Sara Reidy

BASIC INFO

Email: sara-reidy@lambdastudents.com
Test: Lambda School - Whiteboard Fitness Assessment I
Finished On: 26 Aug 2020
Solved: 3/3
Score: 175/175
Time Taken: 99m/3h

Labels: -

Task	Solve Time	Score	Similarity
remove_kth_from_end	41min	50/50	none
tree_paths_sum	13min	50/50	none
merge_packages	13min	75/75	none

Task details: **remove_kth_from_end**

Description:

Write a function that receives as input the head node of a linked list and an integer k. Your function should remove the kth node from the end of the linked list and return the head node of the updated list.

For example, if we have the following linked list:
`(20) -> (19) -> (18) -> (17) -> (16) -> (15) -> (14) -> (13) -> (12) -> (11) -> null`

The head node would refer to the node `(20)`. Let k = 4, so our function should remove the 4th node from the end of the linked list, the node `(14)`.

After the function executes, the state of the linked list should be:
`(20) -> (19) -> (18) -> (17) -> (16) -> (15) -> (13) -> (12) -> (11) -> null`

If k is longer than the length of the linked list, the linked list should not be changed.

Can you implement a solution that performs a single pass through the linked list and doesn't use any extra space?

Note: When reading the tests, the linked list contents are enumerated in between square brackets; this does NOT mean the inputs are arrays.

For example, a test input of `head: [2, 4 ,6]` indicates that the input is a singly-linked list
`(2) -> (4) -> (6) -> null` whose head is the first element in the linked list.

Solution (main.js):

```
// Singly-linked lists are already defined with this interface:
// function ListNode(x) {
//   this.value = x;
//   this.next = null;
// }
//
function remove_kth_from_end(head, k) {
    // init var for node that will go to the end
    let tailNode = head

    // init var for node that will stay k steps behind
    let kNode = head

    // init counter to make sure we're able to get to the end
    let counter = 0

    // if k = 0, do nothing (tail is at k=1)
    if(k === 0){
```

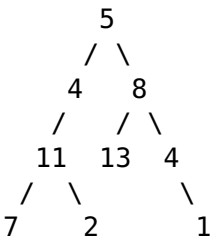
```
        return head
    }

    // loop until we get to the second to last node
    while(tailNode.next){

        // advance the tail by 1
        tailNode = tailNode.next

        // keep track of how many times we've advanced
        counter += 1

        // once the counter is greater than k
        if(counter > k){

            // we can start advancing the node that will be k steps behind the tail
            kNode = kNode.next

        // case: if k is greater than the length of the list, return the list
unchanged
        } else if(tailNode.next === null && counter < k-1){
            return head

        // case: if we're at the end of the list and would be moving the kNode on the
next move
        // this means we're removing the head, so set head to head.next
        } else if(tailNode.next === null) {
            return head.next
        }

    }

    // kNode.next is the node we need to remove
    // we need kNode's next value to be the value after the node that we're removing
    // 0 -> 0 -> kNode -> removedNode -> removedNode.next -> 0 -> tailNode
    let temp = kNode.next
    kNode.next = temp.next

    // return the head
    return head
}

// Time complexity: O(n) => we only loop through the list once
```

Task details: **tree_paths_sum**

Description:

Given the root of a binary tree where each node contains an integer, determine the sum of all of the integer values in the tree.

Example:



The expected output given the above tree is `5 + 4 + 8 + 11 + 13 + 4 + 7 + 2 + 1`, so your function should return `55`.

Solution (main.js):

```
//
// Binary trees are already defined with this interface:
// function Tree(x) {
//   this.value = x;
//   this.left = null;
//   this.right = null;
// }

function tree_paths_sum(root) {
    // init a stack, a sum, and a working node.
    // this will be a DFT; array methods in JS work similar to a stack
    let stack = []
    let sum = 0
    let node = root

    // as long as we have a node to work with, we need to keep adding
    while(node){

        // if there's a left subtree, add it to the stack
        if(node.left){
            stack.push(node.left)
        }

        // if there's a right subtree, add it to the stack
        if(node.right){
            stack.push(node.right)
        }

        // add the current node's value to the sum
        // we won't be returning to this node
        sum += node.value
```

```
        // set the working node to the last item that was added to the stack
        node = stack.pop()
    }

    // return the total
    return sum

}

// Time complexity: O(n) => we'll never deal with the same node more than once.
```

Task details: **merge_packages**

Description:

Given a package with a weight limit `limit` and an array of integers `items` of where each integer represents the weight of an item, implement a function `merge_packages` that finds the first two items in the `items` array whose sum of weights equals the given weight limit `limit`.

Your function should return a pair `[i, j]` of the indices of the item weights, ordered such that `i > j`. If such a pair doesn't exist, return an empty array.

Examples:

```
Input: items = [4, 6, 10, 15, 16], limit = 21
Output: [3, 1]
Explanation: The weight of the items at indices 3 and 1 sum up to the specified limit.
```

Solution (main.js):

```
function merge_packages(items, limit) {

    // use a hashtable for O(1) lookup
    let values = {}

    // we need the index and need to be able to break out of the loop once we find the
pair
    // use a for loop for this
    for(let i = 0; i < items.length; i++){

        // we're looking for two items that add up to the limit
        // using the difference will let us use the hashtable for quick lookup
        let pairWeight = limit - items[i]

        // if we've already encountered an item that will add up to the limit with the
current item
        // can't just check for truthiness; the item at the 0th index would return
false
        if(values[pairWeight] || values[pairWeight] === 0){

            // return the current index and the index stored in the dictionary under
the pair weight
            return [i, values[pairWeight]]

        } else {

            // add the current value to the dictionary; key is the number, value is
its index to be used for the return
            values[items[i]] = i
        }
    }

    // if there's an answer, this function will return before getting here
```

```
    // if there's no answer, the tests are expecting an empty array.
    return []
}
// Time complexity: We only loop through the items once; lookup in a hashtable is
O(1), so this is O(n)
```