Part 1: Multiple choice questions (25%)
These are in a separate quiz on Canvas.

Part 2: Hash tables (25%)

The class ***HashMap*** with is given with an implemented ***__init__*** function.
Two of the lines are commented out and students can choose between these two lines.
Uncomment the one you choose. ***You can not change*** __init__ ***in any other way***.
You can add helper functions, but must use either ***list*** or ***dict*** as buckets, as per the line you choose.
You do **not** need to implement resize/rebuild functionality.

```
def __init__(self):
    self.array_length = 16
```

Choose one of these:

```
  #  self.hash_table = [ [ ] for _ in range(self.array_length) ]
  #  self.hash_table = [ { } for _ in range(self.array_length) ]
    self.item_count = 0
```

Finish these implementations:

- def __setitem__(self, key, data)
    - Adds this data connected to this key
    - *overwrites/updates if already there*
    - 10%
- def __getitem__(self, key)
    - returns data for the key
    - *returns* **None** *if nothing there*
    - 10%
- 5% for the HashMap implementation in general

Part 3: Prefix parsing tree (25%)

Finish implementing the operation ***calculate_value*** into the class PrefixParsingTree.

- **(25%) *calculate_value()***
    - Returns an integer which is the calculated value of the ***root*** of the prefix parsing tree.
    - Each node has a token, stored in a string.
        - The token of a leaf is an integer.
        - The token of a non-leaf node is an operator, plus ("+") or minus ("-")
    - The calculated value of each non-leaf node in the tree is the result of applying the operator of that node to the calculated values of its two children.
    - The calculated value of a leaf is the integer value of the number represented in the token string.
- *Operations for building and populating the tree have already been implemented. Students only need to calculate the value*.

Part 4: BST (25%)

You are given an implementation of a Binary Search Tree with an insert function.

Implement the :

- **(15%) *remove_subtree(data)***
    - Finds and removes the node with **data** in it.
        - All nodes contained in that's nodes subtree are also removed
    - If no node in the tree contains **data**, do nothing.
- **(10%) *finish implementing remove_subtree so that the size of the tree is correctly modified when remove_subtree is used.***
    - Here you could count the amount of nodes in the subtree before finally deleting it