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 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.
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.

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.

```
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()
```

Unfortunately, it's not clear how to efficiently evaluate the skew of a 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.

```
def height(A):                              # Omega(n)
    if A is None: return -1
    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 a 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.

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

def subtree_update(A):                  # O(1)
    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. We will build these interfaces on top of a general Binary Tree data structure that stores a pointer to its root, and the number of items it stores.

```
class Binary_Tree:
    def __init__(T, Node_Type):
        T.root = None
        T.size = 0
        T.Node_Type = Node_Type

    def __len__(T): return T.size
    def __iter__(T):
        if T.root:
            for A in T.root.subtree_iter():
                yield A.item
```

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^{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.

```
class Size_Node(Binary_Node):
    def subtree_update(A):                  # O(1)
        super().subtree_update()
        A.size = 1
        if A.left:   A.size += A.left.size
        if A.right:  A.size += A.right.size

    def subtree_node_at(A, i):              # O(h)
        assert 0 <= i
        if A.left:        L_size = A.left.size
        else:             L_size = 0
        if i < L_size:    return A.left.subtree_node_at(i)
        elif i > L_size:  return A.right.subtree_node_at(i - L_size - 1)
        else:             return A
```

Once we are able to find the i^{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.

Application: Set

To use a Binary Tree to implement a Set interface, we use the traversal order of the tree to store the items sorted in increasing key order. This property is often called the **Binary Search Tree Property**, where keys in a node's left subtree are less than the key stored at the node, and keys in the node's right subtree are greater than the key stored at the node. Then finding the node containing a query key (or determining that no node contains the key) can be done by walking down the tree, recursing on the appropriate side. 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`

Exercise: Make a Sequence Tree and/or a Binary Search Tree by inserting student chosen items one by one. If any node becomes height-imbalanced, rebalance its ancestors going up the tree.

```
class Binary_Tree_Seq(Binary_Tree):
    def __init__(self): super().__init__(Size_Node)

    def build(self, A):
        def build_subtree(A, i, j):
            c = (i + j) // 2
            root = self.Node_Type(A[c])
            if i < c:
                root.left = build_subtree(A, i, c - 1)
                root.left.parent = root
            if c < j:
                root.right = build_subtree(A, c + 1, j)
                root.right.parent = root
            root.subtree_update()
            return root
        self.root = build_subtree(A, 0, len(A) - 1)
        self.size = self.root.size

    def get_at(self, i):
        assert self.root
        return self.root.subtree_node_at(i).item

    def set_at(self, i, x):
        assert self.root
        self.root.subtree_node_at(i).item = x

    def insert_at(self, i, x):
        new_node = self.Node_Type(x)
        if i == 0:
            if self.root:
                node = self.root.subtree_first()
                node.subtree_insert_before(new_node)
            else:
                self.root = new_node
        else:
            node = self.root.subtree_node_at(i - 1)
            node.subtree_insert_after(new_node)
        self.size += 1

    def delete_at(self, i):
        assert self.root
        node = self.root.subtree_node_at(i)
        ext = node.subtree_extract()
        if ext.parent is None:  self.root = None
        self.size -= 1
        return ext.item

    def insert_first(self, x):  self.insert_at(0, x)
    def delete_first(self):     return self.delete_at(0)
    def insert_last(self, x):   self.insert_at(len(self), x)
    def delete_last(self):      return self.delete_at(len(self) - 1)
```

```
class BST_Node(Binary_Node):
    def subtree_find(A, k):                 # O(h)
        if k < A.item.key:
            if A.left:  return A.left.subtree_find(k)
        elif k > A.item.key:
            if A.right: return A.right.subtree_find(k)
        else:           return A
        return None

    def subtree_find_next(A, k):            # O(h)
        if A.item.key <= k:
            if A.right: return A.right.subtree_find_next(k)
            else:       return None
        elif A.left:
            B = A.left.subtree_find_next(k)
            if B:       return B
        return A

    def subtree_find_prev(A, k):             # O(h)
        if A.item.key >= k:
            if A.left:  return A.left.subtree_find_prev(k)
            else:       return None
        elif A.right:
            B = A.right.subtree_find_prev(k)
            if B:       return B
        return A

    def subtree_insert(A, B):                # O(h)
        if B.item.key < A.item.key:
            if A.left:  A.left.subtree_insert(B)
            else:       A.subtree_insert_before(B)
        elif B.item.key > A.item.key:
            if A.right: A.right.subtree_insert(B)
            else:       A.subtree_insert_after(B)
        else:    A.item = B.item
```

```
class Binary_Tree_Set(Binary_Tree):
    def __init__(self): super().__init__(BST_Node)

    def iter_order(self): yield from self

    def build(self, A):
        for x in A: self.insert(x)

    def find_min(self):
        if self.root:   return self.root.subtree_first().item

    def find_max(self):
        if self.root:   return self.root.subtree_last().item

    def find(self, k):
        if self.root:
            node = self.root.subtree_find(k)
            if node:    return node.item

    def find_next(self, k):
        if self.root:
            node = self.root.subtree_find_next(k)
            if node:    return node.item

    def find_prev(self, k):
        if self.root:
            node = self.root.subtree_find_prev(k)
            if node:    return node.item

    def insert(self, x):
        new_node = self.Node_Type(x)
        if self.root:
            self.root.subtree_insert(new_node)
            if new_node.parent is None: return False
        else:
            self.root = new_node
        self.size += 1
        return True

    def delete(self, k):
        assert self.root
        node = self.root.subtree_find(k)
        assert node
        ext = node.subtree_extract()
        if ext.parent is None:  self.root = None
        self.size -= 1
        return ext.item
```

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.

```
def update(A):
    A.subtree_ones = A.item
    if A.left:
        A.subtree_ones += A.left.subtree_ones
    if A.right:
        A.subtree_ones += A.right.subtree_ones
```

To implement `flip(i)`, find the i^{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 symantically equivilent 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.

```
def subtree_count_ones_upto(A, i):
    assert 0 <= i < A.size
    out = 0
    if A.left:
        if i < A.left.size:
            return subtree_count_ones_upto(A.left, i)
        out += A.left.subtree_ones
        i -= A.left.size
    out += A.item
    if i > 0:
        assert A.right
        out += subtree_count_ones_upto(A.right, i - 1)
    return out
```