CSC263 – Problem Set 2

Remember to write your **full name** and **student number** prominently on your submission. To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Remember that you are required to submit your problem sets as both LaTeX `.tex` source files and `.pdf` files. There is a 10% penalty on the assignment for failing to submit both the `.tex` and `.pdf`.

Due Feb. 6, 2025, 22:00; required files: ps2.pdf, ps2.tex, ps2.py

Answer each question completely, always justifying your claims and reasoning. Your solution will be graded not only on correctness, but also on clarity. Answers that are technically correct but are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

You may work in groups of up to TWO to complete these questions.

1. **[6]** You are given a binary search tree and two values, **low** and **high**. Your task is to write a function that calculates the sum of all the values in the BST that fall within this range, inclusive of low and high. Assume that the BST does not contain duplicates. Your algorithm should efficiently traverse the BST, avoiding unnecessary visits to nodes outside the [low, high] range. Your algorithm must have $O(n)$ worst-case running time. Write your algorithm in pseudocode and justify why your algorithm is correct and why it runs in $O(n)$ time. Please also complete the programming question at the end.

   Below is the pseudocode for the algorithm:

   ```
   def range_sum(node, int low, int high):
           if node is NULL return 0
           if node.value < low return range_sum(node.right, low, high)
           else if node.value > high return range_sum(node.left, low, high)
           count = node.value
           count += range_sum(node.left, low, high) + range_sum(node.right, low, high)
           return count
   ```

   The algorithm above starts at the root node and compares its own value to the lower and upper range to determine which child to explore. For example, if the value at the root node is strictly less than the lower range, then the BST property tells us that every node in its left subtree will also be out of range, so there is no need to check it, and we can get away with checking only the right subtree. Likewise, if the value at the root node is strictly greater than the upper range, we only need to check the left subtree. Otherwise, when the value is in the middle, we should check both children as they can possibly contain values in the range we are querying for.

   Since each node is visited at most once, worst-case running time is $\mathcal{O}(n)$, where n is the number of nodes in the BST.

2. **[12]** Consider the following abstract data type that we will call a "`LandUse`."

   Objects: A GIS (geographical information systems) analyst is classifying the land use in Mississauga. The land use map is represented in the form of a pixel-based raster image. A set S of "land use pixels" that are represented by triples (x, y, u), where x and y are positive integers denoting a position of a pixel on the map, and $u \in \{'Agriculture', 'Residential', 'Recreational', 'Commercial', 'Industrial', 'Transportation'\}$ denotes a land use type being classified. Note that each position can be assigned up to one land use type, e.g., $(5, 3, 'Agriculture')$. In other words, $(5, 3, 'Agriculture')$ and $(5, 3, 'Industrial')$ cannot coexist.

   Operations:

   - `ReadType`(S, x, y): Return the land use type at position (x, y), i.e., the value of $u \mid (x, y, u) \in S$.
   - `WriteType`(S, x, y, u): Assign the land use type u to position (x, y), i.e., add the triple (x, y, u) to S. If position (x, y) already has a land use type u, then do nothing.
   - `NextInRow`(S, x, y): Return the position of the next classified pixel that appears after (x, y) and in the same row as (x, y), i.e., return $(x, \min\{\, y' \mid y' > y$ and $(x, y', u) \in S$ for some $u\,\})$. Return $(0, 0)$ if no such pixel exists. **Assumption on input values:** You can assume that a pixel at (x, y) **exists** in the `LandUse`.

- `NextInColumn`(S, x, y): Similar to `NextInRow`, return the position of the next classified pixel that appears after (x, y) and in the same column as (x, y). **Assumption on input values:** You can assume that a pixel at (x, y) **exists** in the `LandUse`.
- `RowEmpty`(S, x): Return whether Row x is empty, i.e., return `True` if and only if there does not exist a triple (x, y, u) with the given x in S.
- `ColumnEmpty`(S, y): Similar to `RowEmpty`, return whether Column y is empty.

Requirements: All above operations must have worst-case runtime $O(\log n)$, where n is the total number of pixels in the `LandUse` S.

Give a *detailed* description of how to use AVL trees to implement `LandUse`. In particular, answer the following questions.

(a) How many AVL trees are you using? What does each node correspond to? What information is stored in each node?

(b) What are the keys that you use for sorting each of the AVL trees? For each AVL tree, define **carefully and precisely** how you compare two pixels positioned at (x, y) and (x', y').

