Recitation 7

Binary Tree Augmentations

Suppose we have a Binary Search Tree and we want to maintain, for each node A, its height. How can we do so efficiently, even as new nodes are being added or deleted? The idea is to augment the information kept at each node. 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))
```

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 or deletions. A 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. 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. We will soon show how to do this 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

```
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 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 maintain(A):                        # O(log n)
        A.subtree_update()
        if A.parent: A.parent.maintain()
```

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.

```python
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_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_at(i)
        elif i > L_size: return A.right.subtree_at(i - L_size - 1)
        else:            return A
```

Once we are able to find the i^{th} node in a balanced binary tree in $O(h)$ 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.

An implementation of the Sequence interface can be found on the following pages.

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

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

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

    def set_at(self, i, x):
        assert self.root
        self.root.subtree_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_at(i - 1)
            node.subtree_insert_after(new_node)
        self.size += 1

    def delete_at(self, i):
        assert self.root
        node = self.root.subtree_at(i)
        ext = node.subtree_delete()
        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)
```

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
```

Exercise: We would like to design a data structure to help Dr. Where store hospital records. Each record consists of a day and a list of patients who came in that day. Unfortunately, his assistant is very forgetful, and often forgets to write down patients who came in on a day, so Dr. Where often needs to update a particular day with an additional patient that his assistant missed.

Dr. Where would like to perform the following operations for records in the given runtimes, where n is the number of records in our data structure:

- `build(record_list)`: build the data structure in $O(n \log n)$ time.
- `insert(day, patient_list)`: insert a record for a given day with a given `patient_list` in $O(\log n)$.
- `update(day, new_patient)`: add a new patient to a record on a particular day in $O(\log n)$ time.
- `query(day)`: find all the patients that came in on a particular day in $O(\log n)$ time.

Solution: Maintain a balanced binary tree on the records, where keys are days and values are a list of patients that visited on that day. When we insert, we add a new node with the key `day` storing the value `patient_list`, which is an array. When we update, we query our tree for the node with key `day` and append the `new_patient` to the list of patients stored in the node. Since each of these operations are performed on a balanced BST, they all run in $O(\log n)$ time. Finally, we know that building a BST on n nodes takes $O(n \log n)$ time.

Exercise: Given a BST on n nodes, we would like to find the diameter of the tree in $O(1)$ time. The **diameter** of a tree is the longest path between a pair of nodes. We will maintain the BST in a way that supports the following operations:

- `insert(key)`: insert a new node with given key in $O(h)$ time, where h is the height of the tree.
- `get_diameter()`: find the diameter of the BST in $O(1)$ time.

Solution: We will augment each node with the node's height and the diameter of the subtree rooted at the node. Thus when `get_diameter()` is called, we simply return `root.diameter`.

Insertion is a bit more complicated. When we insert a new node, we perform the insertion into the tree the usual way, which takes $O(h)$ time. However, now we need to update the augmented data in each of the ancestors of this node. For each ancestor node, the diameter of the subtree rooted at the node either passes through the node itself, or it is entirely contained within the left or right subtrees of the node. For each of these cases, we will calculate the path length.

If the diameter passes through the node, then its length will be
`node.left.height + node.right.height + 2`. We add 2 because of the addition of

the distance between the node and its left and right children. If the diameter is entirely contained within the left subtree of the node, then the length will be `node.left.diameter`. If the diameter is entirely contained within the right subtree, then the length will be `node.right.diameter`.

Thus `node.diameter` will be the maximum of these three lengths. Each of these calculations take constant time when we walk up the tree, so we can update each ancestor node's augmented data in constant time. Each node has at most $O(h)$ ancestors, so insertion will take $O(h)$ time.