CS 120: Intro to Computer Programming II

In-Class Activity - 13 Stacks and Queues - Day 3

Go back and get your code from the previous day's ICA. We will build off of it today.

Activity 1 - Turn in this one

Before we start writing a Queue-based solution for `shallow_word()`, let's look at how we can traverse a tree using a Queue. We are going to traverse the tree in a new order, something you haven't seen before: Breadth-First-Search.

Write a function which traverses a tree in the BFS style (I'll give you pseudocode below). Each time that you come to a new node, print out the value of that node, and **also** print out the value of all of the nodes currently in the queue, like this:

```
This node: foo
Queue    : bar baz fred wilma
```

To do this, write a function `print_BFS(root)`, which does the following:

```
Create a Queue, and place the root node into it

While the Queue is not empty:
    Dequeue one element
    Add all of its children (direct children only, not gradchildren) to the queue
    Print out the value of the node
    Print out the value of all of the nodes in the Queue
```

Build a tree (by hand!), and test it with this function. Can your group describe the pattern about how this function traverses the tree?

Solution:

```
def print_BFS(root):
    q = Queue()         # what is the name of your class?
    q.enqueue(root)

    while not q.is_empty():
        cur = q.dequeue()

        if cur.left is not None:
            q.enqueue(cur.left)
        if cur.right is not None:
            q.enqueue(cur.right)

        print(f"This node: {foo}")
```

```
        print( "Queue:    ", end="")
        for val in q:         # exactly how you do this loop depends on your Queue class
            print(f" {val}", end="")
        print()
```

Activity 2 - Turn in this one

Now you have the tools! Using what you've learned about Breadth-First-Search, rewrite `shallow_word()` using a Queue.

Solution:

```
def shallow_word(tree, letter):
    q = Queue()
    q.enqueue(tree)

    while not q.is_empty():
        cur = q.dequeue()

        if cur.val[0] == letter:
            return cur.val

        if cur.left is not None:
            q.enqueue(cur.left)
        if cur.right is not None:
            q.enqueue(cur.right)

    # no matching node found, anywhere
    return None
```

Activity 3 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Start by considering the worst-case runtime of both of our versions of `shallow_word()`. How long do they take to run, in big-Oh notation?

Hopefully, your group realized that they take the same amount of time (worst case). But in practice, the Queue-based version is often a lot faster. Why is this?

Solution: Both of the functions take $O(n)$ time to run, because (in the worst case) both of them end up doing a constant-time amount of work at every node in the tree.

However, the recursive function **has** to touch every node, every time. But the BFS version can **stop early** - as soo as it finds **any** node, that node is (by definition) the shallowest possible.