ECS529U Algorithms and Data Structures **Mini-Project**

Deadline: Friday 9 December (week 11) at 10am

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 and characters (e.g. `st1 < st2` or `st == ''foo''`), string concatenation (e.g. `st1+st2`), 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) your answer to Question 1 (a tree and an explanation of one addition to it)
   b) your code in full for Question 2.

   In order to be able to check submissions for plagiarism:
   - the report should be written electronically, apart from the drawing in Question 1, which you can draw on paper and embed as an image in your document
   - 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 **`bigtree.ipynb`** containing your code:
   - the file should contain the classes `BigTree` and `BgNode`

   We will mark your code automatically, so make sure it is correctly indented and without syntax errors, and that it can run without errors !!!

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

   Do not include any testing code (this is typically a source of errors).

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

Mark allocation

Achieving 10 marks in Question 1 (i.e. a pass, 40%) is a **prerequisite** for getting any marks in Question 2.

The 75 marks of Question 2 are allocated as: count [30], toInc/DecArray [25], remove [20].

Avoid Plagiarism

- **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.

Questions

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

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

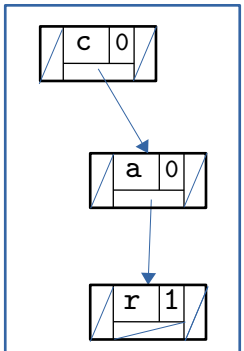


Fig. 1: tree containing: `car`.

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 mid child is the node containing `q`.
So, each node stores one character of the string `car`, and points to the node with the next character in this string using the `mid` 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.

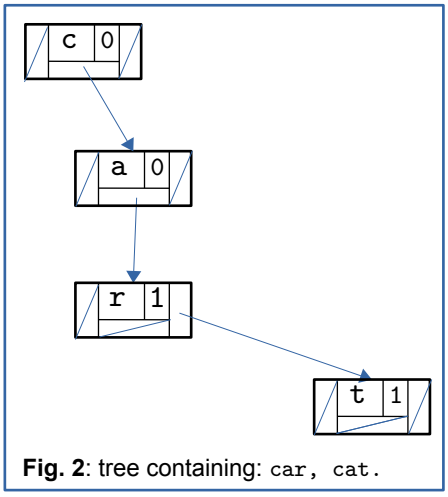
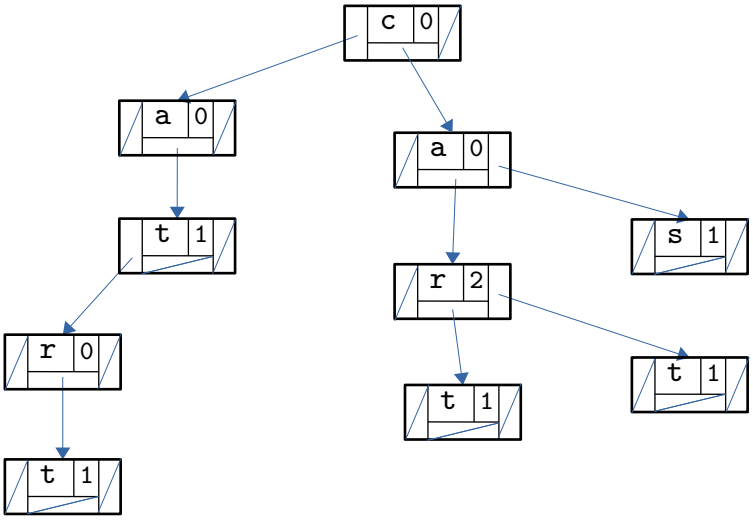


Fig. 2: tree containing: `car, cat`.

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 implemented a class `BigTree` that implements big trees as above. The class provided is very basic and only contains the following functions:

- a constructor (`__init__`) and a string function (`__str__`)
- `add, addAll` : adds a string in the tree, adds an array of strings in the tree
- `printAll` : prints all the elements of the tree, with spaces in between

The implementation uses a class `BgNode` 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 11 names, selected from the ONS list of baby names. For this question you are asked to:

- Draw the big tree that we obtain if we start from the empty big tree and consecutively add the 11 names from your list.
- Explain, in your own words, how the 11th name is added on your big 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 11 additions by hand.

