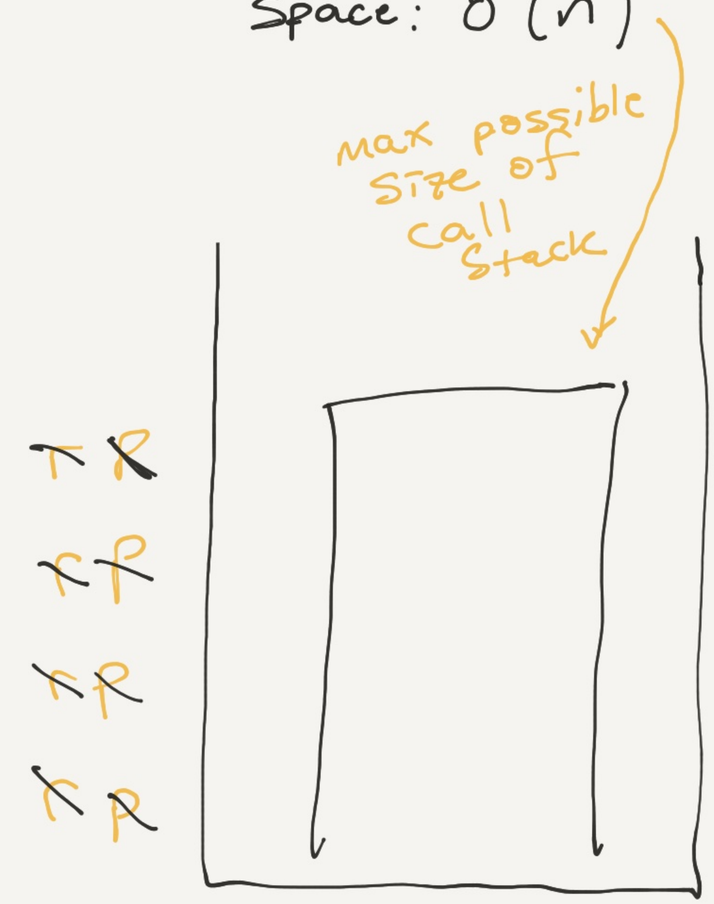
Base Case

```
if (node. == null)
    return
```

Recursive Step

```
reverse (node.next)
print (node.data)
```

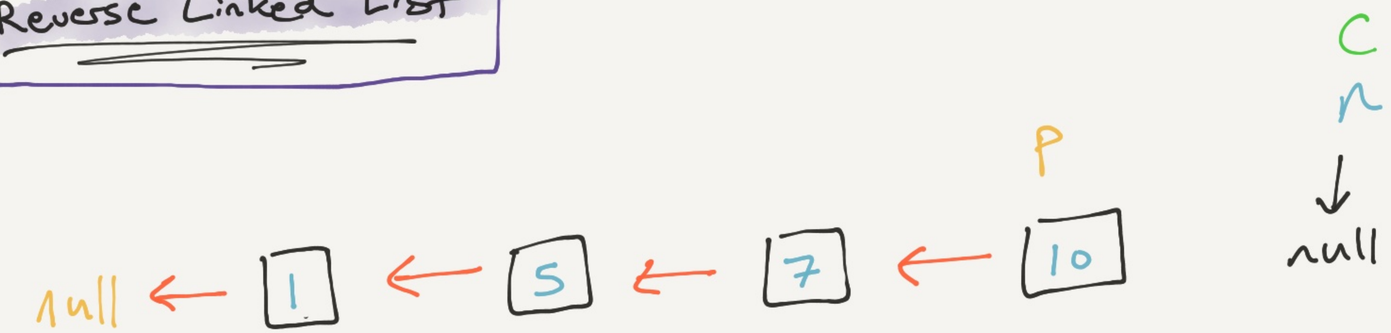
Time: O(n)
Space: O(n)



