Assignment 4: Binary Search Tree - CS Airport

Date: March 4th 2019
Deadline: March 9th 2019 23:59

Objectives

So, you have just been hired as the IT specialist for a local airport completely run by computer scientists, that should be up and running next week. There is just one tiny problem: one of their senior developers got hired by another company, and he didn't finish the system to manage the schedule for airplane take-offs yet. You have been asked to implement a scheduling system that can handle some additional constraints, like the wait time between take-offs. The previous developer left a skeleton for the framework behind, that you will have to use. Everything has to be custom and object-oriented, to allow for maximum flexibility. Your bosses inform you that you are not allowed to use any libraries unless explicitly mentioned.

Airport scheduling

The Airport system will run on a milisecond based system, counting from 01-01-1970, where by default every 5 minutes an airplane is able to take-off (we do not bother with how viable this time limit would be in real-life, we are computer scientists after all). You learned from your co-worker that the minimum requirement for this Airport system is that an airplane with a specific planned time for take-off and a tailnumber (usually stored as : `time/tailnumber`) is rejected if another plane is already taking off within the given minimum wait time. For instance if we try to add an airplane to the schedule at time 600.000, it can only be added if there are no other airplanes between 300.000 and 900.000, as the wait time was set to 300.000 (5 minutes) by default.

The preferred solution to a conflict is to reschedule the airplane to the next available time after the initial planned time. For instance, if we currently only have one plane taking off at 800.000 and we want to add a plane at 600.000, the time at which it should be added would be 1.100.000 (800.000 + 300.000). You should start by implementing the basic option to run a simpler solution that will simply reject an entry if it does not meet the time-constraint and only once that is done focus on the preferred solution (the boss wants to test and see both options).

Binary Search Tree

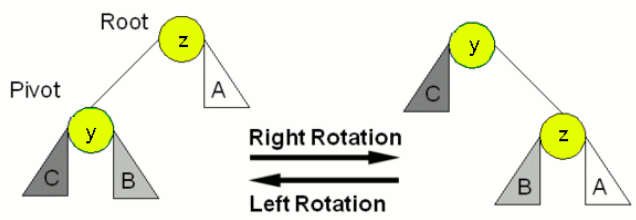
You sit in your new office, reviewing some old slides on `Binary Search Trees (BST)`, which you found among a stack of notes from your predecessor. The basic idea, you've gathered, seems to be first implement a BST to store the scheduling queue for the airport. However, a BST can get pretty unbalanced (where one branch can be significantly longer than the other), which would decrease the systems performance. So, they wanted to make a class that extends this BST and apply the AVL tree balancing algorithm, which will be called AVL. Using this class, the tree will be rebalanced after every insertion and deletion, ensuring the scheduling system can handle massive traffic, without have any of the operations slow down signficantly.

> ***Note***
>
> The slides suggest implementing all BST functions recursively, but this might not be the best approach. It is a nice compact way to explain the idea, but actually getting a recursive solution to work is more difficult than it looks. You'll need a lot of helper functions, updating the root node becomes tricky and debugging recursive functions is always harder. Most functions contain single recursive calls, and could easily also be written as `while` loop with a little more code.
>
> The only function where recursion really makes sense is `in_order_traversal`, which has 2 recursive calls and thus can't be translated to a straightforward loop. You may implement the other functions recursively too, but it is not recommended.