Question 2 [75 marks]

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

- `count` : for counting the number of times that a string is stored in the tree.
- `toIncArray` : for returning an array with all the elements of the tree in increasing order **(if your student number is odd)**
- `toDecArray` : for returning an array with all the elements of the tree in decreasing order **(if your student number is even)**
- `remove` : for removing a string from the tree. The function should remove every node that, after a string removal, has multiplicity 0 and does not have a mid child. Node removal should follow the BST discipline, similarly to how it was presented in the lectures (note this is harder).

We made a start for you in the file `bigtree.ipynb` by including the headers of the functions (and some guidance).

Your implementation should use the class `BgNode` for tree nodes, which you can also expand if you want. **Do not change any of the functions that we have already implemented.** Feel free to use helper functions.

For example, executing the following code should produce the printout on the next page:

```
def testprint(t,message,file):
    print("\n"+message,"tree is:\n",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,", elements: ",end="")
    t.printAll()
    print("Increasing:",t.toIncArray())
    print("Decreasing:",t.toDecArray())

t = BigTree()
t.addAll(["car","can","cat","cat","cat"])
testprint(t,"Initially","0")

t.add("")
testprint(t,"After adding the empty string","1")

t.add("ca")
testprint(t,"After adding 'ca'","2")

t.remove("car")
testprint(t,"After removing 'car'","3")

t.remove("cat"); t.remove("cat");
testprint(t,"After removing 'cat' twice","4")

t.remove("ca"); t.add("cats")
testprint(t,"After removing 'ca' and adding 'cats'","5")

t.remove("can"); t.remove("cats"); t.remove("cat")
testprint(t,"After removing 'can', 'cats' and 'cat'","6")
```

```
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 , elements: can car cat cat cat
Increasing: ['can', 'car', 'cat', 'cat', 'cat']
Decreasing: ['cat', 'cat', 'cat', 'car', 'can']

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 , elements: can car cat cat cat
Increasing: ['can', 'car', 'cat', 'cat', 'cat']
Decreasing: ['cat', 'cat', 'cat', 'car', 'can']

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 , elements: ca can car cat cat cat
Increasing: ['ca', 'can', 'car', 'cat', 'cat', 'cat']
Decreasing: ['cat', 'cat', 'cat', 'car', 'can', 'ca']

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 , elements: ca can cat cat cat
Increasing: ['ca', 'can', 'cat', 'cat', 'cat']
Decreasing: ['cat', 'cat', 'cat', 'can', 'ca']

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 , elements: ca can cat
Increasing: ['ca', 'can', 'cat']
Decreasing: ['cat', 'can', 'ca']

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 , elements: can cat cats
Increasing: ['can', 'cat', 'cats']
Decreasing: ['cats', 'cat', 'can']
1 (c, 0) -> [None, (a, 0) -> [None, (t, 1) -> [None, (s, 1) -> [None, None, None], None], None],
None]
2 (c, 0) -> [None, (a, 0) -> [None, (t, 1) -> [None, None, None], None], None]

After removing 'can', 'cats' and 'cat' tree is:
 None
Count 'ca', 'can', 'car', 'cat', 'cats': 0 0 0 0 0
Size is: 0 , elements:
Increasing: []
Decreasing: []
```

Important notes and guidelines (see also page 1):

- 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 implementations should be efficient and make use of the ordered tree data structure(s):
  - Solutions that find the element to count or remove by looking at all elements in the tree will get few or no marks.
  - For full marks, the sorting functions (`toIncArray` / `toDecArray`) should work in linear time. One way to achieve this is by doing a single depth-first search traversal of the tree:
    - for increasing order, you can do a left-first DFS (but not preorder),
    - for decreasing order, you can do a right-first DFS (but not preorder).

    An internal recursive function called on nodes of the tree could be used – we have included some skeleton code in the file in case it could help. There is no need to follow this idea so long as your solution works in linear time.
- You can use data structures that we saw in the modules, but only for auxiliary purposes and not for replacing the functions required by the big tree data structure.
- Your code is going to be automatically tested and marked. **In order to get any marks, you need to ensure that:**

  (a) you submit a separate code file and that your code runs without errors,

  (b) you do not change any of the functions already implemented.
- **Make sure you answer Question 1 correctly** in order to get marks for Question 2.