Mod 9 - Solving 24 with Binary Expression Trees

Use Binary Expression Trees (BETs) to solve the game 24.

Background

BETs

We will use BETs, a kind of binary tree used to represent expressions, to solve this problem. In a BET, each internal node corresponds to an operator (e.g. `'+'` or `'-'`) and each leaf node corresponds to an operand. We typically use postfix notation to enter an expression into a BET - you might have encountered this in an old Casio calcluator:

```
           *
          / \
         1   +
            / \
           2   -
              / \
             3   4
postfix: 1, 2, 3, 4, -, +, *
expr:    (1*(2+(3-4)))
```

Note that we have several ways of visualizing this equation - the tree structure, the postfix expression which is commonly used to build these trees, and the expression we would write if we were solving this by hand.

24

In the game **24**, 4 cards are randomly picked from a standard deck of 52. Our goal is to get the value 24 by using each of these cards exactly once and some combination of addition, subtraction, multiplication, and division. We will consider the following to be values of face cards (cards 2-10 use their numbers as their values):

- A: 1
- J: 11
- Q: 12
- K: 13

For example, if the 4 cards chosen are "A 2 3 Q," our values are "1 2 3 12," and we can produce 24 as follows: $(3 - 2 + 1) * 12 = 24$

This would be modeled by the following BET:

```
               *
              / \
             +   Q
            / \
           -   A
          / \
         3   2
postfix: 3, 2, -, A, +, Q, *
expr:    (((3-2)+1)*12) = 24
```

We could get a different result by changing i) *the operators used* or ii) *the shape of the tree*:

```
         Different operators        Different shape
               -                          +
              / \                       /       \
             +   Q                     -         *
            / \                       / \       / \
           -   A                     3   2     A   Q
          / \
         3   2

postfix: 3, 2, -, A, +, Q, -        3, 2, -, A, Q, *, +
expr:    (((3-2)+1)-12) = -10       ((3-2) + (1*12))= 12
```

For a given 4-card combination with 4 possible operators (+, -, /, *), there are *many*, *many* trees we can create. We have 4! = 24 permutations of the 4 cards, 64 unique 3-operator combinations (`('+', '+', '+')`, `('+', '+', '-')`, `...`), and 5 valid tree shapes, for a total of 7680 possible trees. For the cards "A 2 3 Q", 33 of these evaluate to 24.

Part 1 - Make a `BET`

We will eschew a separate `BET` class, doing all of our work in `BETNode`. This means the user will have to manually keep track of the root node of any tree they don't want to lose.

Implement the following methods in your `BETNode` class:

- Each node in the binary tree has three instance variables:
  - `value` - a **string** representing a number, character, or mathematical operator (all of your values will be stings - (`'A', '2', '3', ..., '+', '-', ...`)
  - `left` - the left child
  - `right` - the right child
- `add_left(node)` - adds `node` as the left child
- `add_right(node)` - adds `node` as the right child
- `evaluate()`

  Recursively evaluate the subtree rooted at this `BETNode`.

  If the element is an operator (`'+'`, `'-'`, `'*'`, or `'/'`), call the evaluate method for the left and right child nodes, which will eventually return 2 values to apply that operator to.

  If the element is a card value, then simply return that value (this is the base case).

  ```
  >>> root.evaluate()
  24
  ```

  Note that you may end up dividing by 0 - you should handle this gracefully (don't terminate the program by raising an error).
- `repr()`

  Display the expression stored in the BET using infix notation (this is just how you would typically write an expression, see the example below). The logic here is simlar to that used for `evaluate`. Be sure to include parentheses if the element is an operator (i.e. is either '+', '-', '*', or'/'). Your results should look like:

```
>>> repr(root)
(A*(2*3*(4)))
```

We've included a test case for this method to help ensure you get the exact string formatting correctly.

