CSC148 Summer 2018: Midterm 2 Practice (Part 2)

This document contains some additional explanations for why we do things a certain way, as well as additional practice questions to help you prepare for Midterm 2. Take your time to read through this and work through the problems here.

(If you don't want to read the explanations, the practice problems start on Page 6.)

Midterm 2 Q&A

"Are the questions at the level of the exercise questions fair game for the midterm?"

No. You're given a week to do the exercises, whereas you have 50 minutes for the midterm. However, you should know the concepts of the exercises well and be able to understand why your solutions for them work.

Something like get_connecting_nodes() is a bit too hard for a 50 minute midterm, but something like get_path() would be fair (i.e. a method that returns a path from a node with a given value to the root node).

"Are lab questions fair game for the midterm?"

Yes.

"Are questions about the traversals fair game for the midterm?"

Yes. All of the traversals are fair game -- this includes level-order traversals, since the solution for this was covered in the lab. You should be able to understand how they work.

"Do we need to know what depth/height/arity/leaves/internal nodes/etc. is?"

Yes.

"When you say 'subtrees' does that mean of the root, or all subtrees in a tree?"

If I say 'subtrees of t' that means of the root. This is usually what I mean.

If I mean to say 'all subtrees within a tree (including subtrees of its subtrees)' I would clarify or say 'subtrees in t'.

"Will you ask us about arithmetic trees or classification trees?"

If I do, I'll re-introduce them. You don't need to memorize what they are.

Methods vs. Functions

This section contains an explanation for why we're using methods for Trees vs. functions for BinaryTrees.

When we're writing things like count_nodes() for Trees, I've asked you to write them as **methods** instead of functions. For example:

```
class Tree:
    ...

    def count_nodes(self) -> int:
        """
        Return the number of nodes in this Tree.

        >>> t = Tree(1)
        >>> t.count_nodes()
        1
        """
        return 1 + sum([child.count_nodes() for child in self.children])
```

As opposed to:

```
class Tree:
    ...

def count_nodes(t: Tree) -> int:
    """
    Return the number of nodes in this Tree.

    >>> t = Tree(1)
    >>> count_nodes(t)
    1
    """
    return 1 + sum([count_nodes(child) for child in t.children])
```

Which is the approach we use for BinaryTrees. Both of these work correctly for a Tree, but it's important to know the difference between a **method** and a **function** (the main difference being that methods are defined within a class, and have to take that class in as the parameter `self`, whereas functions are defined outside of the class, but can take in that class or others).

For BinaryTrees, we can't write the functions as methods, as our `left` and `right` subtrees can be `None`. For example, consider the code for count_nodes:

```
def count_nodes(t: Union[BinaryTree, None]) -> int:
    """
    Return the number of nodes in t.

    >>> t = BinaryTree(1)
    >>> cont_nodes(t)
    1
    """
    if t is None:
```

```
        return 0

    return 1 + count_nodes(t.left) + count_nodes(t.right)
```

The type annotation indicates that `t` is either a `BinaryTree` or `None`; whereas in the case of methods, the type of `self` must always be the type of our class. If we were to re-write this as a method, it would look like:

```
class BinaryTree:

    def count_nodes(self) -> int:
        """
        Return the number of nodes in this BinaryTree.

        >>> t = BinaryTree(1)
        >>> t.count_nodes()
        1
        """
        if self is None:
            return 0

        return 1 + self.left.count_nodes() + self.right.count_nodes()
```

Which has 2 issues:

1. `if self is None` won't ever happen since `self` must always be a BinaryTree.
2. `self.left.count_nodes()` and `self.right.count_nodes()` will result in an error if `self.left` is `None` or `self.right` is `None`. It's like trying to call:
   `None.count_nodes()`
   Which doesn't make any sense, since None has no methods.

For general Trees, we don't have to worry about this, since self.children is either an empty list (in which case we wouldn't iterate over anything) or a list of Trees. `None`s are never included in it. So we could either write Tree methods, or functions that take in Trees.

Since it's good practice to write methods and to understand the difference, we're writing **methods** for Trees, and **functions** for BinaryTrees.

