Class 27 - Of Trees and Logs

Schedule

Everyone should either have (1) completed the Blue Belt level and be working on Project 5 (target completion by next Monday, 11 April), or (2) be making progress to completing the Blue Belt level. If you are stuck, not making progress, or unsure what you should be doing, make sure to check in with me today (in person, slack, or email) or at office hours tomorrow.

Notes and Questions

What is the *search problem*?

FindMatch Problem:

Inputs: `lst`, a list of N elements of type T; `ftest`, a test function $T \rightarrow$ Bool.
Output: an element, e, of `lst` such that `ftest(e)` is True, or `None` if no such element exists

What is the complexity of FindMatch?

```
def find_match(lst, ftest):
    """
    If there is any element e in lst that
    satisfies ftest(e) == True, returns a
    satisfying element e.

    Otherwise, returns None.
    """
```

What can we do if the asymptotic complexity of the problem we want to solve is too high?

Trees

Recall: A *List* is either (1) None, or (2) a pair whose second part is a *List*.

What is a *Tree*?

```
class Tree:
    """
    A Tree is a data structure where nodes have
    zero or more children, and one parent (except
    the root which has no parent).

    All the children and parents are Tree objects or None.
    """
    def __init__(self, value):
        self._parent = None
        self._children = []
        self._value = value

    def add_child(self, child):
        assert isinstance(child, Tree)
        self._children.append(child)
        child._parent = self

    def traverse(self):
        """
        Returns a list of all Tree objects in this tree.
        """
        result = [self]
        for subtree in self._children:
            if subtree:
                result += subtree.traverse()
            else:
                result += [None]
        return result
```

Binary Tree

```
from tree import Tree

class BinaryTree(Tree):
    """
    A tree where there are at most 2 children per node.
    """

    def __init__(self, value):
        super().__init__(value)
        self._children = [None, None]

    def add_child(self, child):
        raise RuntimeError("Add child not allowed!")

    def set_left_child(self, child):
        assert isinstance(child, BinaryTree)
        assert not self.left_child()
        self._children[0] = child

    def set_right_child(self, child):
        assert isinstance(child, BinaryTree)
        assert not self.right_child()
        self._children[1] = child

    def left_child(self):
        return self._children[0]

    def right_child(self):
        return self._children[1]

    def traverse(self):
        nodes = []
        if self.left_child():
            nodes += self.left_child().traverse()
        nodes.append(self)
        if self.right_child():
            nodes += self.right_child().traverse()
        return nodes
```

Ordered Binary Tree

```
from binarytree import BinaryTree

class OrderedBinaryTree(BinaryTree):
    """
    A tree where the ordered invariant is preserved:
        if left_node is left of current_node,
            fcompare(left_node.value, current_node.value)
        must be True
    """

    def __init__(self, value, fcompare = lambda a, b: a < b):
        super().__init__(value)
        self._fcompare = fcompare

    def add_child(self, child):
        raise RuntimeError("Cannot add children directly!")

    def add_node(self, node):
        """Inserts this child in a proper location."""
        assert isinstance(node, OrderedBinaryTree)

        if self._fcompare(node._value, self._value):
            if self.left_child():
                self.left_child().add_node(node)
            else:
                self.set_left_child(node)
        else:
            if self.right_child():
                self.right_child().add_node(node)
            else:
                self.set_right_child(node)
```

What is the complexity of searching for a node in an OrderedBinaryTree?

