Dictionaries are the **more general** structure, but, in terms of implementation, they're nearly identical.

In a `Set`, we store the `key` directly, but conceptually, there's nothing different in storing an **Item**:

(k, v)

```
class Item {
   Data key;
   Data value;
}
```

The `Set ADT` usually has our favorite operations: intersection, union, etc.

Notice that union, intersection, etc. **still make sense on maps**!

As always, depending on our usage, we might choose to add/delete things from out `ADT`.

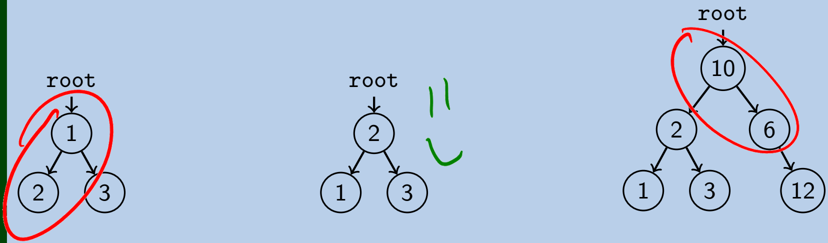
Bottom Line: If we have a set implementation, we also have a valid dictionary implementation (and vice versa)!

For each of the following potential implementations, what is the worst case runtime for `insert`, `find`, `delete`?

	insert	Find	delete
Unsorted Array	O(n)	O(n)	O(n)
Unsorted Linked List			
Sorted Linked List	O(n)	O(n)	O(n)
Sorted Array List	O(n)	O(log n)	O(n)

Am I A BST?

Example (Which of the following are BSTs?)



Height of a Binary Tree

Definition (Height)

The **height** of a binary tree is the length of the longest **path** from the root to a leaf.

- Height of an empty tree?

$h = -1$

$h = 0$

$h = 1$

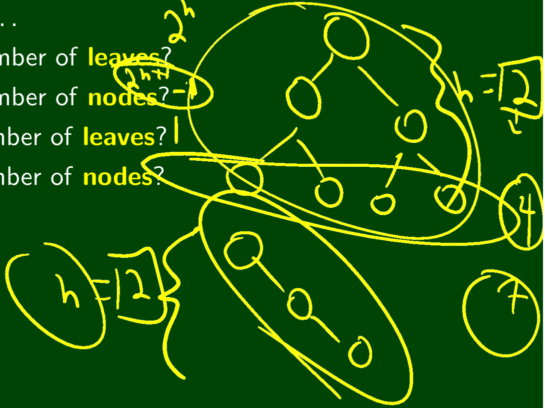
Height

```java
private int height(Node current) {
   if (current == null) { return −1; }
   return 1 + Math.max(height(current.left), height(current.right));
}
```

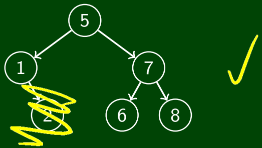
Given that a tree has height h...

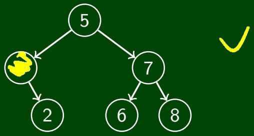
- What is the maximum number of **leaves**? 2^h
- What is the maximum number of **nodes**? $2^{h+1}-1$
- What is the minimum number of **leaves**? 1
- What is the minimum number of **nodes**?

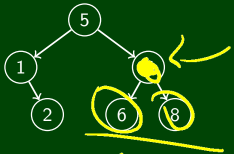
$\sum_{i=0}^{h} 2^i$

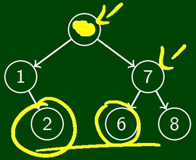


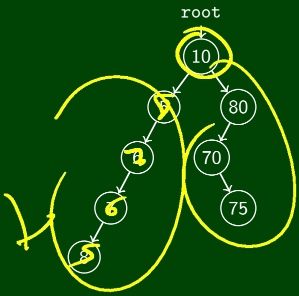
delete from a BST

tree.delete(2)  ⟶

tree.delete(1)  ⟶

tree.delete(7)  ⟶

tree.delete(5)  ⟶



delete(x)

- Case 1: x is a leaf
  - Just delete x
- Case 2: x has one child
  - Replace x with its child
- Case 3: x has two children
  - Replace x with the **successor** or **predecessor** of x

The tricky case is when x has two children. If we think of the BST in sorted array form, to get the successor, we `findMin`(right subtree) (or predecessor is `findMax`(left subtree))

Psuedocode

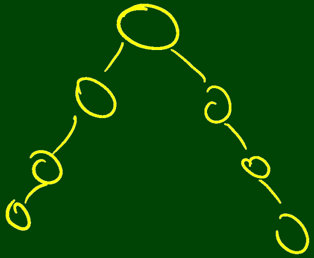
```
void buildTree(int[] input) {
   for (int i = 0; i < input.length; i++) {
      insert(input[i]);
   }
}
```

$O(n)$

What's the best case? The worst case?

Ideas?

- Left and right subtrees of the root have the same number of nodes
- Left and right subtrees of the root have the same **height**

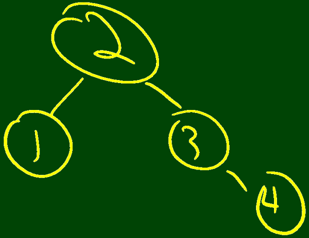


Balance Condition?

Ideas?

- Left and right subtrees ~~of the root~~ **recursively** have the same number of nodes
- Left and right subtrees ~~of the root~~ **recursively** have the same **height**

1, 2, 3, 4



AVL Balance Condition!

Left and right subtrees **recursively** have heights differing by at most one.

Definition (`balance`)

$$\text{balance}(\texttt{n}) = \text{abs}(\text{height}(\texttt{n.left}) - \text{height}(\texttt{n.right}))$$

Definition (AVL Balance Property)

An AVL tree is balanced when:

$$\text{For every node } n, \text{ balance}(n) \leq 1$$

- This ensures a small depth (we'll prove this next time)
- It's relatively easy to maintain (we'll see this next time)