

Recitation 8

Balanced Binary Trees

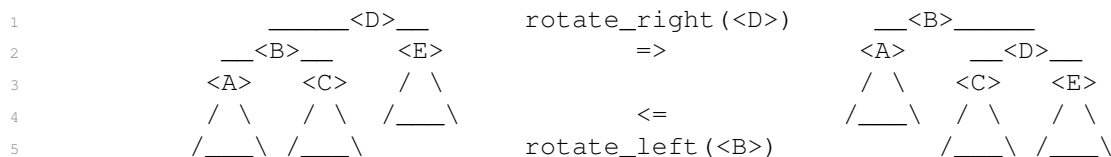
Previously, we discussed binary trees as a general data structure for storing items, without bounding the maximum height of the tree. The ultimate goal will be to keep our tree **balanced**: a tree on n nodes is balanced if its height is $O(\log n)$. Then all the $O(h)$ -time operations we talked about last time will only take $O(\log n)$ time.

There are many ways to keep a binary tree balanced under insertions and deletions (Red-Black Trees, B-Trees, 2-3 Trees, Splay Trees, etc.). The oldest (and perhaps simplest) method is called an **AVL Tree**. Every node of an AVL Tree is **height-balanced** (i.e., satisfies the **AVL Property**) where the left and right subtrees of a height-balanced node differ in height by at most 1. To put it a different way, define the **skew** of a node to be the height of its right subtree minus the height of its left subtree (where the height of an empty subtree is -1). Then a node is height-balanced if its skew is either $-1, 0$, or 1 . A tree is height-balanced if every node in the tree is height-balanced. Height-balance is good because it implies balance!

Exercise: A height-balanced tree is balanced. (height is $O(\log n)$)

Rotations

As we add or remove nodes to our tree, it is possible that our tree will become imbalanced. We will want to change the structure of the tree without changing its traversal order, in the hopes that we can make the tree's structure more balanced. We can change the structure of a tree using a local operation called a **rotation**. A rotation takes a subtree that locally looks like one of the following two configurations and modifies the connections between nodes in $O(1)$ time to transform it into the other configuration.



This operation preserves the traversal order of the tree while changing the depth of the nodes in subtrees `<A>` and `<E>`. Next time, we will use rotations to enforce that a balanced tree stays balanced after inserting or deleting a node.

```

1 def subtree_rotate_right(D):
2     assert D.left
3     B, E = D.left, D.right
4     A, C = B.left, B.right
5     D, B = B, D
6     D.item, B.item = B.item, D.item
7     B.left, B.right = A, D
8     D.left, D.right = C, E
9     if A: A.parent = B
10    if E: E.parent = D
11    B.subtree_update()
12    D.subtree_update()

def subtree_rotate_left(B): # O(1)
    assert B.right
    A, D = B.left, B.right
    C, E = D.left, D.right
    B, D = D, B
    B.item, D.item = D.item, B.item
    D.left, D.right = B, E
    B.left, B.right = A, C
    if A: A.parent = B
    if E: E.parent = D
    B.subtree_update()
    D.subtree_update()

```

Maintaining Height-Balance

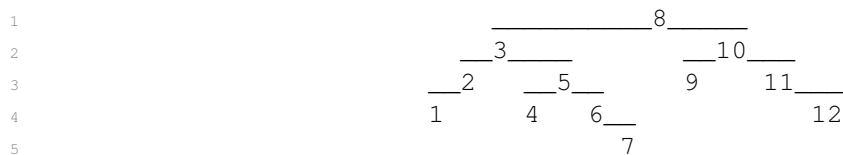
Suppose we have a height-balanced AVL tree, and we perform a single insertion or deletion by adding or removing a leaf. Either the resulting tree is also height-balanced, or the change in leaf has made at least one node in the tree have magnitude of skew greater than 1. In particular, the only nodes in the tree whose subtrees have changed after the leaf modification are ancestors of that leaf (at most $O(h)$ of them), so these are the only nodes whose skew could have changed and they could have changed by at most 1 to have magnitude at most 2. Given a subtree whose root has skew is 2 and every other node in its subtree is height-balanced, we can restore balance to the subtree in at most two rotations. Thus to rebalance the entire tree, it suffices to walk from the leaf to the root, rebalancing each node along the way, performing at most $O(\log n)$ rotations in total. Below is code to implement the rebalancing algorithm:

```

1      def skew(A):                                # O(?)
2          return height(A.right) - height(A.left)
3
4      def rebalance(A):                            # O(?)
5          if A.skew() == 2:
6              if A.right.skew() < 0:
7                  A.right.subtree_rotate_right()
8                  A.subtree_rotate_left()
9          elif A.skew() == -2:
10             if A.left.skew() > 0:
11                 A.left.subtree_rotate_left()
12                 A.subtree_rotate_right()
13
14     def maintain(A):                              # O(h)
15         A.rebalance()
16         A.subtree_update()
17         if A.parent: A.parent.maintain()

```

Exercise: An example operation requiring three rotations during maintenance deletes key 9 from the following tree. Confirm that each node in the tree satisfies the BST and AVL properties, then delete 9 from the tree, and update nodes, performing any necessary rotations up the tree.



