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

Deadline: Friday 8 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 Questions 2 and 3.

   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 **`stringtree.ipynb`** containing your code:
   - the file should contain the classes `StringTree`, `STNode`, `DNode` and `DLinkedList`.

   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 all classes in it.

Mark allocation

Achieving 10 marks in Question 1 (i.e. a pass, 40%) is a **prerequisite** for getting any marks in Questions 2 and 3. The test is marked out of 100, so there are 5 bonus marks.

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.
- No help from AI is allowed (e.g. do not use chatGPT).

Assignment Project Quiz Exam Essay Help
Wechat: cestbon 6888
Email: accoder overseas@163.com

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 `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.

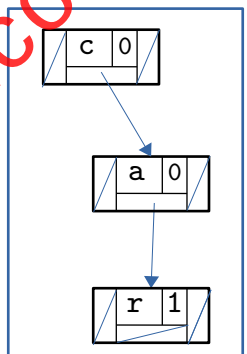


Fig. 1: tree containing: `car`.

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.

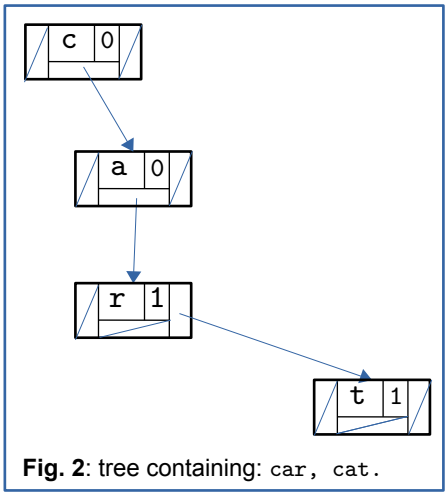


Fig. 2: tree containing: `car, cat`.

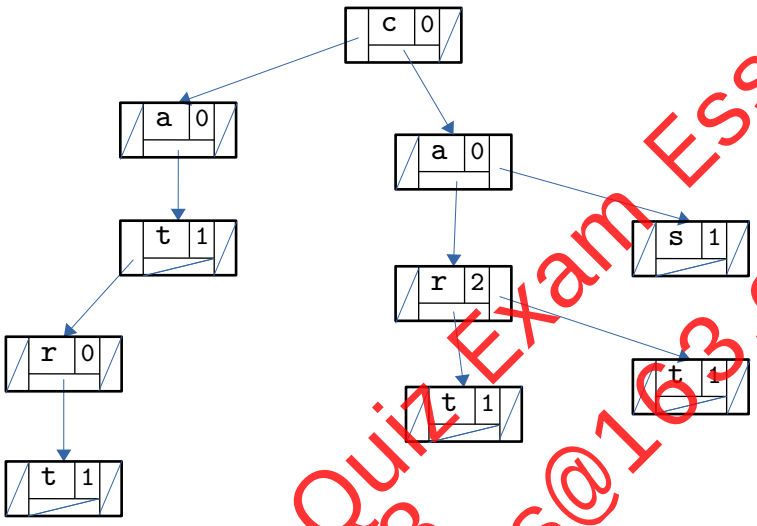
Observe that the multiplicities of the old nodes are not changed. In general, **each node in the tree represents a single string**, and its multiplicity represents the number of times that string occurs in the tree. In addition, a node can be shared between strings

Assignment Project Quiz Exam Essay Help
WeChat: cestbon 6688
Email: accoder overseas@163.com

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 `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, addAll`: adds a string in the tree, adds an array of strings in the tree
- a doubly linked list which is relevant for Question 3 only.

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).

You will also be provided 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.

Assignment Project Quiz Exam Essay Help
WeChat: cestbon_688
Email: acoder_overseas@163.com

Question 2

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

- `count(self, st)`: for counting the number of times that `st` is stored in the tree. [15 marks]
- `_max(self, node)`: that returns the node containing the largest (lexicographically) string in the tree that is within the subtree rooted at `node`. This is useful e.g. for implementing a function `max` that finds the largest string in the tree (see code).
  To help you in this part, we already implemented a function `_min(self, node)` that returns the node containing the smallest (lexicographically) string in the tree that is within the subtree rooted at `node`. [15 marks]

Question 3

We now expand the class `StringTree` by adding a doubly linked list containing the strings in the tree in increasing order. This is useful as it allows us e.g. to get a sorting of the elements in the tree in linear time.

