Questions

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

- each node has three children: left, right and forward;
- 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 forward 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 string tree and add in it the string `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 `fwd` pointer is at the bottom of the box
- the data and multiplicity of the node are depicted at the top of the box.

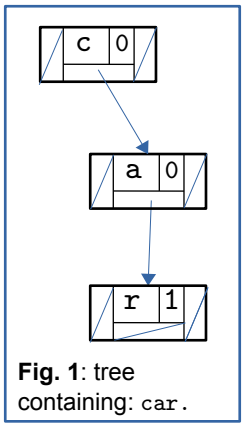


Fig. 1: tree containing: `car`.

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 its forward 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 `fwd` 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.

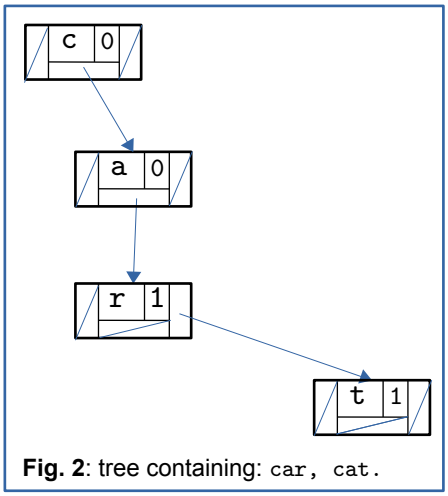
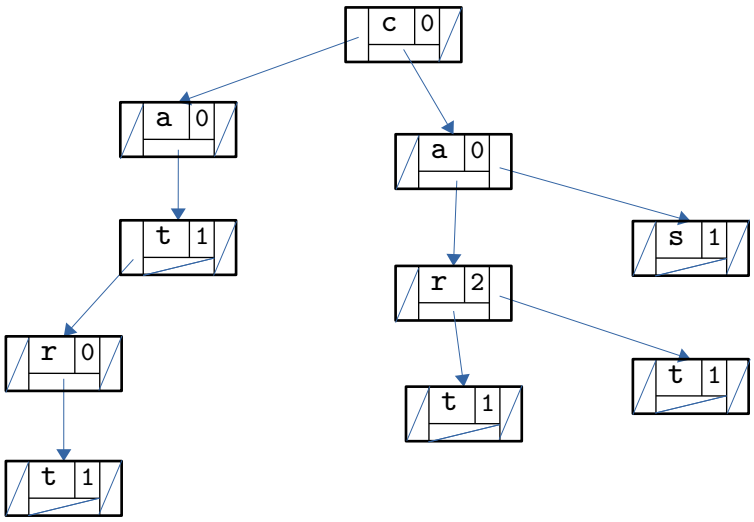


Fig. 2: tree containing: `car, cat`.

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:



Thus, our tree now contains the strings: `art, at, car, car, cart, cat, cs.`

Question 1 [25 marks]

We have provided a class `StringTree` that implements string trees as above. The class provided is very basic and only contains the following functions:

- a constructor (`__init__`) and a string function (`__str__`)
- `add:` adds a string in the tree, and `addAll:` adds an array of strings in the tree
- `printAll:` prints all the elements of the tree, one element per line

The implementation uses a class `STNode` for tree nodes, a basic implementation of which is also provided (consisting just of the constructor and string functions).

We have also provided you with a **personalised** list of 10 names, selected from the ONS list of baby names. For this question you are asked to:

- Draw the string tree that we obtain if we start from the empty string tree and consecutively add the 10 names from your list.
- Explain, in your own words, how the 10th name is added on your string tree.

In each case, you should use the algorithm explained above for adding strings. For the first part of the Question, you may want to use the provided code or/and do the 10 additions by hand.

Question 2 [75 marks]

Now expand the class `StringTree` adding the following functions:

- `count` : for counting the number of times that a string is stored in the tree
- `remove` : for removing a string from the tree (note this is harder)
- `max` : for finding the lexicographically largest string stored in the tree **(if your student number is odd)**
- `min` : for finding the lexicographically smallest string stored in the tree **(if your student number is even)**

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

Your implementation should use the class `STNode` for tree nodes, which you can also expand if you want.

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`. Moreover, if `min` or `max` are called on the empty tree, they should return `None`.
- The solution for `remove` should remove every node that, after a string removal, has multiplicity 0 and does not have a forward child. Node removal should follow the BST discipline, similarly to how it was presented in the lectures.
- 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:

```
def testprint(t,message):
    print("\n"+message,"tree is:",t)
    print("Count 'ca', 'can', 'car', 'cat', 'cats':",t.count("ca"),t.count("can"),
t.count("car"),t.count("cat"),t.count("cats"))
    print("Size is:",t.size,", min is:",t.min(),", max is:",t.max())
    t.printAll()


t = StringTree()
t.addAll(["car","can","cat","cat","cat"])
testprint(t,"Initially")
t.add("")
testprint(t,"After adding the empty string")
t.add("ca")
testprint(t,"After adding 'ca'")
t.remove("car")
testprint(t,"After removing 'car'")
t.remove("cat"); t.remove("cat");
testprint(t,"After removing 'cat' twice")
t.remove("ca"); t.add("cats")
testprint(t,"After removing 'ca' and adding 'cats'")
```

should produce the following output:

```
Initially tree is: (c, 0) -> [None, (a, 0) -> [None, (r, 1) -> [(n, 1) -> [None, None, None], None, (t, 3) -> [None,
None, None]], None], None]
Count 'ca', 'can', 'car', 'cat', 'cats': 0 1 1 3 0
Size is: 5 , min is: can , max is: cat
car
can
cat
cat
cat

After adding the empty string tree is: (c, 0) -> [None, (a, 0) -> [None, (r, 1) -> [(n, 1) -> [None, None, None], None,
(t, 3) -> [None, None, None]], None], None]
Count 'ca', 'can', 'car', 'cat', 'cats': 0 1 1 3 0
Size is: 5 , min is: can , max is: cat
car
can
cat
cat
cat

After adding 'ca' tree is: (c, 0) -> [None, (a, 1) -> [None, (r, 1) -> [(n, 1) -> [None, None, None], None, (t, 3) ->
[None, None, None]], None], None]
Count 'ca', 'can', 'car', 'cat', 'cats': 1 1 1 3 0
Size is: 6 , min is: ca , max is: cat
ca
car
can
cat
cat
cat

After removing 'car' tree is: (c, 0) -> [None, (a, 1) -> [None, (t, 3) -> [(n, 1) -> [None, None, None], None, None],
None], None]
Count 'ca', 'can', 'car', 'cat', 'cats': 1 1 0 3 0
Size is: 5 , min is: ca , max is: cat
ca
cat
cat
cat
can

After removing 'cat' twice tree is: (c, 0) -> [None, (a, 1) -> [None, (t, 1) -> [(n, 1) -> [None, None, None], None,
None], None], None]
Count 'ca', 'can', 'car', 'cat', 'cats': 1 1 0 1 0
Size is: 3 , min is: ca , max is: cat
ca
cat
can

After removing 'ca' and adding 'cats' tree is: (c, 0) -> [None, (a, 0) -> [None, (t, 1) -> [(n, 1) -> [None, None, None],
(s, 1) -> [None, None, None], None], None], None]
Count 'ca', 'can', 'car', 'cat', 'cats': 0 1 0 1 1
Size is: 3 , min is: can , max is: cats
cat
can
cats
```

If we then remove the remaining strings

```
t.remove("can"); t.remove("cats"); t.remove("cat")
print(t, t.size)
```

we should get the empty tree: `None 0`