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 `mid`; and a `parent`
- 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 `parent` pointer always points to the parent node and is not depicted for economy; the root node's `parent` is set to `None`
- the data and multiplicity of the node are depicted at the top of the box.

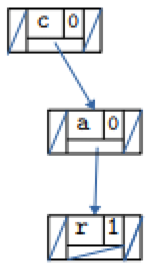


Fig. 1: tree containing: `car`.

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.

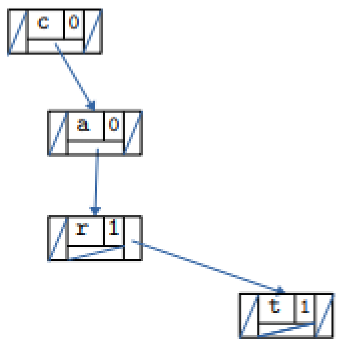
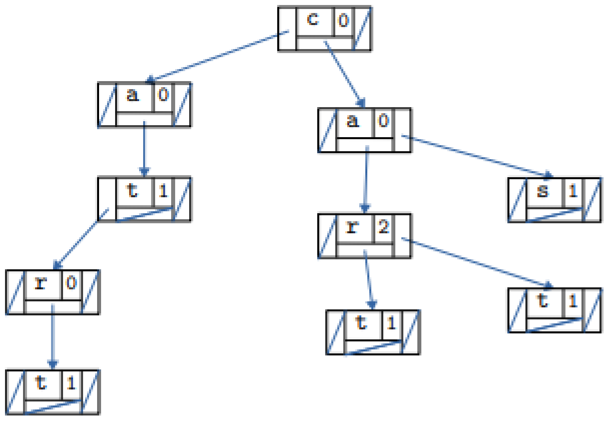


Fig. 2: tree containing: `car`, `cat`.

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.`