We thus decide to keep a doubly linked list called `dlist`. The data in the nodes of `dlist` are unique strings (i.e. ignoring multiplicities) that are in the tree, ordered lexicographically. Furthermore, every tree node with non-zero multiplicity has a pointer to a node in the `dlist` which corresponds to the string associated with it. The instance variable for this pointer is called `dlistPtr`. For nodes with multiplicity zero, `dlistPtr` is equal to `None`.

We have added the instance variables `dlist` and `dlistPtr` to the respective classes. The function `updateDList(self, st)` is called in the `add` function and updates the `dlist` by adding `st` and returning a pointer to its node, if `st` is not already present in `dlist` (if `dlist` contains `st`, the function returns without changing the list).

You are asked to implement the following functions:

- `remove(self, st)`: for removing string `st` from the tree. The function should remove every node that, after the 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).
  Note that, in case `st` is completely removed from the tree (i.e. it had multiplicity 1) then `st` should also be removed from the `dlist`. [20 marks]
- `succ(self, st)` **(if your student id is even)**: that returns the string that is the lexicographic successor of `st` in the tree; i.e. the smallest (lexicographically) string in the tree that is larger than `st`. If `st` is larger than all elements in the tree, the function should return `None`.
  Note that `st` itself does not have to be a string in the tree. [25 marks]
- `pred(self, st)` **(if your student id is odd)**: that returns the string that is the lexicographic predecessor of `st` in the tree; i.e. the largest (lexicographically) string in the tree that is smaller than `st`. If `st` is smaller than all elements in the tree, the function should return `None`.
  Note that `st` itself does not have to be a string in the tree. [25 marks]
- `updateDList2(self, st, ptr)`: that provides a more efficient implementation of `updateDList` using the tree structure and runs in time O(m), where m is the length of the longest string that is in the tree. The current implementation of `updateDList` is inefficient and runs in time O(n) where n is the size of the tree (the number of strings stored in it).

Assignment Project Exam Help
WeChat: cestbon6688
Email: acoder_overseas@163.com

One way to go about this is to use the same logic as the `succ` or `pred` function that you have implemented and locate the successor/predecessor node in the tree of the one you want to add, and use its `dlistPtr` pointer to access the corresponding node in the `dlist`. [5 marks]

We made a start for you in the file `stringtree.ipynb` by including the headers of the functions (and some guidance). We also included some helper functions that could be of use.

Your implementation should use the provided classes `STNode,DNode,DLinkedList` for tree nodes, doubly-linked list nodes and doubly-linked lists respectively. You can expand these classes if you want but **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,step):
    print("\n"+step,":",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.printElems()
    print("Min and Max strings:",t.min(),t.max())
    print("Successors: '':",t.succ(""),", a:",t.succ("a"),",\
          ca:",t.succ("ca"),", can:",t.succ("can"),", cat:",t.succ("cat"))
    print("Predecessors: '':",t.pred(""),", can:",t.pred("can"),",\
          car:",t.pred("car"),", cat:",t.pred("cat"),", z:",t.pred("z"))

t = StringTree()
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")
```

Assignment Project Quiz Exam Essay Help
Wechat: cestbon 6888
Email: accoder_overseas@163.com

```
0 : Initially tree is:
 (c, 0) -> [□, (a, 0) -> [□, (r, 1) -> [(n, 1), □, (t, 3)], □], □]
Count 'ca', 'can', 'car', 'cat', 'cats': 0 1 1 3 0
Size is: 5 , elements: can, car, cat
Min and Max strings: can cat
Successors: '': can , a: can , ca: can , can: car , cat: None
Predecessors: '': None , can: None , car: can, cat: car, z: cat

