

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.

```

1 def height(A):                                # Omega(n)
2     if A is None: return -1
3     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 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 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

```

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
17    def subtree_iter(A): # O(n)
18        if A.left: yield from A.left.subtree_iter()
19        yield A
20        if A.right: yield from A.right.subtree_iter()
21
22    def subtree_first(A): # O(log n)
23        if A.left: return A.left.subtree_first()
24        else: return A
25
26    def subtree_last(A): # O(log n)
27        if A.right: return A.right.subtree_last()
28        else: return A
29
30    def successor(A): # O(log n)
31        if A.right: return A.right.subtree_first()
32        while A.parent and (A is A.parent.right):
33            A = A.parent
34        return A.parent
35
36    def predecessor(A): # O(log n)
37        if A.left: return A.left.subtree_last()
38        while A.parent and (A is A.parent.left):
39            A = A.parent

```

```
40     return A.parent
41
42     def subtree_insert_before(A, B):          # O(log n)
43         if A.left:
44             A = A.left.subtree_last()
45             A.right, B.parent = B, A
46         else:
47             A.left, B.parent = B, A
48         A.maintain()
49
50     def subtree_insert_after(A, B):           # O(log n)
51         if A.right:
52             A = A.right.subtree_first()
53             A.left, B.parent = B, A
54         else:
55             A.right, B.parent = B, A
56         A.maintain()
57
58     def subtree_delete(A):                   # O(log n)
59         if A.left or A.right:
60             if A.left: B = A.predecessor()
61             else:      B = A.successor()
62             A.item, B.item = B.item, A.item
63             return B.subtree_delete()
64         if A.parent:
65             if A.parent.left is A: A.parent.left = None
66             else:                  A.parent.right = None
67             A.parent.maintain()
68         return A
69
70
71     def maintain(A):                         # O(log n)
72         A.subtree_update()
73         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.

```

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^{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.

```
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: 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

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.

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.