

## Recitation 7

### 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.

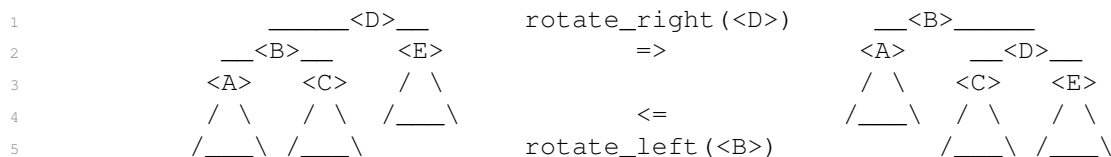
**Solution:** Balanced means that  $h = O(\log n)$ . Equivalently, balanced means that  $\log n$  is lower bounded by  $\Omega(h)$  so that  $n = 2^{\Omega(h)}$ . So if we can show the minimum number of nodes in a height-balanced tree is at least exponential in  $h$ , then it must also be balanced. Let  $F(h)$  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 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):          def subtree_rotate_left(B): # O(1)
2      assert D.left                    assert B.right
3      B, E = D.left, D.right           A, D = B.left, B.right
4      A, C = B.left, B.right           C, E = D.left, D.right
5      D, B = B, D                     B, D = D, B
6      D.item, B.item = B.item, D.item  B.item, D.item = D.item, B.item
7      B.left, B.right = A, D           D.left, D.right = B, E
8      D.left, D.right = C, E           B.left, B.right = A, C
9      if A: A.parent = B               if A: A.parent = B
10     if E: E.parent = D               if E: E.parent = D
11     # B.subtree_update()              # B.subtree_update()      # wait for R07!
12     # D.subtree_update()              # D.subtree_update()      # wait for R07!

```

## 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. As shown in lecture via a brief case analysis, 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. A detailed proof is outlined in the lecture notes and is not repeated here; but the proof may be reviewed in recitation if students would like to see the

full argument. Below is code to implement the rebalancing algorithm presented in lecture.

```

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()

```

Unfortunately, it's not clear how to efficiently evaluate the skew of a node to determine whether or not we need to perform rotations, because computing a node's height naively takes time linear in the size of the subtree. The code below to compute height recurses on every node in  $A$ 's subtree, so takes at least  $\Omega(n)$  time.

```

1     def height(A):                              # Omega(n)
2         if A is None: return -1
3         return 1 + max(height(A.left), height(A.right))

```

Rebalancing requires us to check at least  $\Omega(\log n)$  heights in the worst-case, so if we want rebalancing the tree to take at most  $O(\log n)$  time, we need to be able to evaluate the height of a node in  $O(1)$  time. Instead of computing the height of a node every time we need it, we will speed up computation via augmentation: in particular each node stores and maintains the value of its own subtree height. Then when we're at a node, evaluating its height is as simple as reading its stored value in  $O(1)$  time. However, when the structure of the tree changes, we will need to update and recompute the height at nodes whose height has changed.

```

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

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

```

In the dynamic operations presented in R06, we put commented code to call update on every node whose subtree changed during insertions, deletions, or rotations. A rebalancing insertion or deletion operation only calls `subtree_update` on at most  $O(\log n)$  nodes, so as long as updating a

node takes at most  $O(1)$  time to recompute augmentations based on the stored augmentations of the node's children, then the augmentations can be maintained during rebalancing in  $O(\log n)$  time.

In general, the idea behind **augmentation** is to store additional information at each node so that information can be queried quickly in the future. You've done some augmentation already in PS1, where you augmented a singly-linked list with back pointers to make it faster to evaluate a node's predecessor. To augment the nodes of a binary tree with a **subtree** property  $P(<X>)$ , you need to:

- clearly define what property of  $<X>$ 's subtree corresponds to  $P(<X>)$ , and
- show how to compute  $P(<X>)$  in  $O(1)$  time from the augmentations of  $<X>$ 's children.

If you can do that, then you will be able to store and maintain that property at each node without affecting the  $O(\log n)$  running time of rebalancing insertions and deletions. We've shown how to traverse around a binary tree and perform insertions and deletions, each in  $O(h)$  time while also maintaining height-balance so that  $h = O(\log n)$ . Now we are finally ready to implement an efficient Sequence and Set.

## 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()
```

## Application: Set

Using our new definition of `Binary_Node` that maintains balance, the implementation presented in R06 of the `Binary_Tree_Set` immediately supports all operations in  $h = O(\log n)$  time, except `build(X)` and `iter()` which run in  $O(n \log n)$  and  $O(n)$  time respectively. This data structure is what's normally called an **AVL tree**, but what we will call a **Set AVL**.

## Application: Sequence

To use a Binary Tree to implement a Sequence interface, we use the traversal order of the tree to store the items in Sequence order. Now we need a fast way to find the  $i^{\text{th}}$  item in the sequence because traversal would take  $O(n)$  time. If we knew how many items were stored in our left subtree, we could compare that size to the index we are looking for and recurse on the appropriate side. In order to evaluate subtree size efficiently, we augment each node in the tree with the size of its subtree. A node's size can be computed in constant time given the sizes of its children by summing them and adding 1.

```

1 class Size_Node(Binary_Node):
2     def subtree_update(A):                                # O(1)
3         super().subtree_update()
4         A.size = 1
5         if A.left:    A.size += A.left.size
6         if A.right:   A.size += A.right.size
7
8     def subtree_at(A, i):                                  # O(h)
9         assert 0 <= i
10        if A.left:    L_size = A.left.size
11        else:         L_size = 0
12        if i < L_size: return A.left.subtree_at(i)
13        elif i > L_size: return A.right.subtree_at(i - L_size - 1)
14        else:         return A

```

Once we are able to find the  $i^{\text{th}}$  node in a balanced binary tree in  $O(\log n)$  time, the remainder of the Sequence interface operations can be implemented directly using binary tree operations. Further, via the first exercise in R06, we can build such a tree from an input sequence in  $O(n)$  time. We call this data structure a **Sequence AVL**.

