*Introduction to Algorithms: 6.006*
Massachusetts Institute of Technology March 12, 2021
Instructors: Mauricio Karchmer, Anand Natarajan, Nir Shavit Recitation 8

# 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 it's 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!

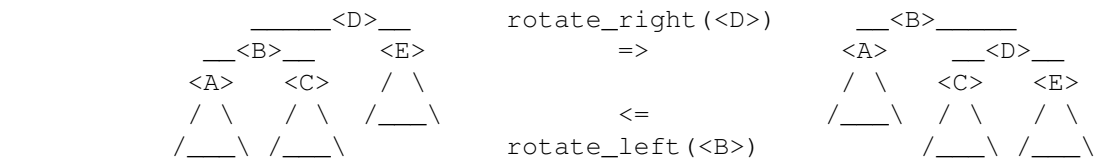
**Exercise:** A height-balanced tree is balanced. (height is $O(\log n)$)
**Solution:** Balanced means that $h = O(\log n)$. Equivalently, balanced means that $\log n$ is lower bounded by $\Omega(h)$ so that $n = 2^{\Omega(h)}$. So if we can show the minimum number of nodes in a height-balanced tree is at least exponential in $h$, then it must also be balanced. Let $F(h)$ denote the fewest nodes in any height-balanced tree of height $h$. Then $F(h)$ satisfies the recurrence:

$$F(h) = 1 + F(h-1) + F(h-2) \geq 2F(h-2),$$

since the subtrees of the root's children should also contain the fewest nodes. As base cases, the fewest nodes in a height-balanced tree of height 0 is one, i.e., $F(0) = 1$, while the fewest nodes in a height-balanced tree of height 1 is two, i.e., $F(1) = 2$. Then this recurrence is lower bounded by $F(h) \geq 2^{h/2} = 2^{\Omega(h)}$ as desired.

## 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 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.

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

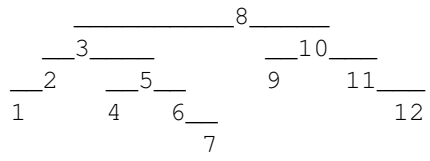
Suppose we have a height-balanced AVL tree, and we perform a single insertion or deletion by adding or removing a leaf. Either the resulting tree is also height-balanced, or the change in leaf has made at least one node in the tree have magnitude of skew greater than 1. In particular, the only nodes 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:

```
    def skew(A):                                # O(?)
        return height(A.right) - height(A.left)

    def rebalance(A):                           # O(?)
        if A.skew() == 2:
            if A.right.skew() < 0:
                A.right.subtree_rotate_right()
            A.subtree_rotate_left()
        elif A.skew() == -2:
            if A.left.skew() > 0:
                A.left.subtree_rotate_left()
            A.subtree_rotate_right()

    def maintain(A):                            # O(h)
        A.rebalance()
        A.subtree_update()
        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. (**Solution:** Left rotation at 10, left rotation at 3, right rotation at 8).




**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.

**Solution: First Solution (tree of pairs)**: Store an AVL tree of **working** printers, keyed on the pair `(address, quality)`. Keys are sorted by address first (in increasing order) and then quality (in increasing order). To take a printer online or offline, we can `insert` or `delete` the node from the tree, in $O(\log n)$ time.

To query an address $a$, define

`p = find_prev((a, infinity))` and `q = find_next((a, -infinity))`.

Finally, let `r = find_prev((q.address, infinity))`. Return whichever of $p$ or $r$ is closer, or both if they're the same distance away.

To show that this returns the correct printer(s), first observe that $p$ is the working printer that has highest address $\leq a$ and, of those, the one with highest quality. So $p$ is the desired printer in the backward direction. Similarly, $q$ is the working printer with least address $\geq a$ and, of those, the one with **least** quality. This is not the most desirable printer in the forward direction, but it has the closest forward address of any printer ahead of $a$. Printer $r$ is the highest-quality printer with the same address as $q$ and is thus the most desirable printer in the forward direction. This performs up to three AVL tree operations and thus takes $O(\log n)$ time.

**Second Solution (tree of trees)**: Store an AVL tree $T$ whose items correspond to the clusters that have at least one working printer, keyed on address. Each cluster item stores its own AVL tree of its working printers, keyed on quality.

To insert a printer, first `find` the cluster item at the correct address (or create it if there isn't one already), and then `insert` the printer into that cluster's tree. To delete, we similarly `find` the cluster at the correct address and `delete` that printer from the tree, but in addition, if that cluster now has no working printers (its tree is empty), we `delete` that custer from $T$. This maintains the invariant that every cluster in $T$ has at least one working printer.

To query an address $a$, we find the closest forward and backward clusters with `find` or, if there is no cluster directly at $a$, with `find_next` and `find_prev`. For whichever cluster is closer (or both), return the highest-quality printer from that cluster with `find_max`.

There are at most $n$ clusters and each cluster contains at most $n$ printers, so each AVL tree operation on any of the trees is $O(\log n)$. This gives $O(\log n)$ time overall.

## Binary Node Implementation with AVL Balancing

```
def height(A):
    if A:   return A.height
    else:   return -1

class Binary_Node:
    def __init__(A, x):                     # O(1)
        A.item   = x
        A.left   = None
        A.right  = None
        A.parent = None
        A.subtree_update()

    def subtree_update(A):                  # O(1)
        A.height = 1 + max(height(A.left), height(A.right))

    def skew(A):                            # O(1)
        return height(A.right) - height(A.left)

    def subtree_iter(A):                    # O(n)
        if A.left:   yield from A.left.subtree_iter()
        yield A
        if A.right:  yield from A.right.subtree_iter()

    def subtree_first(A):                   # O(log n)
        if A.left:  return A.left.subtree_first()
        else:       return A

    def subtree_last(A):                    # O(log n)
        if A.right: return A.right.subtree_last()
        else:       return A

    def successor(A):                       # O(log n)
        if A.right: return A.right.subtree_first()
        while A.parent and (A is A.parent.right):
            A = A.parent
        return A.parent

    def predecessor(A):                     # O(log n)
        if A.left:  return A.left.subtree_last()
        while A.parent and (A is A.parent.left):
            A = A.parent
        return A.parent

    def subtree_insert_before(A, B):        # O(log n)
        if A.left:
            A = A.left.subtree_last()
            A.right, B.parent = B, A
        else:
```

```
            A.left,  B.parent = B, A
        A.maintain()

    def subtree_insert_after(A, B):          # O(log n)
        if A.right:
            A = A.right.subtree_first()
            A.left,  B.parent = B, A
        else:
            A.right, B.parent = B, A
        A.maintain()

    def subtree_delete(A):                   # O(log n)
        if A.left or A.right:
            if A.left:  B = A.predecessor()
            else:       B = A.successor()
            A.item, B.item = B.item, A.item
            return B.subtree_delete()
        if A.parent:
            if A.parent.left is A:  A.parent.left  = None
            else:                   A.parent.right = None
            A.parent.maintain()
        return A



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

```

```
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()
```