Ian Kaiserman

4/23/2020

CSCD 300

HW3 Progress

HW3 Progress

This contains detailed processes for the implemented methods in HW3

- Public MyLinkedList reverse(ListNode node)
    - Initially takes in the next of head when called by the implementation method reverse()
    - Checks if the passed in node is null
        - If true, return new MyLinkedList (this means list was empty, or gets called when the recursion reaches the end to create a new list)
    - RECURSIVE STEP: Create new MyLinkedList object newList, which is assigned this method calling itself passing in the next of node until the list reaches the end
        - When it hits the end, the if statement fires and creates an empty list, and the next processes below fire starting from the end of the list moving backwards
    - Have newList call addLast (HELPER METHOD), adding each data piece of recursive node to the end of the list, but backwards, causing the new list to be a reversed version of the original
    - Return newList
- Public void addLast(Object data) // HELPER METHOD
    - Checks if the list is empty

        - If it is, call addFirst instead, passing in data
    - Else, create ListNode reference cur assigned to the next of head
        - While loop while the next of cur does not equal null
            - Reassign cur to next of cur (iterate to end)
        - Reassign next of cur to a new ListNode passing in data
        - Increase size by 1
- Public ListNode reverse(ListNode first, ListNode second)
    - Check if second is null
        - If true, return first (this will return what needs to be the new first data after the recursion)
    - Else, (RECURSIVE STEP) create ListNode reference h assigned to the result of calling this method passing in the next of first and the next of second. This iterates through the list until it eventually returns the last node in the list, which will be assigned as the new first data element
        - Reassign next of second to first. This will reverse the order of these two nodes every time the recursive step is going back to the beginning
        - Reassign next of first to null. This will be reassigned to the previous ListNode every time the recursive step runs its course, and prevents the first two ListNodes from looping each other when it gets to the beginning
        - Return h, as it is going to be set in reverse2() as the first data element in the list and ultimately create the reversed list
- Public int div(int m, int n)
    - Precondition checking if n is 0

        - If true, throw illegal argument exception, cannot divide by zero
    - Check if n is greater than m
        - If true, return zero, because n goes into m zero times, or zero more times if gotten here from the recursive step below
    - Else, return 1 + the result of calling div passing in m-n and n
        - * Effectively counting one multiple, taking it away from m, and testing again to see if n is smaller. When the recursion is done, the resulting returned number is how many n's could be taken from m (RECURSIVE STEP)
- Private boolean isSum24(int arr[], int targetSum)
    - If the length of arr is zero, return false, sum obviously cannot be 24
    - If the length is one and the one value equals the target sum, return true
        - This will be used for both initial arrays that contain just 24, and for the recursive steps shown later
    - Else, if the length is still one but the one value does NOT equal the target sum, return false
        - Again, this is used for an initial array and for recursive steps
    - Create a new int array called arrSmall that contains every value of arr except for the last one using a for loop
    - Return the boolean that results in recursively calling isSum24, passing in arrSmall, and targetSum after subtracting the last value in arr (RECURSIVE STEP)
        - Therefore, we do not include the last value of arr in arrSmall

        - Also, when it reaches the condition checking the size being 1, if the sum was indeed 24, that last remaining value should equal the integer resulting in subtracting each value of the array from the initial targetSum, in this case, 24.

- Public int countSpace(String str)
    - Return 0 if str is empty (used as a stopping case and a check for the initial string as well)
    - Create char variable current set to the char at index 0 of str (first character)
    - If current is equal to whitespace, return 1 + recursively calling countSpace passing in the substring of str cutting off index 0 (testing the rest of the string character by character) (RECURSIVE STEP 1)
    - Else, meaning current is some sort of actual character, return 0 + recursively calling countSpace the same way
        - This effectively adds nothing to the total amount of whitespace and a way to "iterate" through the string
    - The resulting number at the very end is the total amount of whitespace in the original string
- Public boolean myContains(String s1, String s2)
    - Preconditions: Check if either string input is null, or empty, or if s2 is shorter than s1, if any of these are true return false
    - If s2 starts with s1 (the substring) return true
    - Else, recursively call myContains passing in the smaller string s1 and s2 with the substring starting at index 1

        - This cuts off the first character in s2 and puts it back through the method so that startsWith() can be checked again, and repeats this to check every subsequent series of characters in the string, until s2 is shorter than s1 meaning it's impossible, or it returns true.

- Public void reverseArray(int a[], int low, int high)
    - Check if low is less than high (if not then this method is useless)
        - If true, create an int temp and assign it to the int at index low, assign the int at index low to the int at index high, and assign the int at index high to the int at index low (this swaps the indexes low and high
        - Recursively call the method again passing in a[] again, low increased by 1, and high decreased by 1
    - Each time this method goes through, it reverses by swapping the "inverse" positions of each integer, meaning the second to last and second to first switch for example, until low and high reach each other, at which point it stops at the middle
- Private void recursiveSelectionSort(int a[], int low, int high)
    - To create this, I used the typical selection sort algorithm we learned in class, modifying it slightly to turn the outer-most for loop into the recursive step
    - Create int min equal to the passed in index low
    - Create a for loop that creates a counting int search equal to low + 1, loops while search is equal to or smaller than passed in high, and search increases every time it loops
        - Check if the data at index search is less than the data at index min. if so, reassign the index min to the index search

- Outside the for loop, min is now pointing to the index of the smallest unsorted value
- Create int temp assigned to the value at index low (low will be reassigned first)
- Reassign the value at index low to the value at index min (put min in the right spot)
- Reassign the value at index min to temp (effectively swap the data at min and low)
- Check if adding 1 to low is still less than or equal to high, meaning we have not hit the bounds of the array yet. If so, recursively call this method passing in the modified array a, low + 1 (same as incrementing “start” in the original algorithm), and high.