CS 120: Intro to Computer Programming II

In-Class Activity - 03 Linked Lists - Day 6

In this Activity, we'll we writing some algorithms which start to experiment with a recursive view of our linked lists.

Activity 1 - Turn in this one

Write a function, `count_v1(head)`, which takes a linked list as a parameter (remember, it might be empty!). Count up how many nodes it has, and return the count.

Once you have that completed, write a second function, which **does the exact same thing** - but does it a different way. In this function, `count_v2()`, you must handle the function locally if the list is empty. But if the list is **not** empty, you must call `count_v1()`, passing it the **second** node in the list. Save what it returns to you, and then return the proper value to your own caller. How should you modify the return value that you're given, so that you return the correct value?

Test out both of your functions. While you are not required to turn in correct answers to your ICAs, I want your group to work together to (try) to debug the function.

Once you've written the new function, discuss with the group: what happens with this function if the list has only one node? Does it work well, or not?

Turn in both functions, and also a record of your group discussion.

Activity 2 - Optional

NOTE:
I'm really hoping that all of the groups have time to work on this problem. If you do, go ahead and turn it in as normal. But if you run out of time, it's OK to skip this problem; just include a note to your TA that your group didn't get to it in time.

Write a function, `printVals_v1(head)`, which takes a linked list, and prints out its values, all in order. Use a loop.

Then, write a function, `printVals_v2(head)`, which does exactly the same thing. But this time - just like last time - your function will handle a little bit of the work on its own: if the list is empty, return immediately. Otherwise, call `printVals_v1()`, and pass it the **second** node of the list. What will your new function need to do, in order to print the whole list?

As before, test out your code. While it's not required that your answers be correct for an ICA, do your best!

If you are able to make `_v2()` work properly, then make one more change: instead of calling `_v1()` with the second node, call `_v2()` instead. Yes, you read that right: **make the function call itself!**

Now test your code. Does it still work? Discuss with your group why it does.

(activity continues on the next page)

Challenge Activity - Do not turn in this one

Write a function `spaceStr_v1(head)` which, given a linked list, returns a string, with the values separated by spaces. To make this easier, this function should not only have a single space between each value; it should also have one at the end. For example, the list

```
"abc" -> 123 -> "jkl"
```

must return the string

```
"abc 123 jkl "
```

(Remember that a list may hold things which are not string data; make sure to convert each value to a string.)

Once that's working, write a second version, `spaceStr_v2(head)`, which works the way the previous two exercises did: if the list is empty, handle it in the new function; otherwise, pass the **second** node of the list to `_v1()` - but still return the same value in the end.

As a third step, change `_v2()` so that it calls itself, instead of calling `_v1()`.

Challenge Activity - Do not turn in this one

Repeat the previous Challenge Activity, but this time, don't return a trailing space on the string; instead, in the previous example, you would return

```
"abc 123 jkl"
```

To make this work, start by figuring out what to return for an empty list. Then figure out what to return if the list has only one element. Finally, figure out what to return if the list has two elements (or maybe even three or four). Once you've done this, to a few steps, it should be possible to write a `while` loop, which will do this for a list of any length.

This is a classic example of what's called a "Fencepost problem". Read up on it here:
`https://en.wikipedia.org/wiki/Off-by-one_error#Fencepost_error`

A final challenge: can you build a recursive solution for this last algorithm? That is, a function which handles small lists on its own - but for longer lists, calls itself?