Code Guidance

Unless stated otherwise, **you are not allowed to use** built-in Python functions. Moreover, no other built-in data structures can be used apart from arrays and strings. In particular, you cannot use built-in list operations for appending an element to a list.

You can use substring/subarray-creating constructs like `A[lo:hi]`, operations for lexicographically comparing strings (e.g. `st1` < `st2` or `st` == `"foo"`), string/array indexing (e.g. `st[3]`), and taking the length of strings/arrays (e.g. `len(st)`).

You can solve the project using Java, if you prefer, though the guidance and hints are given in Python. The same restrictions as above apply on what you can use.

What to submit

You should submit a single ZIP file containing three files:

1. A **PDF file `report.pdf`** with a report, in which you should include:
   a) an explanation for each of the functions you implemented,
   b) your code in full.

   In order to be able to check submissions for plagiarism:

   - the report should be written electronically, i.e. no scans or pictures of hand-written reports will be accepted,
   - do not embed the code in the report as an image; rather, copy-and-paste it in your document as text.
2. Two **plain text files** containing your code for question 1 and question 2 respectively:
   - the file **question1.py** should contain the classes `BST2` and `BTNode2`
   - the file **question2.py** should contain the classes `WordTree` and `WTNode`

   We will mark your code automatically, so it is crucial that you make sure that it is correctly indented and without syntax errors.

   Do not include any module imports, etc.; include only your classes.

   Do not submit Python notebook files (i.e. ipynb), just PY files with your code in **plain text format**. If you use Jupyter Notebook as your IDE, the simplest way to save your code as a PY file is to go to File -> Download as -> Python (.py).

Avoid Plagiarism

Please make sure you follow these guidelines:

- This is an **individual assignment** and you should not work in teams.
- Note that showing your solutions to other students is an examination offence.
- You can use material from the web, so long as you **clearly reference your sources** in your report. However, what will get marked is your own contribution, so if you simply find code on the web and copy it you will most likely get no marks for it.
- More info: https://qmplus.qmul.ac.uk/mod/book/view.php?id=674515&chapterid=77910

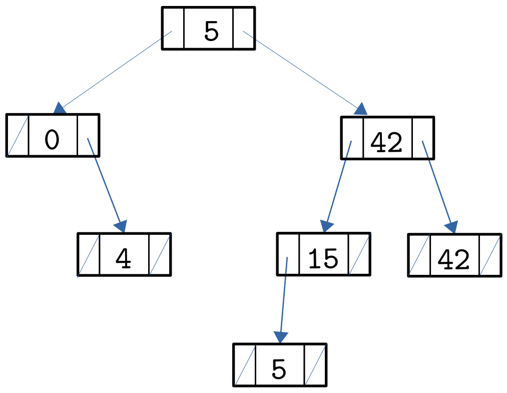
Question 1 **[50 marks]**

This question is about binary search trees. Binary search trees will be covered on week 9, and you can find on QM+ a Python notebook file `bst.ipynb` with their implementation.

The implementation we provide you uses tree nodes where each node contains some data, and pointers to the left and right child respectively of the node. For example, adding the numbers:

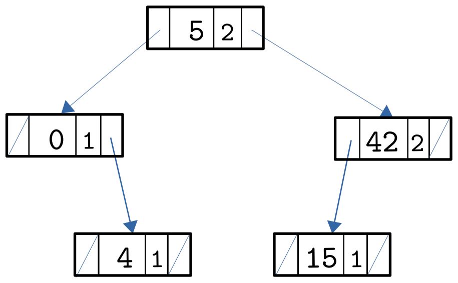
```
5, 42, 0, 15, 42, 4, 5
```

consecutively on an initially empty binary search tree we end up with the tree:



Note in particular that 5 and 42 require two nodes each in the tree.

You are asked to write an alternative implementation of binary search trees using tree nodes which also store the multiplicity of their data. That is, adding the same numbers as above to an initially empty tree would give us:



Note that in each node we have included the multiplicity of the integer that it contains (the numbers in smaller font).

Deliverable:

Write a class `BST2` that implements binary search trees where nodes contain a multiplicity number. Your class should contain the following functions:

- a constructor (`__init__`) -- this has been implemented for you
- one for checking if the tree is empty (`isEmpty`)
- one for adding a new element in the tree (`add`)
- one for counting the number of times that an element occurs in the tree (`count`)
- one for removing an element from the tree (`remove`)

We have made a start for you in the file `bst2.ipynb` by including the signatures of the functions (i.e. what their inputs and outputs should be).

