Binary Search Trees

New Abstract Data Type, Refresh of one we've seen

- We've seen one/two ended access data structures: Stack/Queue
- We've seen sequence containers (array, linked list)
- We've seen trees for hierarchical data
- We've briefly touched on Sets, but will go into more detail today
- We'll introduce another ADT similar to a set called a Map

Sets

- Recall, sets have 3 main operations:
  - bool add(element) — returns true if it wasn't in there aleady
  - bool contains(element)
  - bool remove(element) — returns true if it was there before calling remove
- Sets are often iterable, and have size, etc methods
- So far, our best bet is to use arrays

Unsorted Array Implementation

- Append to the end of the array Insert is O(1) or O(N) if we want to ensure no duplicates
- Contains, Remove are O(N)
- Note: remove is O(N) because of searching. We can swap the deleted element with another since order doesn't matter!

Sorted Array Implementation

- O(N) insert/remove (requires shifting)
- O(Lg N) contains (binary search)
- Having a fast search means modification operations must be slow

Ideas?

- Want to have the Lg N search characteristics of the sorted array
- Want to have the fast insert (and potentially remove) of unsorted
- Means we need to be able to insert into the middle of a sorted list efficiently
- Linked lists let us insert into the middle efficiently, but searching/iterating are slow

Solution: Binary search tree

- Keep the data sorted
- Fast insert by using a linked data structure
- Use a tree data structure to avoid iterating through the whole list
- Ideally we start every search/traversal from the middle of the list, best case for binary search

BST definition

- Binary tree (nodes have left + right pointers)
- For each node, n: all elements in the left subtree have values less than the value at n. All elements in the right subtree have values greater than value at n
- This is the “Binary Search Property”
- What happens if we do an “in-order” traversal on a BST?

Searching a BST

```
search(Node n, target):
    if n.data == target, return n
    else if target ≤ n.data:
        search n.left
    else
        search n.right
```

- Base cases?
- What if n's not in the BST?

Contains:

- Return search(root, target) != null, and its data == target

Insert:

```
n = search(root, target)
    if n.data == target: return false
    else:
        create a new node and attach it to n
        return true
```

Remove:

- The tricky one!
- Predecessor is the node with the next-smallest value

```
n = search(root, target)
if n is null: return false
if n is a leaf: update its parent to set that child to null
if n has 1 child, set n.parent.child = n.child (left/right appropriately)
else:
    find the "predecessor" or "successor" of target
    swap it's value into target's node
    delete the predecessor/successor, which much be a leaf, so it's easy to delete
```

Finding predecessor/successor:

- Predecessor: go left, then right as far as you can
- Successor: go right, the left as far as you can

Performance

- All operations may requiring traversing from the top to the bottom of the tree
- So all operations are O(tree height)
- Big O because they might not go all the way down the tree
- So how tall is the tree?

Best Case

- If a tree has N nodes, how tall does it have to be?
- The shortest tree is one that is as well balanced as possible
- How tall is it?
- First level has 1 node
- Second level has 2 nodes
- 3rd level has 4 nodes
- ...
- How many levels do we need?

Best Case

- N_h = 1 + 2 + 4 + ... 2^h
- N_h = 2^(h +1) - 1 ~= 2^(h + 1)
- What H gives us N? lg(N)
- So all operartions are best case O(lg N)

Worst Case

- Tree is as unbalanced as possible
- An unbalanced BST is a linked list
- O(N) height, so O(N) complexity for everything

Map ADT

- Map is an ADT with similar implementation, but different interface to Set
- Map is also called “associative array”
- Looks like an array, except that instead of integer indices, we have “keys” which can be of any type
- Java interface is Map where K is the key type, and V is the value type (analogous to the Array type)
- Method are basically get(K) and set(K, V), and delete(K)

TreeMap

- Data is a K, V pair
- Same implementation as the set based BST we've discussed, except we only look at the K part when searching