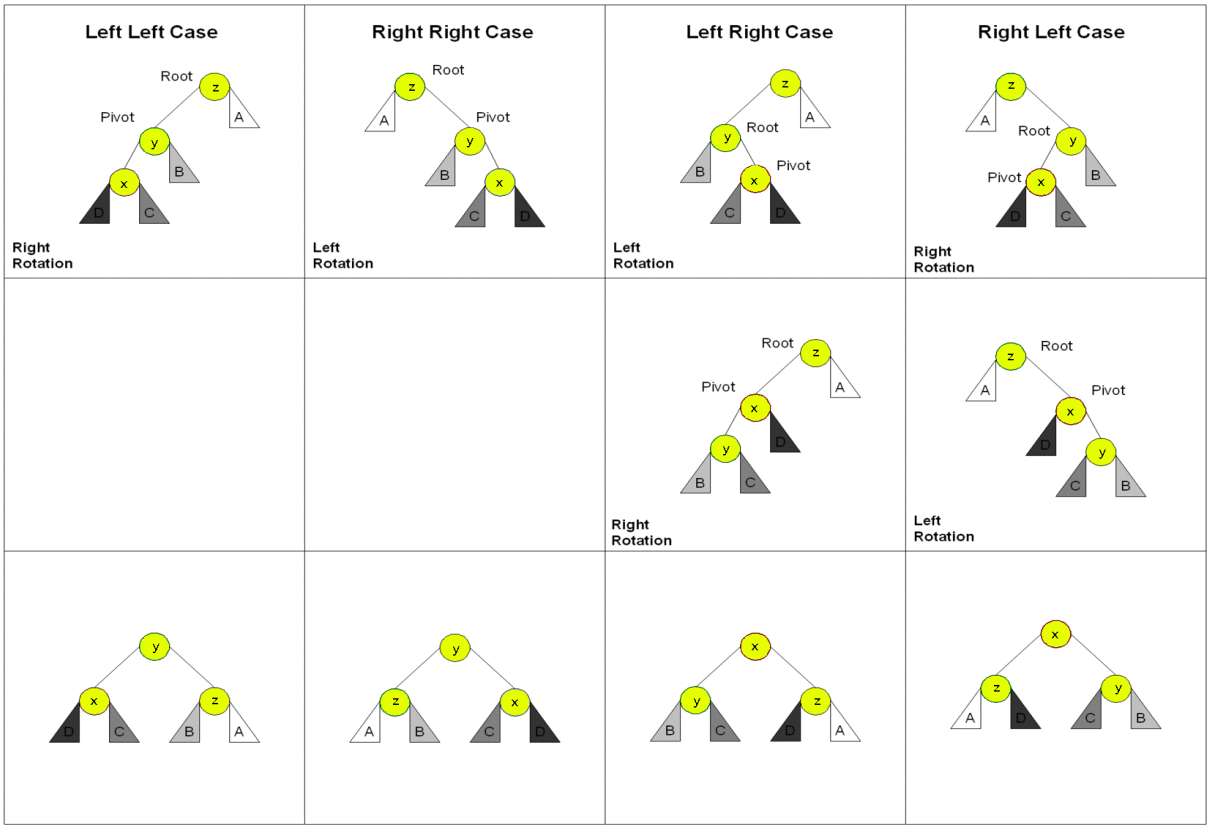
AVL Tree

The AVL tree rebalances by applying either a left rotation, right rotation, or a mix of the two, in order to fix a single imbalance. Imbalances can occur after any `insert` or `delete` function, which increase or decrease the height of subtree respectively. A node is considered unbalanced if the difference between the height of its children is greater than 1.

This can be resolved using the following 2 operations; if we consider two nodes with `y` and `z`, and their subtrees called `A`, `B` and `C`, we can change the relative height of the subtrees **and maintain the BST property**, as long as we swap some of their connections as well:



In the end you will have to implement 4 different cases, as described below. Each case is a sequence of rotations that should be read from top to bottom.



If we compute the `balance` for each of the nodes, we can define the 4 cases as follows:

1. `balance(z) < -1 and balance(y) < 0` : **Left Left**
2. `balance(z) > 1 and balance(y) > 0` : **Right Right**
3. `balance(z) < -1 and balance(y) > 0` : **Left Right**
4. `balance(z) > 1 and balance(y) < 0` : **Right Left**

In order to perform an AVL insert or delete, you just need to do the regular BST operation, travel up and find the first unbalanced node *z* and rebalance the subtree rooted at *z*. After rebalancing, travel further up to the next unbalanced node and repeat the balancing procedure until you reach the root of the tree. This

should take at most *height of the tree* number of rebalances, and will ensure the entire tree is rebalanced after an insert or delete operation.

> ***Note***
>
> The 2 main rotation functions are not a lot of code, but can be hard to get correct. Before you start writing, consider all the pointers you will have to change, as each node has 3 that might need to be updated; *parent*, *left* and *right*, and rotations can occur anywhere in the tree (not just at the *root*). Then carefully consider in what order you can update them while still reaching the ones you need to do, and write/draw this out.
>
> *Hint:* Make use of the existing BST functions to test this; you can do a sequence of simple inserts to build a tree, do 1 rotation and use `breadth_first_traversal` and `in_order_traversal` to check the results make sense.

