Priority Queues

Introduction

In this lab, we are going to write Python code that simulates events in an operating system. To this end you will write different implementations for the priority queue ADT. In particular, you will implement priority queue using list, BST, and balanced BST.

Objectives

The purpose of this lab is to help you:

1. Gain further familiarity with class inheritance
2. Understand the priority queue ADT
3. Learn about different implementations of priority queue
4. Understand Balanced BST

Events in an operating system

Event is an action or occurrence recognized by software. Events often are generated asynchronously and may be handled by the system. They are added to a queue of unprocessed events and handled in order of their priority. These events will be handled by pulling them from this queue of events.

The purpose of this lab is to simulate events in an operating system. Every event is an entity that has a timestamp, rank, and maybe some other data. A timestamp is a number generated by a global clock (`Simulator.clock`) that will be incremented by 1 for every add/delete operation of the queue.

Events will be handled not according to their insertion time but rather according to their priority. Priority is defined as follows:

- Events with lower rank have higher priority.
- If two events have the same rank, then the event with smaller (earlier) timestamp will have higher priority.

We have provided you with a simulator for simulating the events, creation and their handling. It is an object that continuously either creates an event and adds an event to a queue or removes an event from a queue. This simulator will run in a loop and randomly decide what to do. The user can either interrupt it or pre-specify the number of loop iterations before the simulator stops.

Accompanied code

This lab includes the following classes:

`PQ` represents the abstract methods of priority queue. These methods are needed in every implementation that you choose. We have provided you with the code for this.

`ListPQ` is class that represents a `PQ` implemented using Python list. This class inherits from `PQ` all abstract methods. We have provided you with the code for this, too.

`BST` is a class that represents a different implementation for priority queue. It also inherits from `PQ`, and all of the `PQ` `NotImplemented` methods MUST be implemented for it.

`BalancedBST` is a subclass of `BST` that overwrites `BST`'s `add` method to keep the tree balanced. You are required to implement it.

`TreeNode` represents a node in a `BST` or `BalancedBST`.

`Simulator` is a class that simulates the creation and handling of the events in the system. It it initialized with a priority queue (either `ListPQ`, `BST`, or `BalancedBST`) so as to add/remove event to/from it. It also has methods to make it QUIET or keep it LOUD (where it prints out for you what's going on). It also outputs a log of events, which can be used as an alternate way of running the simulator. By using the same log for two different simulator runs (with different `PQ` implementations), you can make sure that they work the same way.

Part 1 (in-lab) The event simulator

We have provided you with the `ListPQ` implementation and the `Simulator`. You should be able to do the following with them:

```
x = List() # we can alsodo BST or BalancedBST here
x.add(5)
x.add(9)
x.add(11)
x.add(10)
x.add(3)
x.add(4)
x.draw()
# len(x) should be 6, highest priority is 3
print("This ListPQ has", len(x), "items, highest priority is", x.peekMin())
y = x.getMin()
```

```
print("Removed", y, "here is what's left")
x.draw()

s = Simulator(ListPQ()) # interactive simulator with ListPQ impl
s1.setLimit(17) # will stop after processing 17 events
log1 = s1.run()

s2 = Simulator(List(), False) # the second argument makes it quiet
s2.useLog(log1) # this will run from log
log2 = s2.run()  # log1 and log2 should be identical
print("Total add time:", s2.addTime, "; Total get time:", s2.getTime)
```

Please try this code, and variations that you think of yourself. Also, study the implementations to understand what's going on.

Note that we've included timing in our simulator. It's printed on the last line above. It will come in useful later.

In the given implementations, event is represented by a tuple of two elements: `(rank, timestamp)`. To find the event with the highest priority we need find the one with the lowest rank. If there are two or more events with the same lowest rank we need to find the one with the lowest timestamp.

In the accompanied code we call the `priority` function for every event and compare priorities. In this part you are required to implement the `priority` function.

Part 2: Implementing `BST` `add` method

Another implementation for priority queue ADT is `BST`. You are given the following `BST` partial definition where a `BST` class inherits from `PQ` base class. This definition includes the magic method `__init__` that initializes new `BST` objects where `self.root` points to the tree root and `self.size` represents the number of events that have been added to the tree so far. Notice that every node in the `BST` is an object of type `TreeNode` class which is also given to you.

In this part you are required to write the `add` method of the `BST` class. In particular you have to implement/override all inherited methods from the `PQ` class that raise the `NotImplemented` exception.