Your implementation should use a class BTNode2 for tree nodes, which you also need to implement. For a start, we have already provided you the constructor of class BTNode2, and a function for converting a node and all its children into a string.

For example, executing the following code:

```
t = BST2()
t.add("cat"); t.add("car"); t.add("cav"); t.add("cat")
t.add("put"); t.add("cart"); t.add("cs")
print(t)
print(t.count("cat"),t.remove("cat"),t.count("cat"),t.size)
print(t)
print(t.count("cat"),t.remove("cat"),t.count("cat"),t.size)
print(t)
print(t.count("put"),t.remove("put"),t.count("put"),t.size)
print(t)
```

should produce the following printout:

```
(cat, 2) -> [(car, 1) -> [None, (cart, 1) -> [None, None]], (cav, 1) -> [None, (put, 1) ->
[(cs, 1) -> [None, None], None]]]
2 None 1 6
(cat, 1) -> [(car, 1) -> [None, (cart, 1) -> [None, None]], (cav, 1) -> [None, (put, 1) ->
[(cs, 1) -> [None, None], None]]]
1 None 0 5
(cav, 1) -> [(car, 1) -> [None, (cart, 1) -> [None, None]], (put, 1) -> [(cs, 1) -> [None,
None], None]]
1 None 0 4
(cav, 1) -> [(car, 1) -> [None, (cart, 1) -> [None, None]], (cs, 1) -> [None, None]]
```

Notes:

- Make sure your functions work in corner cases, such as when the tree is empty, when it remains empty after a remove operation, etc.
- Some marks are going to be reserved for space efficiency of remove. I.e. when removing an element of multiplicity 0, the whole node should be removed (instead of simply setting its multiplicity to 0).
- Do not change any str function that we provide you.
- Do not change the signatures (or names) of any of the functions provided as we are going to test your code automatically. E.g. you can implement remove in any way you want, but its header should be:

  ```
  def remove(self, d):
  ```

- Feel free to use helper functions in your classes if you like. It would be useful if you start the names of any helper functions with underscores:

  ```
  def _helper(self, foo, bar):
  ```

Question 2 **[50 marks]**

This question is about an extension of binary search trees that is useful for storing strings. We define a **word tree** to be a tree where:

- each node has three children: left, right and next;
- each node contains a character (the data of the node) and a non-negative integer (the multiplicity)
- the left child of a node, if it exists, contains a character that is smaller (alphabetically) than the character of the node
- the right child of a node, if it exists, contains a character that is greater than the character of the node

Thus, the use of left and right children follows the same lines as in binary search trees. On the other hand, the next child of a node stands for the next character in the string that we are storing, and its role is explained with the following example.

Suppose we start with an empty word tree and add in it the word `car`. We obtain the tree seen in the Figure 1, in which each node is represented by a box where:

- the `left`/ `right` pointers are at the left/right of the box
- the `next` pointer is at the bottom of the box
- the data and multiplicity of the node are depicted at the top of the box.

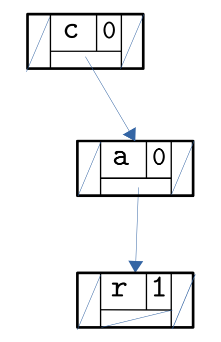


Fig. 1: tree containing: `car`.

For example, the root node is the one storing the character `c` and has multiplicity 0. Its left and right children are both `None`, while it next child is the node containing `a`.
So, each node stores one character of the string `car`, and points to the next node with the next character in this string using the `next` pointer. The node containing the last character of the string (i.e. `r`) has multiplicity 1. The other two nodes are intermediate nodes and have multiplicity 0 (e.g. if the node with `a` had multiplicity 1, then that would mean that the string `ca` were stored in the tree).

Suppose now want we add the string `cat` to the tree. This string shares characters with the first two nodes in the tree, so we reuse them. For character `t`, we need to create a new node under `a`. Since there is already a node there (the one containing `r`), we use the `left`/ `right` pointers and find a position for the new node as we would do in a binary tree. That is, the new node for `t` is placed on the right of `r`. Thus, our tree becomes as in Figure 2.