(c) For each of the above operations, describe in detail how it works, and argue why it works correctly and why its worst-case runtime is $O(\log n)$.

`Hint:` Try to make use of textbook algorithms for BST and AVL trees, and please do **not** repeat algorithms or runtime analyses from class or the textbook—just refer to known results as needed.

Solution.

(a): We can implement LandUse by using two augmented AVL trees, where each node corresponds to a unique land pixel. Both trees store the same nodes, but ordered differently. In both trees, each node stores three values, `node.x`, `node.y`, and `node.u` and its left and right children. We consider the first tree "x-skewed" and the second tree "y-skewed".

(b): The way we compare the two nodes is different depending on the tree.

In general, we compare two nodes by comparing their x and y values, treating it as tuple. For example, in the x-skewed tree, $(x, y) > (x', y')$ if either $x > x'$ or $x = x'$ and $y > y'$, so we treat the x value with higher precedence than the y value. This is reversed in the y-skewed tree: we have $(x, y) > (x', y')$ if either $y > y'$ or $y = y'$ and $x > x'$.

(c): We will describe each operation below:

`ReadType`(S, x, y):

To find the desired pixel and return its land use type, we can use standard BST iteration on either tree, but as convention we will search the x-skewed AVL tree. The algorithm is exactly the same as a standard binary search, only the comparisions are done with tuples rather than scalar values. There are n nodes in the x-skewed tree, so performing a binary search will result in a worst-case runtime of $\mathcal{O}(\log n)$.

`WriteType`(S, x, y, u):

We insert the triple (x, y, u) into both the trees. We use the insertion algorithm for AVL trees on each tree. creating the node that stores the x, y, and land use type. Since this is all we are doing, we can guarantee that the worst-case runtime is $\mathcal{O}(\log n)$.

`NextInRow`(S, x, y):

For this operation, we will first find the successor of (x, y) by traversing the x-skewed AVL tree. We will call the position of the successor (x', y'). We know that $(x', y') \neq (x, y)$ since every pixel is unique. As well, we also know that there is no smaller pixel than (x', y') that is larger than (x, y). If $x' = x$, then (x', y') is indeed the pixel next in row, so we return (x', y'). Otherwise, we return $(0, 0)$ because there is no pixel that is in the same row as (x, y).

In terms of time complexity, all we do is find the successor of (x, y) and perform basic comparisons, so our worst-case runtime is $\mathcal{O}(\log n)$.

`NextInColumn`(S, x, y):

We perform the same algorithm as in `NextInRow`, only that we are traversing through the y-skewed AVL tree, and we are comparing y and y' rather than x and x'. Thus the worst-case runtime is also $\mathcal{O}(\log n)$.

`RowEmpty`(S, x):

We traverse the x-skewed tree, attempting to find any pixel in the xth row. However, we perform binary search while only comparing the x-value and ignore the y value. We return true once we encounter any pixel with their row being equal to x,

and return false if we arrive at a leaf node without seeing x once. Since the worst-case occurs when the travel to the bottom of the tree, and the height of an AVL tree is $\log n$, our worst-case runtime is $\mathcal{O}(\log n)$.

`ColumnEmpty`(S, y):

This operation is the exact same as `RowEmpty`, only we search the y-skewed tree and compare y values, so the worst-case runtime is also $\mathcal{O}(\log n)$.

Programming Question

The best way to learn a data structure or an algorithm is to code it up. In each problem set, we will have a programming exercise for which you will be asked to write some code and submit it. You may also be asked to include a write-up about your code in the PDF/TEXfile that you submit. Make sure to **maintain your academic integrity** carefully, and protect your own work. The code you submit will be checked for plagiarism. It is much better to take the hit on a lower mark than risking much worse consequences by committing an academic offence.

1. **(6 points)** Please implement the `range_sum_bst` function for Question 1 following the guidelines below. A starter code ps2.py is provided to you on Quercus.

   Requirements:

   - Your code must be written in Python 3, and the filename must be `ps2.py`.
   - We will grade only the `range_sum_bst` function; please do not change its signature in the starter code. Include as many helper functions as you wish.
   - All code that you submit must be your own, including any helper functions.
   - Your code must compile and otherwise be testable in order to earn credit for the auto-graded portion of this question.
   - Your coding components should be based on your written explanation and runtime analyses in Question 1.
   - For each test-case that your code is tested on, your code must run within 10x the time taken by our solution. Otherwise, your code will be considered to have timed out.