ECS529U Algorithms and Data Structures **Coursework – part 2**

Deadline: Tuesday 10 December (week 12) at 12 noon

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)`).

What to submit

You should submit exactly **both** of the following files:

1. A **PDF file** **`report.pdf`** with a report, in which you should include:
   a) an explanation for the **`count`** function that you implemented (how does it work),
   b) your code for **`count`** 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. A **Jupyter file** called **`wordtree.ipynb`** containing your code:
   - the file 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, and can be imported without errors.

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

   Put all your code in one cell (i.e. a single "box" of code) with both classes in it.

Mark allocation

The report counts as 20 marks. You should explain what your code does, **not read it out line-by-line**. Line-by-line explanations typically indicate that the code is someone else's and not yours!

The code counts as 80 marks: count [20], add [30], minsrt [20], remove [10].

Avoid Plagiarism

Please make sure you follow these guidelines:

- This is an **individual assignment** and you should **not work in teams**.
- Showing your solutions to other students is also 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

Questions

This project 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 midl (which is short for “middle”);
- each node contains a character (the data of the node) and a non-negative integer (the multiplicity of the stored data)
- 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 midl 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 `midl` pointer is at the bottom of the box
- the data and multiplicity of the node are depicted at the top of the box.

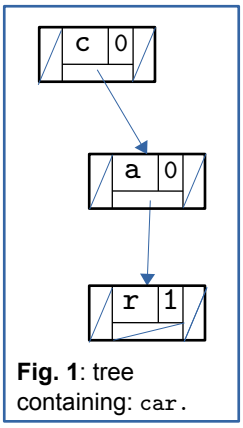


Fig. 1: tree containing: `car`.

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 midl child is the node containing `a`.

So, each node stores one character of the string `car`, and points to the node with the next character in this string using the `midl` 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.

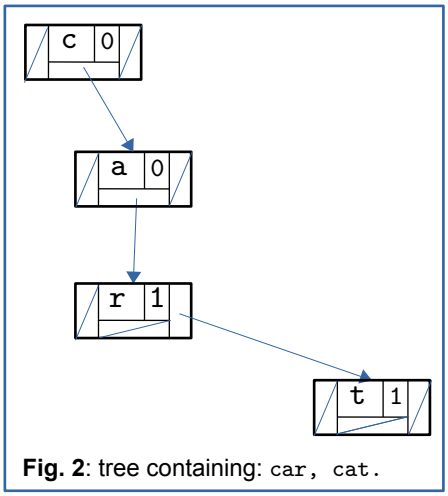
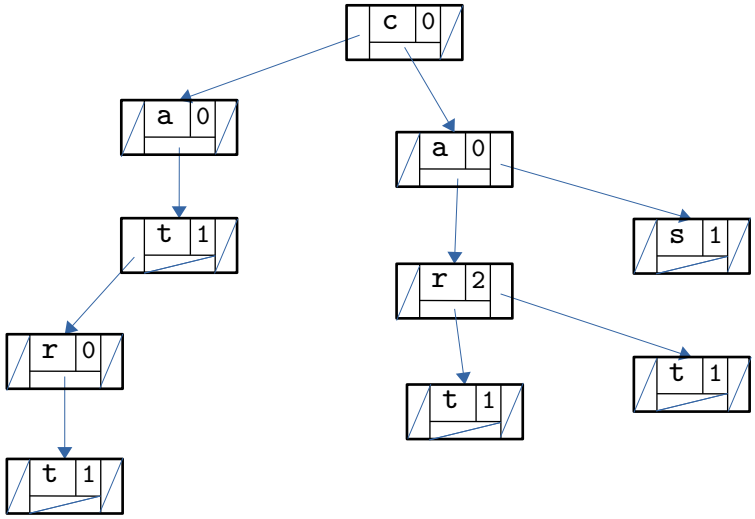


Fig. 2: tree containing: `car, cat`.

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 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
- `count`: for counting the number of times that a string is stored in the tree
- `add`: for adding a new string in the tree
- `minst`: for finding the lexicographically smallest string stored in the tree
- `remove`: for removing a string from the tree (note this is harder)

We 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, remove` or `minst` are called with the empty string, then they should not change the tree and return `None`.
- The solution for `remove` should remove a node if, after a removal, it reaches multiplicity 0 and does not have a midl child.
- 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:

```
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]
```