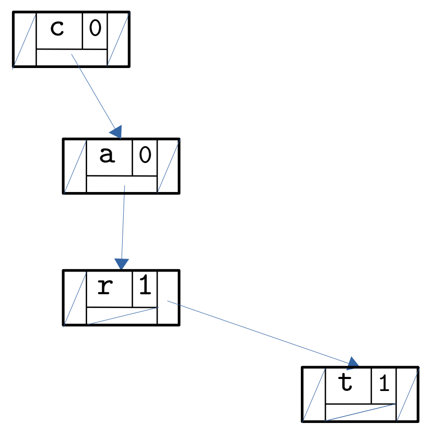
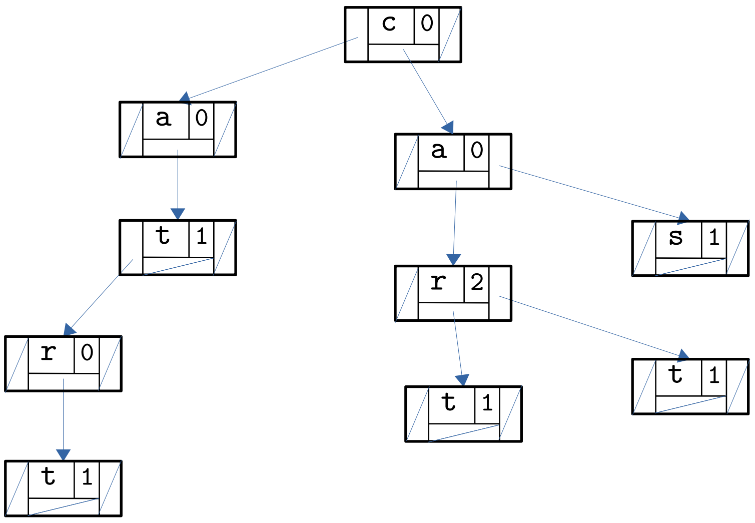


Fig. 2: tree containing: `car`, `cat`.

Observe that the multiplicities of the old nodes are not changed. In general, **each node in the tree represents a single string**, and its multiplicity represents the number of times which that string occurs in the tree. In addition, a node can be shared between strings that have a common substring (e.g. the two top nodes are shared between the strings `car` and `cat`).

We next add in the tree the string `at`. We can see that its first character is `a`, which is not contained in the tree, so we need to create a new node for it at the same level as `c`. Again, the position to place that node is determined using the binary search tree mechanism, so it goes to the left of `c`. We then also add a new node containing `t` just below the new node containing `a`.

We continue and add in the tree the strings: `art`, `cart`, `cs`, `car`, and our tree becomes:



Deliverable:

Write a class `WordTree` that implements word trees as above. Your class should contain the following functions:

- a constructor (`__init__`) -- this has been implemented for you
- one for checking if the tree is empty (`isEmpty`)
- one for adding a new string in the tree (`add`)
- one for counting the number of times that a string occurs in the tree (`count`)
- one for removing a string from the tree (`remove`)

We have made a start for you in the file `wordtree.ipynb` by including the signatures of the functions (i.e. what their inputs and outputs should be).

Your implementation should use a class `WTNode` for tree nodes, which you also need to implement. For a start, we have already provided you the constructor of the class `WTNode`.

Notes:

- All strings stored are non-empty. If `add`, `count` or `remove` are called with the empty string, then they should not change the tree and return `None`.
- Your solution for `remove` does not need to be space efficient, in the sense that if a node reaches multiplicity 0 after a removal, you can leave it on the tree (without losing marks).
- Do not change the signatures (or names) of any of the functions provided, and do not change any `str` function at all. Feel free to use helper functions.

For example, executing the following code:

```
t = WordTree()
t.add("cat"); t.add("car"); t.add("cat"); t.add("cat"); t.add("cat")
print(t.size,t.add(""),t.size,t.remove(""),t.size,t.count(""))
print(t)
print(t.count("ca"),t.count("car"),t.count("cat"),t.count("are"))
t.add("ca")
print(t.count("ca"),t.count("car"),t.count("cat"),t.count("are"))
t.remove("cat"); t.remove("car")
print(t.count("ca"),t.count("car"),t.count("cat"),t)
t.remove("cat"); t.remove("cat"); t.remove("cat")
print(t.count("ca"),t.count("car"),t.count("cat"),t)
t.remove("cat"); t.add("car")
print(t.count("ca"),t.count("car"),t.count("cat"),t)
```

Should produce the following output, apart from any differences in the tree printout due to `remove` not removing a node when its multiplicity is 0 (see second note above):

```
5 None 5 None 5 None
(c, 0) -> [None, (a, 0) -> [None, (t, 4) -> [(r, 1) -> [None, None, None], None, None],
None], None]
0 1 4 0
1 1 4 0
1 0 3 (c, 0) -> [None, (a, 1) -> [None, (t, 3) -> [None, None, None], None], None]
1 0 0 (c, 0) -> [None, (a, 1) -> [None, None, None], None]
1 1 0 (c, 0) -> [None, (a, 1) -> [None, (r, 1) -> [None, None, None], None], None]
```