Summary

You also know that once you have implemented the `BST`, `AVL`, and `Node` class properly, it will not be lot of work to extend them into the `Airport` class that will be required for the actual implementation of the airport system.

And thus, you look at the skeleton of the required files that the previous developer left behind, and that you have to complete:

- `Node`: The class that represents a node in the tree, which will contain the key and value of the node, and the links towards its parent and its children.
- `BST`: The class stores the root of the tree and handles all BST operations
- `AVL`: Inherits the `BST`, and overwrites insert and delete to perform the rebalancing.
- `Airport`: This class inherits most of its functionality from the `AVL`, but now before we add a node, we have to determine if the time is allowed according to time constraints of the airport.

Requirements

We ask you to implement these requested functions as specified. Note that you should not modify their function prototype definition in any way, as they will be tested automatically. You are free to add any other functions as long as they maintains the following requirements for each of the classes:

- Node (node.py)
    - Stores its parent node
    - Stores its children (order is important, and a maximum of 2)
    - Stores its key (by which it is indexed) and an optional additional value
    - Stores its current height in the tree
    - Can be easily compared to other Node objects or numbers using all comparison operators.
        - **Hint:** Overwrite the class comparator functions and use *isinstance* to determine the type of an object:

```
if node1 < node2:
  print("I can check against fellow Node objects!")
if node1 >= 20:
  print("I can check numbers too")
```

- Minimally has the following functions:
    - `get_key(self)`: Returns the key of the Node.
    - `get_value(self)`: Returns the value of the Node.
    - `get_parent(self)`: Return the parent object. None otherwise.
    - `get_left_child(self)`: Returns the left child object. None otherwise.
    - `get_right_child(self)`: Returns the right child object. None otherwise.
    - `get_height(self)`: Returns the height of the node (leaf-node is height 0).
- When the node is printed, it prints its key and value, or just its key if no value was stored.

- BST (bst.py)
    - Only stores Nodes with unique keys; duplicate keys cannot be inserted.
    - Stores the root node
    - Minimally has the following functions:
        - `__init__(self, key_list=[])`: Initialization function for the `BST` with an optional argument. If provided, all the keys in `key_list` are inserted into the tree in the order of the list.
        - `find_max(self)`: Finds the maximum node of the tree in `O(h)` operations.
        - `find_min(self)`: Finds the minimum node of the tree in `O(h)` operations.
        - `search(self, key)`: Returns the node with the specified `key` if it is in the tree, else return None.
        - `contains(self, key)`: Returns True if the specified `key` is in the tree, else return None.
        - `insert(self, key, value)`: Adds a new node with the provided `key` and `value` to the tree, and preserves the `BST` property. Returns the new node object if the node was succesfully insert, `None` otherwise.
        - `delete(self, key)`: Removes the node with the specified `key` from the tree, and preseves the `BST` property. Returns the node that was actually removed from the tree, which might be the successor of the original node containing `key`, or returns `None` if the key was not found.
        - `is_empty(self)`: returns True if the tree no nodes, else False
        - `in_order_traversal(self)`: Returns a list of the elements (Node objects) of the tree in sorted order. You should not use any built-in function like *.sort()* or *sorted()* (or any function that does something similar) for this.
        - `breadth_first_traversal(self)`: Returns a list of lists, where each inner lists contains the Node objects of one layer in the tree. Layers are filled in breadth-first-order. This traversal should contain all elements linked in the BST, including the *None* elements. So a leaf node had 2 *None* children, but a *None* node itself never has any children:

```
>>> BST([5, 8]).breadth_first_traversal()
[[Node(5)], [None, Node(8)], [None, None]]
```

    - When the `BST` is printed, it prints the elements of the tree in breadth-first order, layer by layer, printing `None` as _. After this, all the nodes are printed in sorted order on a single line.