Linked Lists

Reverse Linked List

```
ListNode {
    int data
    ListNode next
}
```

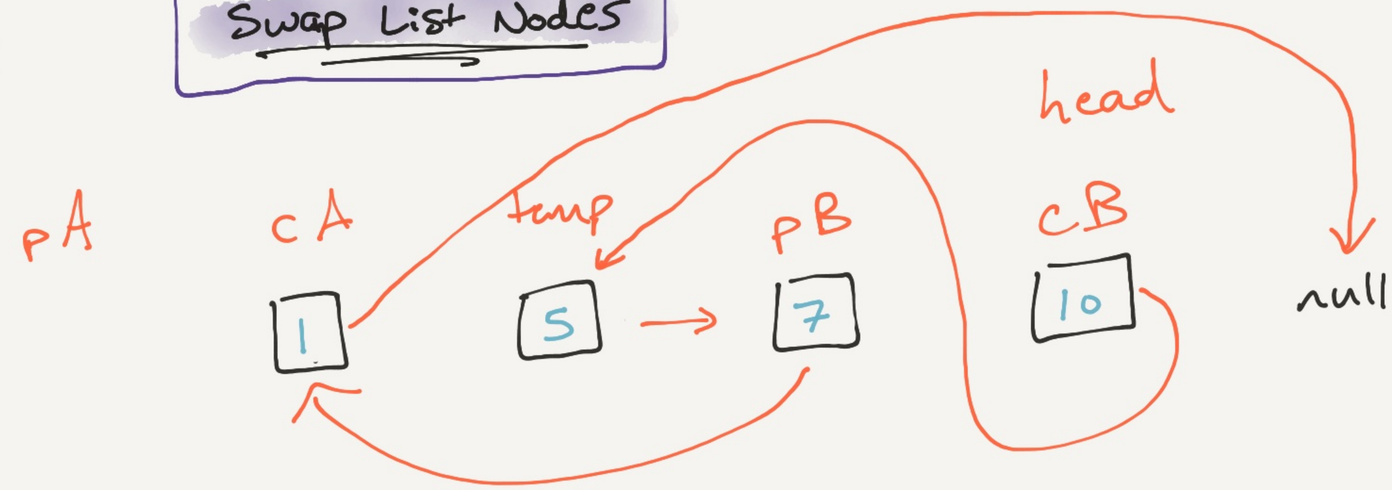



```
function reverse(node)
    prev = null;
    curr = node;
    next = null
    while (curr != null) {
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
    }
return prev
```

☆ Consider ListNode pointers as references that you can set aside for later use

Linked Lists

Swap List Nodes



0. if a == b return head

1. Search for nodes and their ancestors

2. Check that both nodes found

3. Swap next pointers of pA & pB

   check if cA or cB is head

   a) if pA == null // cA is head
      pB.next = cA
      head = cB

   b) if pB == null // cB is head
      pA.next = cB
      head = cA

   c) else Swap pA.next & pB.next normally

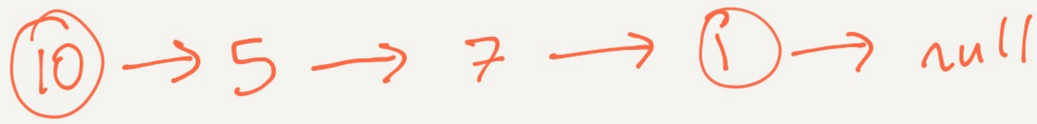
4. Swap next pointers of cA & CB

   temp = cA.next

   cA.next = cB.next

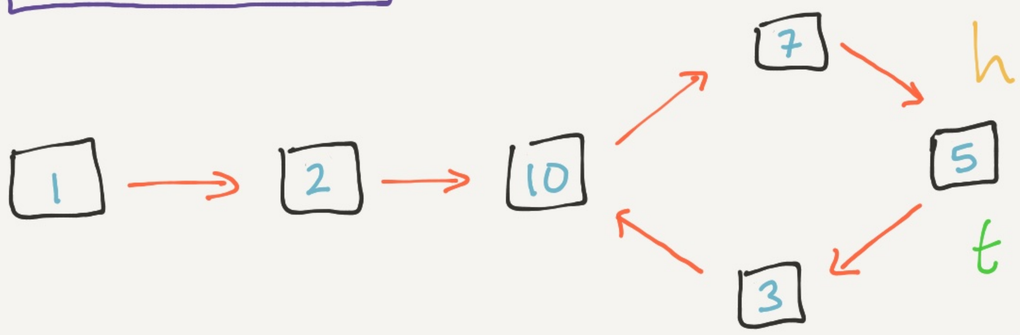
   cB.next = temp

5. Return head node



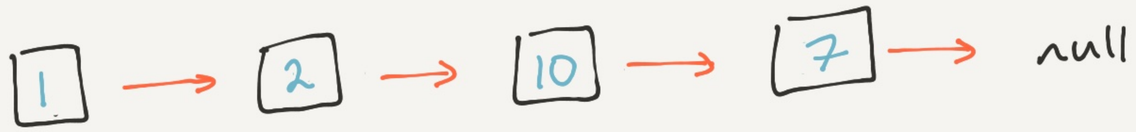
Linked Lists

Is Circular




```
ListNode {
    int data
    ListNode next
}
```

☆ Use two pointers moving @ different speeds. If the fast pointer becomes null, the list terminates. Else, the two pointers will eventually meet



Seen/visited set Approach

Time: O(n)

Space: O(n)

Tortoise/Hare Approach

Time: O(n)

Space: O(1)