Implementations of both the Sequence and Set interfaces can be found on the following pages. We've made a CoffeeScript Balanced Binary Search Tree visualizer which you can find here:

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

```
1 class Seq_Binary_Tree(Binary_Tree):
2     def __init__(self): super().__init__(Size_Node)
3
4     def build(self, X):
5         def build_subtree(X, i, j):
6             c = (i + j) // 2
7             root = self.Node_Type(A[c])
8             if i < c:
9                 root.left = build_subtree(X, i, c - 1)
10                root.left.parent = root
11            if c < j:
12                root.right = build_subtree(X, c + 1, j)
13                root.right.parent = root
14            root.subtree_update()
15            return root
16        self.root = build_subtree(X, 0, len(X) - 1)
17        self.size = self.root.size
18
19    def get_at(self, i):
20        assert self.root
21        return self.root.subtree_at(i).item
22
23    def set_at(self, i, x):
24        assert self.root
25        self.root.subtree_at(i).item = x
26
27    def insert_at(self, i, x):
28        new_node = self.Node_Type(x)
29        if i == 0:
30            if self.root:
31                node = self.root.subtree_first()
32                node.subtree_insert_before(new_node)
33            else:
34                self.root = new_node
35        else:
36            node = self.root.subtree_at(i - 1)
37            node.subtree_insert_after(new_node)
38        self.size += 1
39
40    def delete_at(self, i):
41        assert self.root
42        node = self.root.subtree_at(i)
43        ext = node.subtree_delete()
44        if ext.parent is None: self.root = None
45        self.size -= 1
46        return ext.item
47
48    def insert_first(self, x): self.insert_at(0, x)
49    def delete_first(self): return self.delete_at(0)
50    def insert_last(self, x): self.insert_at(len(self), x)
51    def delete_last(self): return self.delete_at(len(self) - 1)
```



**Exercise:** Make a Sequence AVL Tree or Set AVL Tree (Balanced Binary Search Tree) by inserting student chosen items one by one. If any node becomes height-imbalanced, rebalance its ancestors going up the tree. Here's a Sequence AVL Tree example that may be instructive (remember to update subtree heights and sizes as you modify the tree!).

```

1 T = Seq_Binary_Tree()
2 T.build([10,6,8,5,1,3])
3 T.get_at(4)
4 T.set_at(4, -4)
5 T.insert_at(4, 18)
6 T.insert_at(4, 12)
7 T.delete_at(2)

```

### Solution:

Line #	1	2,3	4	5	6	7
Result	None	8	8	8	8	12
		10 1	10 -4	10 -4	10 -4	6 -4
		6 5 3	6 5 3	6 5 3	6 12 3	10 5 18 3
				18	5 18	

Also labeled with subtree height H, size #:

```

10 None
11      8H2#6
12 10H1#2      1H1#3
13   6H0#1    5H0#1    3H0#1
14
15      8H2#6
16 10H1#2      1H1#3
17   6H0#1    5H0#1    3H0#1
18
19      8H2#6
20 10H1#2      -4H1#3
21   6H0#1    5H0#1    3H0#1
22
23      8H3#7
24 10H1#2      -4H2#4
25   6H0#1    5H1#2      3H0#1
26             18H0#1
27
28      8H3#8
29 10H1#2      -4H2#5
30   6H0#1    12H1#3      3H0#1
31             5H0#1    18H0#1
32
33      12H2#7
34 6H1#3      -4H1#3
35 10H0#1    5H0#1    18H0#1    3H0#1

```

**Exercise:** Maintain a sequence of  $n$  bits that supports two operations, each in  $O(\log n)$  time:

- `flip(i)` : flip the bit at index  $i$
- `count_ones_upto(i)` : return the number of bits in the prefix up to index  $i$  that are one

**Solution:** Maintain a Sequence Tree storing the bits as items, augmenting each node  $A$  with `A.subtree_ones`, the number of 1 bits in its subtree. We can maintain this augmentation in  $O(1)$  time from the augmentations stored at its children.

```

1 def update(A):
2     A.subtree_ones = A.item
3     if A.left:
4         A.subtree_ones += A.left.subtree_ones
5     if A.right:
6         A.subtree_ones += A.right.subtree_ones

```

To implement `flip(i)`, find the  $i^{\text{th}}$  node  $A$  using `subtree_node_at(i)` and flip the bit stored at `A.item`. Then update the augmentation at  $A$  and every ancestor of  $A$  by walking up the tree in  $O(\log n)$  time.

To implement `count_ones_upto(i)`, we will first define the subtree-based recursive function `subtree_count_ones_upto(A, i)` which returns the number of 1 bits in the subtree of node  $A$  that are at most index  $i$  within  $A$ 's subtree. Then `count_ones_upto(i)` is semantically equivalent to `subtree_count_ones_upto(T.root, i)`. Since each recursive call makes at most one recursive call on a child, operation takes  $O(\log n)$  time.

```

1 def subtree_count_ones_upto(A, i):
2     assert 0 <= i < A.size
3     out = 0
4     if A.left:
5         if i < A.left.size:
6             return subtree_count_ones_upto(A.left, i)
7         out += A.left.subtree_ones
8         i -= A.left.size
9     out += A.item
10    if i > 0:
11        assert A.right
12        out += subtree_count_ones_upto(A.right, i - 1)
13    return out

```