There are a few ways to build BETs for testing. You could always manually build the tree, specifying which node should be a child of which:

```
root = BETNode('*')
root.add_left(BETNode('A'))
root.add_right(BETNode('*'))
root.right.add_left(BETNode('2'))
root.right.add_right(BETNode('*'))
root.right.right.add_left(BETNode('3'))
root.right.right.add_right(BETNode('4'))
```

This is fine, but can be a bit tedious. If you'd prefer, a more elegant solution is to write a function that takes a postfix expression and generates a BET from that. We're intentionally leaving the implementation details vague, but a stack will be useful if you decide to do this.

Write unittests for 2 different tree shapes. Your unittests should build the trees, then evaluate them, asserting that you get the expected result. Include Ascii art of the trees so it is clear what you are trying to build (this will also make it easier for you to write your tests). **Don't use any of the trees given in this assignment in these tests.** It's fine to use them to get started, but we are assessing your ability to synthesize your own tests.

2) Playing 24

We will use 2 functions to play the game - `create_trees(cards)`, which generates up to 7680 trees for a given assortment of 4-cards, and `find_solutions(cards)`, which scans through a collection of trees to find those that evaluate to 24.

2.1 `create_trees(cards)`

Return a **set** of every valid tree for a given collection of 4 cards. There are five valid shapes ('C' denotes a card and 'X' denotes an operator):

1. `CCXCCXX` e.g. `"A2*34-+"` -> $((A*2)+(3-4))$

```
      +
   /     \
  *       -
 / \     / \
A   2   3   4
```

2. `CCXCXCX` e.g. `"A2*3-4+"` -> $(((A*2)-3)+4)$

```
      +
     / \
    -   4
```

```
   / \
  *   3
 / \
A   2
```

3. `CCCXXCX` e.g. `"A23-*4+"` -> $((A * (2 - 3)) + 4)$

```
    +
   / \
  *   4
 / \
A   -
   / \
  2   3
```

4. `CCCXCXX` e.g.`"A23-4+*"` -> $(A * ((2 - 3) + 4))$

```
    *
   / \
  A   +
     / \
    -   4
   / \
  2   3
```

5. `CCCCXXX` e.g.`"A234+-*"` -> $(A * (2 - (3 + 4)))$

```
  *
 / \
A   -
   / \
  2   +
     / \
    3   4
```

For each shape, generate every possible permutation of operators. `itertools` is a helpful module here - feel free to use the following methods in this assignment:

- `itertools.permutations(collection)` - helpful for gnerating all unique permutations of a collection. By default, only creates permutations that use all objects in the collection, and does not repeat any objects.
- `itertools.sample(collection, k=1)` - helpful for generating all unique permutations of length `k` of a collection. This technique *does* generate permutations where an item is used more than once, which is useful for generating 3-operator possibilities from the provided collection of 5 operators (e.g. `(('+', '+', '+'), ('+', '+', '-'), ...)` when `k=3`).

Write a unittest to ensure you get the correct number of trees - 7680 if all cards are unique, and half of that if two of the cards are duplicates. Because you are returning a set, duplicates will only be added once.

2.2) `find_solutions(cards)`

Calls `create_trees(cards)` to get all valid trees for a passed in 4-card hand, then evaluates each tree and returns a **set** of all the ways to get 24. Store the string representation of each card in your set (e.g. `repr(root)` for reach `root` that is a valid solution)

Write a unittest to verify you get the correct number of solutions - at minimum, test a 4-card hand that gives you 0 solutions and one the `'A', '2', '3', 'Q'` hand described above that gives 33.

X.X) Extra flex

A few extra questions that are not for credit, but could be fun to figure out:

- What is the best hand you can find, and how many solutions does it produce?
- What percentage of hands have 0 solutions?
- What percentage of hands have exactly 1 solution?

Submission

At minimum, submit the `BET.py` and `TestBET.py`.

Students must submit **individually** by the due date (typically Tuesday at 11:59 PM EST).