      For example:

```
>>> print(BST([5, 8, 3, 12, 15, 1, 4, 21]))
5
3 8
1 4 _ 12
_ _ _ _ _ 15
_ 21
_ _
1 3 4 5 8 12 15 21
```

Note

The insert and delete operations will modify the height of several nodes in the tree, and you should ensure that the height of all nodes are correct at the end of each operation. You should **not** do this by recursively recomputing the height for *all* nodes in the tree, as this is an `O(N)` operation, meaning adding this step would change the complexity of `insert` and `delete` from `O(h)` to `O(N)`.

So, you should consider adding an operation that could update the heights of all the nodes after 1 insert or 1 delete in `O(h)` steps. Using this instead of recomputing *all* the heights will ensure your insert and delete operations remain efficient.

- AVL (avl.py)
    - Inherits all functionality of the `BST` class.
    - Overwrites the functionality of the `insert` and `delete` function to do rebalance according to the AVL description after the operation.
    - Minimally has the following additional functions
        - Static methods for `left_rotate(node)` and `right_rotate(node)` to perform the basic rotations of a node.
        - `fix_balance(self, node)` should fix the balance of a `node` and all nodes *above* it (i.e. parent nodes). This function also updates the root of the tree if needed.
- Airport (airport.py)
    - Inherits all functionality and of the `AVL` class
    - Accepts a `-t <wait_time>` in the command-line, indicating the minimum wait time that should be use for the scheduling.
    - Accepted a `-s` parameters in the command-line, if added makes the scheduler run "simple bounded insert" (reject insert if not allowed) instead of the default "replanning insert" (insert at next available spot if not allowed initially).
    - Minimally has the following additional functions
        - `find_conflict(self, time)`: Returns the first Node it finds that conflicts with the specified `time`, in combination with the `wait_time` set for the Airport. Returns `None` if no such conflict is found.
        - `bounded_insert(self, time, tailnumber)`: Inserts the airplane at the given time (or not), based on the policy that has been set at the creation of the Airport. Return the inserted Node if successful, None otherwise.

- When the airport is printed it only shows the airplanes in-order (e.g. "5000/BLA 10000/OOP 15000/TRO 20000/STO")

Input and output format

Your program should schedule all the airplanes in the order in which they are presented on the command-line and each airplane should be of the format *time/tailnumber*. Example usages of the final scheduling program:

```
$ python3 airport.py 37/BLZ 256/TAA 87/OOP 123/TRX 73/STO 11/CRO 252/WRQ -t 10 -s
11/CRO 37/BLZ 73/STO 87/OOP 123/TRX 256/TAA

$ python3 airport.py 37/BLZ 256/TAA 87/OOP 123/TRX 73/STO 11/CRO 252/WRQ -t 50 -s
37/BLZ 87/OOP 256/TAA

$ python3 airport.py 37/BLZ 256/TAA 87/OOP 123/TRX 73/STO 11/CRO 252/WRQ -t 50
37/BLZ 87/OOP 137/TRX 187/STO 256/TAA 306/CRO 356/WRQ
```

Getting started

You should start by writing the `Node` class, then `BST`, then `AVL` class and finally the `Airport` class. Basic tests have been provided for each of these classes, but remember that they might not be complete. For example, to run the basic tests on your `Node` implementation:

```
$ python3 check_node.py
```

Note: If you get stuck on the `AVL` rotations, you can still make the `Airport` class using the `BST`. This should be functionally equivalent to using the `AVL`, but less efficient as it might not be balanced.

When you are done you should submit your work as a tarball:

```
$ tar -cvzf airport_submit.tar.gz cs_airport/*.py
```

Grading

Your grade starts from 0, and the following tests determine your grade:

- +1pt if your `Node` contains all the required functionality and works properly.
- +2.5pts if your `BST` `insert` and `delete` work correctly, and update the node heights in `O(h)`.
- +1pt if `in_order` and `breadth_first` traversal of the `BST` work correctly.
- +0.5pt if the remaining `BST` functions work correctly.
- +3pts if the `AVL` tree is implemented correctly.
- +2pts if the `Airport` is implemented correctly.
- -10pts if any core problems of the assignment are purposely avoided by using built-in functions or libraries. (e.g. using a `sort` function for in-order traversal or statements like *import binary_tree*)