Afterwards, the following code should work:

```
x = BST()
x.add(5)
```

```
x.add(9)
x.add(11)
x.add(10)
x.add(3)
x.add(4)
x.draw()
# len(x) should be 6, highest priority is 3
print("This ListPQ has", len(x), "items, highest priority is", x.peekMin())
```

Note that if the same elements are added to the list in a different order, the tree will look differently. Try it! Depending on the order in which the elements were added, the height of the `BST` can either be `O(log n)` or `O(n)`. This is an important point, because it means that we cannot count on the `BST` to give us `O(log n)` performance for `add` or `peekMin` or `getMin`. Balancing is needed to fix this problem, which we will do in part 4 of the lab.

Debugging tips

To help you debug here and in later parts, we suggest that you use the Simulator's capability to run from a log. It allows you to compare the behavior of your `BST` with that of `ListPQ`, over the same run of events. Also, we have provided `draw` methods for both of `BST` and `PQ` you can call for debugging purposes. Feel free to change them if it helps you.

Part 3: Implementing `BST` `getMin` method

Now implement `getMin` to complete the `BST` implementation. This method should remove the node that we find with `peekMin`, obtaining a `BST` with one less item.

After this, all the code that you see at the end of part 1 should work for `BST`s - meaning, if you replace `ListPQ()` by `BST()` - including the use of the Simulator. For example, this will work:

```
s1 = Simulator(BST()) # interactive simulator with BalancedBST impl
s1.setLimit(17) # will stop after processing 17 events
s1.run()

s = Simulator(List(),False) # this will be a long run, don't want it loud
s.setLimit(10000) # will stop after processing 10000 events
log = s.run()

s2 = Simulator(List(), False)
s2.useLog(log) # this will run from log
log1 = s2.run()  # log and log1 should be identical
print("Total add time:", s2.addTime, "; Total get time:", s2.getTime)

s3 = Simulator(BST(), False)
s3.useLog(log) # this will run from log
```

```python
log1 = s3.run()  # log and log1 should be identical
print("Total add time:", s3.addTime, "; Total get time:", s3.getTime)
```

Part 4: Implementing balanced BST `add` method

We have implemented priority queue using different data structures. In particular we used Python `list` and `BST`. However, neither implementations guarantees `O(log n)` performance. For example, finding the minimum event or removing it from a list will take `O(n)`. And in the `BST` implementation performance depends on the height of the tree, which can degrade to `O(n)` when the tree becomes unbalanced.

Definition (balanced tree): A tree is considered balanced if all of its nodes are balanced, meaning that heights of their left and right subtrees do not differ by more than 1. Otherwise, the tree is unbalanced.

Definition (height): If the child is `None`, its height is 0; if it's a leaf, its height is 1; if it's an internal node, its height is `1 + max(heights of its children)`.

To do better in the case of `BST`, we want the tree height to always be `O(log n)`. This is achieved by keeping the tree balanced; we call this a balanced BST, and it will be the third implementation of priority queues that you are seeing in this lab.

As balanced BST is a special case of `BST`, the `BalancedBST` class is a subclass of `BST`. The only `BST` method that needs to be overridden for the `BalancedBST` is `add`, and that is what you will be implementing for this part of the lab.

Note: we are also overriding `draw`, so the heights of the nodes are displayed when we draw the tree.

Here is the starting point definition of `BalancedBST` class:

```python
class BalancedBST(BST):
    def add(self, val):   # TO IMPLEMENT
        pass

    def draw(self):
        drawTree(self.root, 0, True)
```

Step 1: The `add` method for the `BalancedBST` class starts by calling the `add` method for the `BST` (which stays unchanged). Now this new node is added as a leaf, so it should have height `1` and it is already balanced by itself since both of its children have height `0`.

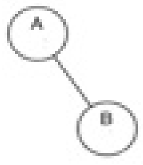
Note: the `BST` `add` method returns this new node to you.

Step 2: The addition of this new node might increase the heights of its ancestors, and as a result the tree might become unbalanced. After we add the new node, we need to traverse the tree from the new added node upward, using the parent link in our `BST` nodes have (note that this is done in a single loop without any recursion). That is, you will be going through all the ancestors of the new added node. For each ancestor, update its height using the formula in the definition above. When you are done, draw the tree and check that all the heights are correct.

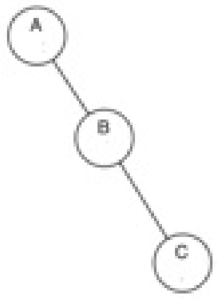
Step 3: in the same loop you created for step 2, right after updating the height of the ancestor, check if it's balanced, using the definition above. If not, then we know that the tree rooted at this ancestor is unbalanced and it needs to be rebalanced.

Note there are two cases for rebalancing: (a) left height is greater than right height, (b) right height is greater than left height. We call the difference between the left and right heights the balance factor. Note that in the first case, the balance factor is negative, and in the second, the balance factor is positive. Here is an example:

The balance factor of ‘B’ is 0 and of ‘A’ is -1 (left child height is 0 and right child height is 1). In this case the tree is balanced.