Tracing Recursion

When we're trying to come up with recursive solutions, we have 2 main steps:

1. The base case: In this, we outline what should happen at the simplest case (i.e. a case where no recursion is needed).
2. The recursive step: In this, we make the assumption that our recursive calls will always work correctly. Then we use the results of our recursive calls as needed.

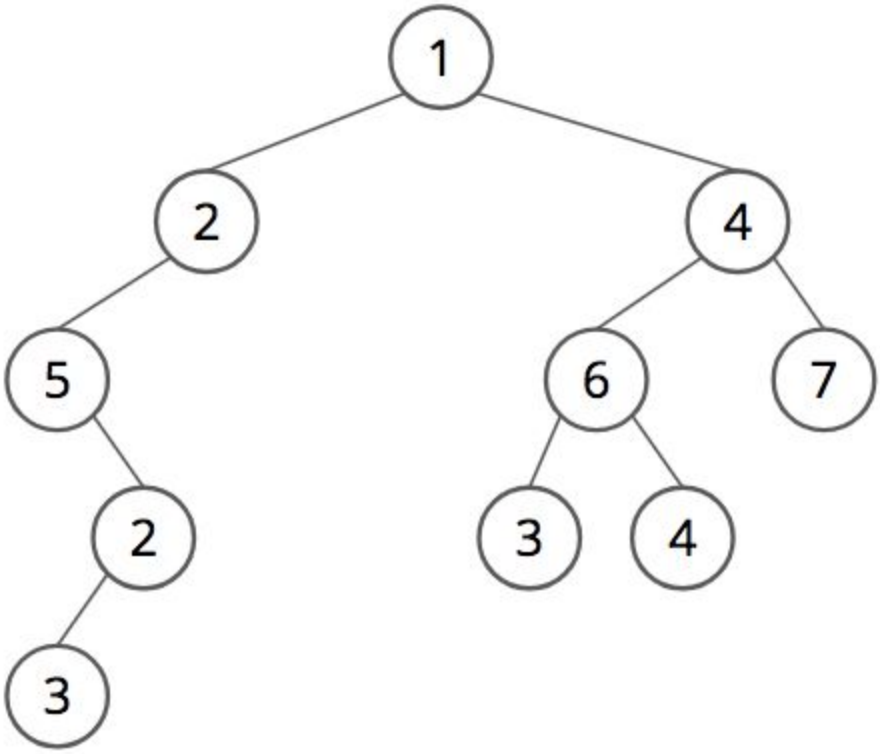
Step (2) requires a bit of faith, so feeling uncomfortable with "Why does this work?" is fairly common. As an example and an explanation, consider the code for the recursive BinaryTree function get_values():

```
def get_values(t: Union[BinaryTree, None]) -> list:
    """
    Return a list of all values in t in pre-order.

    >>> t = BinaryTree(1, BinaryTree(2))
    >>> get_values(t)
    [1, 2]
    """
    if t is None:
        return []

    return [t.value] + get_values(t.left) + get_values(t.right)
```

Suppose we have the following BinaryTree `t`:

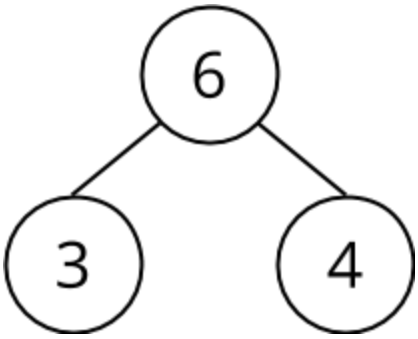


If we were to assume the recursive call on the subtree works properly, we intuitively know that we should get back `[2, 5, 2, 3]` for `get_values(t.left)` and `[4, 6, 3, 4, 7]` for `get_values(t.right)`. Given that, we know that we want to put those 2 lists together and put `t.value` at the front of that list and return it.

When coming up with this solution though, you might feel a bit uncomfortable assuming the recursive calls work properly. However, consider an empty BinaryTree (`None`). We know we'll always want to return `[]` when we call `get_values(None)`. Given that, suppose we have the following BinaryTree:

Its left and right children are both `None`. If we make recursive calls on that, we would expect empty lists to be returned. So, we know our recursive step makes sense: adding 3 to the front of an empty list is indeed what we want back. So for a leaf, we've proven our recursion works.