Interactive Exercise: Make a BST by inserting student chosen keys one by one. If the tree ever does not satisfy the AVL Property, re-balance its ancestors going up the tree.

Previous 6.006 staff made a CoffeeScript BST/AVL visualizer which you can find here:

<https://codepen.io/mit6006/pen/NOWddZ>

Exercise: Infinite Printers: In the future, MIT's Infinite Corridor has extended to be even more infinite, with Athena clusters stationed at many locations along the **linear** hallway. Each Athena cluster has a unique integer **address** corresponding to its location along the Infinite Corridor, and contains many plasma printers which students utilize to print their futuristic homework via the online Pharos printer system (largely unchanged from its current implementation). Each printer has a **unique integer print quality** denoting how well it prints, though printers often go offline when they run out of resources, preventing students from printing to them until resources are replenished. Students want Pharos to tell them the best place to print. Design a print server which maintains the set of **working** printers, supporting the following operations: given a printer's cluster address and quality, take the printer online or offline; and given a student's location along the Infinite, return the quality of the **highest-quality working** printer, from among all working printers with **cluster address closest** to the student, keeping in mind there might be two clusters tied for closest.

Binary Node Implementation with AVL Balancing

```
1 def height(A):
2     if A: return A.height
3     else: return -1
4
5 class Binary_Node:
6     def __init__(A, x): # O(1)
7         A.item = x
8         A.left = None
9         A.right = None
10        A.parent = None
11        A.subtree_update()
12
13    def subtree_update(A): # O(1)
14        A.height = 1 + max(height(A.left), height(A.right))
15
16    def skew(A): # O(1)
17        return height(A.right) - height(A.left)
18
19    def subtree_iter(A): # O(n)
20        if A.left: yield from A.left.subtree_iter()
21        yield A
22        if A.right: yield from A.right.subtree_iter()
23
24    def subtree_first(A): # O(log n)
25        if A.left: return A.left.subtree_first()
26        else: return A
27
28    def subtree_last(A): # O(log n)
29        if A.right: return A.right.subtree_last()
30        else: return A
31
32    def successor(A): # O(log n)
33        if A.right: return A.right.subtree_first()
34        while A.parent and (A is A.parent.right):
35            A = A.parent
36        return A.parent
37
38    def predecessor(A): # O(log n)
39        if A.left: return A.left.subtree_last()
40        while A.parent and (A is A.parent.left):
41            A = A.parent
42        return A.parent
43
44    def subtree_insert_before(A, B): # O(log n)
45        if A.left:
46            A = A.left.subtree_last()
47            A.right, B.parent = B, A
48        else:
```

```

49         A.left, B.parent = B, A
50         A.maintain()
51
52     def subtree_insert_after(A, B):          # O(log n)
53         if A.right:
54             A = A.right.subtree_first()
55             A.left, B.parent = B, A
56         else:
57             A.right, B.parent = B, A
58             A.maintain()
59
60     def subtree_delete(A):                  # O(log n)
61         if A.left or A.right:
62             if A.left: B = A.predecessor()
63             else:      B = A.successor()
64             A.item, B.item = B.item, A.item
65             return B.subtree_delete()
66         if A.parent:
67             if A.parent.left is A: A.parent.left = None
68             else:                  A.parent.right = None
69             A.parent.maintain()
70         return A
71
72
73
74     def subtree_rotate_right(D):            # O(1)
75         assert D.left
76         B, E = D.left, D.right
77         A, C = B.left, B.right
78         D, B = B, D
79         D.item, B.item = B.item, D.item
80         B.left, B.right = A, D
81         D.left, D.right = C, E
82         if A: A.parent = B
83         if E: E.parent = D
84         B.subtree_update()
85         D.subtree_update()
86
87     def subtree_rotate_left(B):             # O(1)
88         assert B.right
89         A, D = B.left, B.right
90         C, E = D.left, D.right
91         B, D = D, B
92         B.item, D.item = D.item, B.item
93         D.left, D.right = B, E
94         B.left, B.right = A, C
95         if A: A.parent = B
96         if E: E.parent = D
97         B.subtree_update()
98         D.subtree_update()
99

```

```
100     def rebalance(A):                                     # O(1)
101         if A.skew() == 2:
102             if A.right.skew() < 0:
103                 A.right.subtree_rotate_right()
104                 A.subtree_rotate_left()
105         elif A.skew() == -2:
106             if A.left.skew() > 0:
107                 A.left.subtree_rotate_left()
108                 A.subtree_rotate_right()
109
110     def maintain(A):                                       # O(log n)
111         A.rebalance()
112         A.subtree_update()
113         if A.parent: A.parent.maintain()
```