CSC148 Lab#8, summer 2016

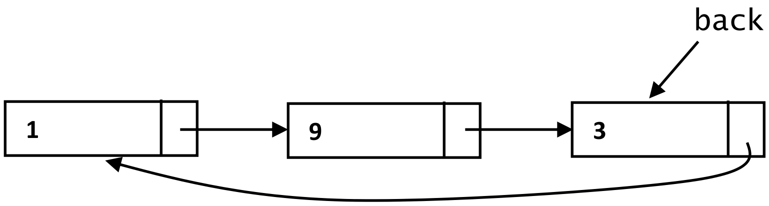
learning goals

In this lab you will review Test02 questions together with other topics you learned in this course.

setup

Download circular_linkedlist.py, tree.py, binary_tree.py, csc148_stack, and csc148_queue.py to a sub-directory called lab08.

Circularly Linked Lists



Example of a circular linked list

implement reverse_print1

Read the `docstring` of `reverse_print1(self, current)`, and implement it using recursion, without using stacks, queues, or Python lists.

Step 1: write an `if` statement to cover the base case, i.e. when there is only one node in the list—that is when `current` references to the same object that back (and `back.next_`) is referencing to.

Step 2: traverse the list *recursively* all the way to the end (i.e. where back is referencing to). This means that the recursive call should advance `current`. Note that the Python interpret (automatically) pushes the parameters (such as `current`) onto the stack of activation records. So, the last `current` pushed onto the stack is referencing to the same object `back` is referencing to, which means the last element in the list; the 2nd last `current` pushed onto the stack is referencing to the 2nd last element of the list; the 3rd last `current` pushed onto the stack is referencing to the 3rd last element of the list, … so if we print the value of `current` just before every return, the list is printed in the reverse order all the way to the first element current was referencing to.

implement reverse_print2

Read the `docstring` of `reverse_print2(self, current)`, and implement it without using recursion. Instead, let's do it explicitly ourselves what the Python Interpret did for us in `reverse_print1`.

Step 1: define a Stack.

Step 2. push `current` onto the stack.

Step 3. traverse the list *in a loop* all the way to the end (i.e. where `back` is referencing to). This means that you should advance `current` in each iteration and push it onto the stack.

Step 4. when Step 3 is done, the last `current` pushed onto the stack is referencing to the same object `back` is referencing to, which means the last element in the list; the 2nd last `current` pushed onto the stack is referencing to the 2nd last element of the list; the 3rd last `current` pushed onto the stack is referencing to the 3rd last element of the list. … so if we pop the `current` from the sack and print its value in a loop until stack is empty, the list is printed in the reverse order all the way to the first element current was referencing to.

implement reverse_print3

Read the `docstring` of `reverse_print3(self, current)`, and implement it with using minimal knowledge from CSC148!

Step 1: define a Python list, call it `easy_list`, containing the value of current.

Step 2: traverse the list *in a loop* all the way to the end (i.e. where `back` is referencing to). This means that you should advance `current` in each iteration and add its value to `easy_list`.

Step 3: reverse `easy_list.`

Step 4: print `easy_list.`

General Trees

implement is_full1

Read the `docstring` of `is_full1(t,n)`, and implement it using recursion, without using stacks, queues, or Python lists.

Step 1: write an `if` statement to cover the first base case you see in the `docstring`.

Step 2: write an `if` statement to cover the second base case (you can see it implicitly in the `docstring`).

Step 3: The recursive case invokes the `is_full` on all children of the current node if the current node has exactly n children. (recall an n-ary tree is full if all its non-leaf nodes have exactly n children). Use List comprehension to develop is_full1.

Hint: lookup `all` and `any` for Python lists (CSC108), it may help.

implement is_full2

Develop this very similar to `is_full1`, recursively. But do not use list comprehension.

implement is_full3

In last lecture, we discussed a tree traversal that was not recursive. It was level order traversal, using a queue. Implement `is_full3`, in a similar manner without using recursion.

Additional Exercises

implement max_value1 for BinaryTree

Read the `docstring` of `max_value(t,n)`, and implement it using recursion, without using stacks, queues, or Python lists.

implement max_value2 for BinaryTree

In last lecture, we discussed a tree traversal that was not recursive. It was level order traversal, using a queue. Implement `max_value2`, in a similar manner without using recursion.