Given that, suppose we have the following BinaryTree `t`:



We proved that our recursive calls on leaves work correctly, so we know we'll definitely get back `[3]` and `[4]` from calling `get_values(t.left)` and `get_values(t.right)`. Thus, we know that we'll be able to return [6, 3, 4] if we just add 6 to the front of that list. So, for a BinaryTree whose left and right subtrees are leaves, we know our function works.

We can continue this pattern further: What if we have a BinaryTree whose subtrees are BinaryTrees whose subtrees are leaves? That should work too! We can complicate our example as much as we want, but within each subtree, we can always break it down into smaller and smaller subproblems, to the point where we reach our base case.

Practice Problems: Recursion with Nested Lists

1. Write a function called `count_strings_with_length()` that takes in either a string or a nested list and returns a dictionary that maps a length to the number of strings with that length. For example:
```
count_strings_with_length('cat') == {3: 1}
count_strings_with_length(['cat', ['a', 'dog'], [[['b'], 'no'], 'yes']])
== {3: 3, 1: 2, 2: 1}
```

2. Write a function called `return_values_with_type()` that takes in a parameter that could be anything (i.e. a nested list containing any value or any value itself) along with a type (e.g. `int`) and returns all of the values with that type. For example:
```
return_values_with_type(1, int) == [1]
return_values_with_type(1, str) == []
return_values_with_type([1, [[3], 'yes'], 'b', 2], int) == [1, 3, 2]
```

3. Write a function called `get_max()` that takes in either an int or a nested list and returns the largest int found. For example:
```
get_max(1) == 1
get_max([1, [[3], 0], 2]) == 3
```

4. Write a function called `get_min()` that takes in either an int or a nested list and returns the smallest int found. For example:
```
get_min(1) == 1
get_min([1, [[3], 0], 2]) == 0
```

5. Write a function called `all_true()` that takes in either an bool or a nested list and returns whether all booleans are True or not.
```
all_true(True) == True
all_true([]) == True
all_true([True, [[False, True], True], [True]]) == False
```

6. Write a function called `any_true()` that takes in either an bool or a nested list and returns whether there is at least one True or not.
```
all_true(True) == True
all_true([]) == False
all_true([True, [[False, True], True], [True]]) == True
```

Practice Problems: Trees

1. Write a method for a Tree containing int values called `level_sums()` that returns a list containing the sum of the integers at each depth. For example, index 0 of that list should be the sum of all values at depth == 0, index 1 should be the sum of all values at depth == 1, and so on.
   The level-order solution from the lab might be helpful.
   As practice with recursion: write a recursive solution to this problem.

2. Write a method for a Tree containing int values called `get_condition_passers()` that takes a function (which takes in an int and returns a boolean value) and returns all values in the Tree that pass the function in pre-order.

   a. Write this so that it returns it in post-order.

   b. Write this so that it returns it in level-order.

3. Write a method for a Tree that returns the values in the longest path possible from the

root to a leaf.

4. Write a method for a Tree that takes in a list of values and returns True if all items in that list are in the Tree.

5. Try writing solutions to Nested List problems but instead of a nested list, suppose we have Trees instead (i.e. a nested list of strings would turn into a Tree with string values).

Practice Problems: BinaryTrees

1. Write a function called `is_partitioned()` that takes in a BinaryTree with int values or None and returns True if all numbers on the left subtree are less than all numbers in the right subtree, with this holding for all subtrees as well.

   You'll want to write `get_max()` and `get_min()` for this (from Nested List problems #3 and #4).

2. Write a function called `could_be_a_linked_list()` that takes in a BinaryTree or None and returns whether we could make a LinkedList that matches the BinaryTree (i.e. the BinaryTree only contains nodes with 1 or 0 subtrees).

   a. Write a function called `turns_to_linked_list()` that takes in a BinaryTree that passes `could_be_a_linked_list()` and returns a LinkedListNode that represents the front of such a LinkedList.

3. Try writing solutions to Nested List problems but instead of a nested list, suppose we have Trees instead (i.e. a nested list of strings would turn into a Tree with string values).

4. Try writing solutions to Tree problems, but instead of a Tree, suppose we have a BinaryTree instead (i.e. instead of a list of subtrees, we just have left and right subtrees). Write these as functions instead of methods.

   For #2, write an in-order solution too.