1 : After adding the empty string tree is:
 (c, 0) -> [□, (a, 0) -> [□, (r, 1) -> [(n, 1), □, (t, 3)], □], □]
Count 'ca', 'can', 'car', 'cat', 'cats': 0 1 1 3 0
Size is: 5 , elements: can, car, cat
Min and Max strings: can cat
Successors: '': can , a: can , ca: can , can: car , cat: None
Predecessors: '': None , can: None , car: can, cat: car, z: cat

2 : After adding 'ca' tree is:
 (c, 0) -> [□, (a, 1) -> [□, (r, 1) -> [(n, 1), □, (t, 3)], □], □]
Count 'ca', 'can', 'car', 'cat', 'cats': 1 1 1 3 0
Size is: 6 , elements: ca, can, car, cat
Min and Max strings: ca cat
Successors: '': ca , a: ca , ca: can , can: car , cat: None
Predecessors: '': None , can: ca , car: can, cat: can, z: cat

3 : After removing 'car' tree is:
 (c, 0) -> [□, (a, 1) -> [□, (t, 3) -> [(n, 1), □, □], □], □]
Count 'ca', 'can', 'car', 'cat', 'cats': 1 1 0 3 0
Size is: 5 , elements: ca, can, cat
Min and Max strings: ca cat
Predecessors: '': None , can: ca , car: can, cat: can, z: cat

4 : After removing 'cat' twice tree is:
 (c, 0) -> [□, (a, 1) -> [□, (t, 1) -> [(n, 1), □, □], □], □]
Count 'ca', 'can', 'car', 'cat', 'cats': 1 1 0 1 0
Size is: 3 , elements: ca, can, cat
Min and Max strings: ca cat
Successors: '': ca , a: ca , ca: can , can: cat , cat: None
Predecessors: '': None , can: ca , car: can, cat: can, z: cat

5 : After removing 'ca' and adding 'cats' tree is:
 (c, 0) -> [□, (a, 0) -> [□, (t, 1) -> [(n, 1), (s, 1), □], □], □]
Count 'ca', 'can', 'car', 'cat', 'cats': 0 1 0 1 1
Size is: 3 , elements: can, cat, cats
Min and Max strings: can cats
Successors: '': can , a: can , ca: can , can: cat , cat: cats
Predecessors: '': None , can: None , car: can, cat: can, z: cats

6 : After removing 'can', 'cats' and 'cat' tree is:
empty
Count 'ca', 'can', 'car', 'cat', 'cats': 0 0 0 0 0
Size is: 0 , elements:
Min and Max strings: None None
Successors: '': None , a: None , ca: None , can: None , cat: None
Predecessors: '': None , can: None , car: None, cat: None, z: None
```

Assignment Project Quiz Exam Essay Help
WeChat: cestbon_6688
Email: accoder_overseas@163.com

Important notes and guidelines (see also page 1):

- All strings stored are non-empty. If `add`, `remove` or `pred` are called with the empty string, then they should not change the tree and return `None`. If `count` is called with the empty string, it should return 0. If `succ` is called with the empty string, it should return the (lexicographically) smallest string in the tree.
- 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 $\Theta(n)$-many elements, where n is the number of strings in the tree, or do linear search in a BST data structure, will get few or no marks. Similarly for linear-time solutions to finding the min/max in a subtree or the successor/predecessor of a string.
  - As explained, the `updateDList2` function should run in time $O(m)$ where m is the length of the longest string that is in the tree.
- 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 string 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 can be imported without errors,

  (b) you do not change any of the functions already implemented,

  (c) do not include any code you used for testing, nor any debugging messages.
- **Make sure you answer Question 1 correctly** in order to get marks for Question 2.

About lexicographic ordering. This is the order we find in a dictionary (that is what the name means). In other words, given two strings `s` and `t`, we have `s < t` when:

- either `s` is a prefix (i.e. an initial substring) of `t` and `s` is shorter than `t`,
- or `s` and `t` are of the form $xc_1\ldots$ and $xc_2\ldots$ respectively, with common prefix string `x`, and letter c_1 is before c_2 in the alphabet.

For example:

```
a < ab < ac < art < at < car < cart < cat < cs
```

Also, the empty string is the lexicographically smallest string.

Assignment Project Quiz Exam Essay Help
WeChat: cestbon 6888
Email: accoder.overseas@163.com