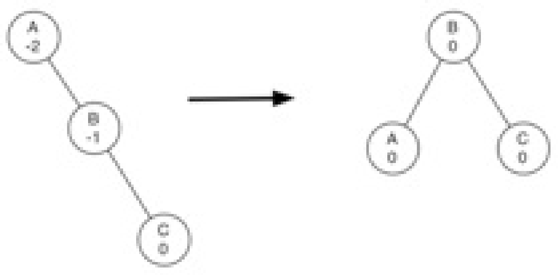
Now assume we have a new node and our previous tree becomes as follows:



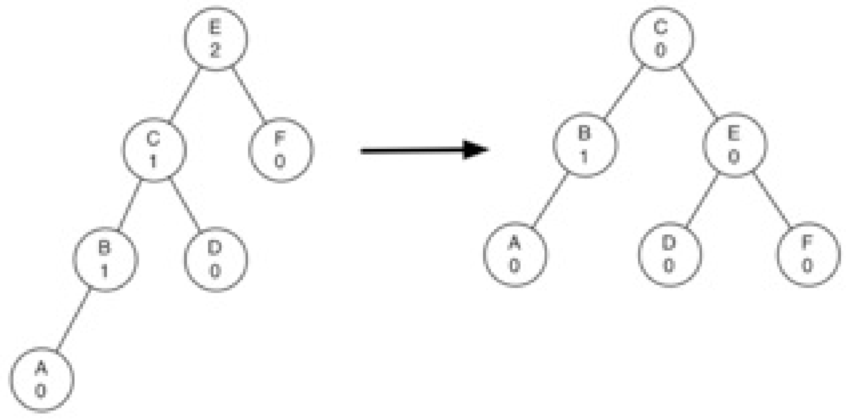
The balance factor of ‘C’ is 0 and for ‘B’ is -1 and for ‘A’ is -2 (the left child height is 0 and right child height is 2). In this case node ‘A’ is unbalanced and needs to be rebalanced.

For this step, update the draw method to show to you if a node is unbalanced and what its balance factor is. Try it for various simulator runs, and make sure this information looks correct.

Step 4: The `rebalance` method works on the idea of rotations; they come in two types, right and left. Here is an example of a left rotation, followed by an example of a right rotation.



And here is an example of a right rotation:



Here is the code for `rotateLeft`; `rotateRight` is a mirror image of this code with right and left swapped.

```
def rotateLeft(self,rotRoot):
    newRoot = rotRoot.rightChild
    rotRoot.rightChild = newRoot.leftChild
    if newRoot.leftChild != None:
        newRoot.leftChild.parent = rotRoot
    newRoot.parent = rotRoot.parent
    if rotRoot.isRoot():
        self.root = newRoot
    else:
        if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
        else:
            rotRoot.parent.rightChild = newRoot
    newRoot.leftChild = rotRoot
    rotRoot.parent = newRoot
```

In this step, you will implement left and right rotations as a separate (private) methods of `BalancedBST`, and then perform the following algorithm for rebalancing the node:

```
If the balance factor of current node < 0:
    if balance factor of the right child of the current node > 0
        rotate to the right the right child
        rotate to left the current node
    else
        rotate to the left the current node
else:  # the balance factor of current node > 0
    If balance factor of the left child of the current node < 0
        rotate to the left the left child
        rotate to the right the current node
    else
        rotate to the right the current node
```

Please run it on a bunch of examples and draw how the tree looks. It should be better than before, but if you play with it long enough, you should still find cases when things are unbalanced! Please find several such cases.

The reason there is a problem is because our rotation method does not update the heights. As you can see in the diagrams above, rotations can change the height of some nodes. So fix your rotation method, making sure to update again the heights of all the nodes whose children were changed during the rotation, still using the same formula as before.

Afterwards, try a bunch of runs to make sure that your trees always come out balanced. Congratulations, you've done it!

Part 5: Time Performance

Note that all the work that's done for each ancestor inside your loop in Part 4 is `O(1)`: updating the `height`, detecting if rebalancing is needed, and performing 1 or 2 rotations if needed. And going back to the root will take at most `O(log n)` loop iterations (the number of levels we have). So the add operation will run in `O(log n)`.

The get operation will also run in `O(log n)`. We've achieved optimal time complexity for implementing priority queues!

To compare the `BalancedBST` performance with other implementations please run the simulator for these different implementations for long runs. After each one, print out the time information, as we did in the example in part 1.

Try runs of length 10K, 100K, 1000K, 1M, 10M, and 100M. What do you observe??