4. (6 points) A Classy Election

Implement the `VotingMachine` and `Ballot` classes based on the doctest below. The voting machine must determine which choice has the most votes (the `winner`) and detect if a ballot is used more than once. In case of a tie, the `winner` among choices with maximal votes is the one that most recently received a vote. `Ballot.vote` takes a string, and a `VotingMachine` must handle an arbitrary number of choices.

```
class VotingMachine:
    """A machine that creates and records ballots.

    >>> machine = VotingMachine(4)
    >>> a, b, c, d = machine.ballots
    >>> d.vote('Bruin')
    'Bruin is winning'
    >>> b.vote('Bruin')
    'Bruin is winning'
    >>> c.vote('Bear')
    'Bear is losing'
    >>> a.vote('Bear')
    'Bear is winning'
    >>> c.vote('Tree')
    'Fraud: multiple votes from the same ballot!'
    >>> machine.winner
    'Bear'
    """
    def __init__(self, k):
        self.ballots =   [___________________________________ for i in range(k)]
        self.votes = {}

    def record(self, ballot, choice):
        if ballot.used:
            return 'Fraud: multiple votes from the same ballot!'


        -----------------------------------------------------------------------


        self.votes[choice] = ______________________________________________ + 1


        if _____________________________________________________________________:
            return choice + ' is losing'
        else:

            -------------------------------------------------------------------
            return choice + ' is winning'
class Ballot:

    ---------------------------------------------------------------------------

    def __init__(self, machine):
        self.machine = machine
    def vote(self, x):

        return ________________________________________________________________
```

3. (14 points) Will Code for Points

(a) (2 pt) Implement `objectify`, which takes a tree data abstraction and returns an equivalent `Tree` instance. Both the `Tree` class and the tree data abstraction appear on the midterm 2 study guide. *Warning: Do not violate the tree data abstraction! (Exams are flammable.)*

```
def objectify(t):
    """Return a Tree instance equivalent to a tree represented as a list.

    >>> m = tree(2)
    >>> m
    [2]
    >>> objectify(m)
    Tree(2)
    >>> r = tree(3, [tree(4, [tree(5), tree(6)]), tree(7, [tree(8)])])
    >>> r
    [3, [4, [5], [6]], [7, [8]]]
    >>> objectify(r)
    Tree(3, [Tree(4, [Tree(5), Tree(6)]), Tree(7, [Tree(8)])])
    """


    return ____________________________________________________
```

(b) (2 pt) Circle the Θ expression that describes the number of `Tree` instances constructed by calling `objectify` on a tree with n nodes.

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$

(c) (4 pt) Implement `closest`, which takes a `Tree` of numbers `t` and returns the smallest absolute difference anywhere in the tree between an entry and the sum of the entries of its branches. The `Tree` class appears on the midterm 2 study guide. The built-in `min` function takes a sequence and returns its minimum value. *Reminder*: A branch of a branch of a tree `t` is *not* considered to be a branch of `t`.

```
def closest(t):
    """Return the smallest difference between an entry and the sum of the
    entries of its branches.

    >>> t = Tree(8, [Tree(4), Tree(3)])
    >>> closest(t) # |8 - (4 + 3)| = 1
    1
    >>> closest(Tree(5, [t])) # Same minimum as t
    1
    >>> closest(Tree(10, [Tree(2), t])) # |10 - (2 + 8)| = 0
    0
    >>> closest(Tree(3)) # |3 - 0| = 3
    3
    >>> closest(Tree(8, [Tree(3, [Tree(1, [Tree(5)])])])) # |3 - 1| = 2
    2
    >>> sum([])
    0
    """
    diff = abs(_____________________________________________________)


    return min(_____________________________________________________)
```

(d) **(6 pt)** Implement `double_up`, which mutates a linked list by inserting elements so that each element is adjacent to an equal element. The `double_up` function inserts as few elements as possible and returns the number of insertions. The `Link` class appears on the midterm 2 study guide.

```
def double_up(s):
    """Mutate s by inserting elements so that each element is next to an equal.

    >>> s = Link(3, Link(4))
    >>> double_up(s) # Inserts 3 and 4
    2
    >>> s
    Link(3, Link(3, Link(4, Link(4))))
    >>> t = Link(3, Link(4, Link(4, Link(5))))
    >>> double_up(t) # Inserts 3 and 5
    2
    >>> t
    Link(3, Link(3, Link(4, Link(4, Link(5, Link(5))))))
    >>> u = Link(3, Link(4, Link(3)))
    >>> double_up(u) # Inserts 3, 4, and 3
    3
    >>> u
    Link(3, Link(3, Link(4, Link(4, Link(3, Link(3))))))
    """
    if s is Link.empty:

        return 0

    elif s.rest is Link.empty:

        _____________ = ____________________________________________________

        return _____________________________________________________________

    elif ______________________________________________________________________:

        return double_up(_____________________________________________________)

    else:

        _____________ = ____________________________________________________

        return _____________________________________________________________
```