# Lecture 7: Dictionary ADT, BSTs

CSE 332: Data Structures & Parallelism

Yafqa Khan

Summer 2025

# Announcements

- EX 2 Due Friday
- EX 3 Due Next Monday

# Today

- Dictionary ADT
- Review: Binary Search Trees
  - Trees
  - Basics, Properties, Operations
- Balanced BSTs?
- AVL Tree
  - Basics, Properties, Operations

# Today

- Dictionary ADT
- Review: Binary Search Trees
  - Trees
  - Basics, Properties, Operations
- Balanced BSTs?
- AVL Tree
  - Basics, Properties, Operations

# Where we are

**ADTs so far:**

1. Stack: `push`, `pop`, `isEmpty`, etc.
2. Queue: `enqueue`, `dequeue`, `isEmpty`, etc.
3. PriorityQueue: `insert`, `deleteMin`, etc.

**Next:**

4. Dictionary (a.k.a. Map): Associating `keys` with `values` (`k-v` pairs)
   - ONE OF THE MOST IMPORTANT ADTs
   - Also Set

# The Dictionary (a.k.a. Map) ADT

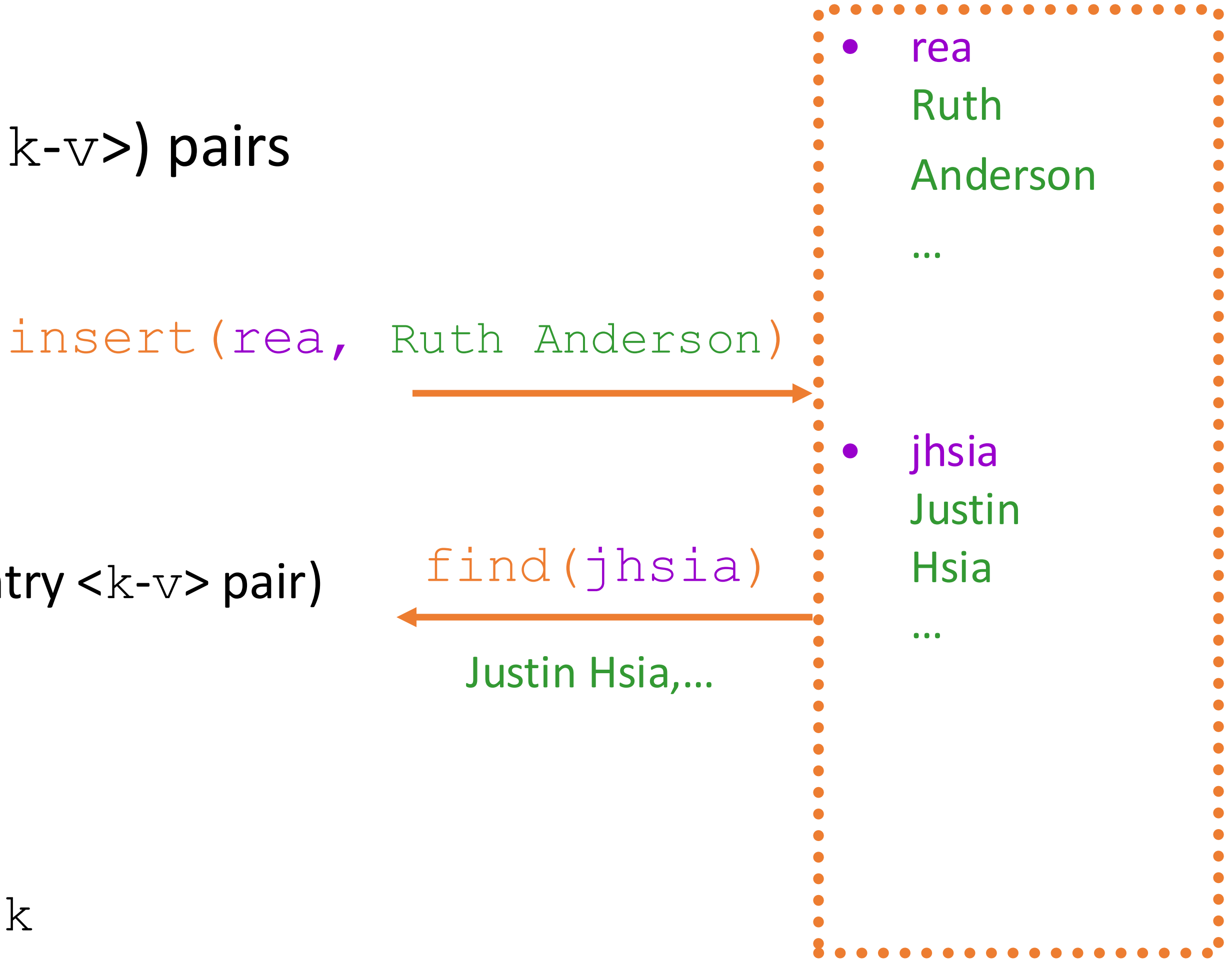
Data:

- Set of <u>unique</u> <`key-value`> (i.e., <`k-v`>) pairs

Operations:

- `insert(k,v)`:
  - places <`k-v`> in map

  (if `k` already used, overwrites existing entry <`k-v`> pair)
- `find(k)`:
  - returns `v` associated with `k`
- `delete(k)`:
  - returns and deletes `v` associated with `k`




We will tend to emphasize the `keys`, but don't forget about the stored `values`!

# Comparison: Set ADT vs. Dictionary ADT

The Set ADT is similar to a Dictionary ADT without any values

- Set: A `key` exists or not (no duplicates)
- Dictionary: A `key` has a `value` or not (no duplicates)

For `find`, `insert`, `delete`, there is little difference

- In Dictionary, values are "just along for the ride"
- So same data structure ideas work for Dictionaries and Sets
  - Java `HashSet` implemented using a `HashMap`, for instance

Set ADT may have other important operations

- `union`, `intersection`, `isSubset`, etc.
- Notice these are binary operators on sets
- We will want different data structures to implement these operators

# Dictionary: Applications

Any time you want to store information according to some key and be able to retrieve it efficiently - Dictionary is the ADT to use!

- Lots of programs do that!

- Networks: router tables
- Operating systems: page tables
- Compilers: symbol tables
- Databases: dictionaries with other nice properties
- Search: inverted indexes, phone directories, …
- Biology: genome maps
- etc...

# Dictionary: Primitive Data Structures

For Dictionary with $n$ unique `k-v` pairs, worst case,

| | `insert` | `find` | `delete` |
|---|---|---|---|
| Unsorted Linked List | $\Theta(\ )$ | $\Theta(\ )$ | $\Theta(\ )$ |
| Unsorted Array | $\Theta(\ )$ | $\Theta(\ )$ | $\Theta(\ )$ |
| Sorted Linked List | $\Theta(\ )$ | $\Theta(\ )$ | $\Theta(\ )$ |
| Sorted Array | $\Theta(\ )$ | $\Theta(\ )$ | $\Theta(\ )$ |

# Dictionary: Primitive Data Structures (Soln.)

For Dictionary with $n$ unique `k-v` pairs, worst case,

| | `insert` | `find` | `delete` |
|---|---|---|---|
| Unsorted Linked List | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| Unsorted Array | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| Sorted Linked List | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| Sorted Array | $\Theta(n)$ | $\Theta(\log n)$ | $\Theta(n)$ |

# Dictionary: Lazy Deletion (e.g., Sorted Array)

| 10 | 12 | 24 | 30 | 41 | 42 | 44 | 45 | 50 |
|---|---|---|---|---|---|---|---|---|
| ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |

`k`=int, `v`=int

boolean "is-it-deleted"

A general technique for making delete as fast as find:

- Don't remove element (i.e., item, `k-v` pairs), just **mark it** as deleted
- No need to shift values

**Advantages**

- Simpler
- Can do removals later in batches
- If re-added soon after, just unmark the deletion

**Disadvantages**

- Extra *space* for the "is-it-deleted" flag
- Data structure full of deleted nodes wastes *space*
- `find` $\Theta(\log m)$ time

($m \geq n$, includes deleted things)

- May complicate other operations

# Dictionary: Better Data Structures

1. AVL Trees
   - Binary Search Trees (BST) with guaranteed balancing
2. HashTables
   - Not tree-like at all

Not in this class: red-black trees, splay trees, B-Trees

# Today

- Dictionary ADT
- Review: Binary Search Trees
  - Trees
  - Basics, Properties, Operations
- Balanced BSTs?
- AVL Tree
  - Basics, Properties, Operations

# Review: Binary Search Tree (BST) Runtime

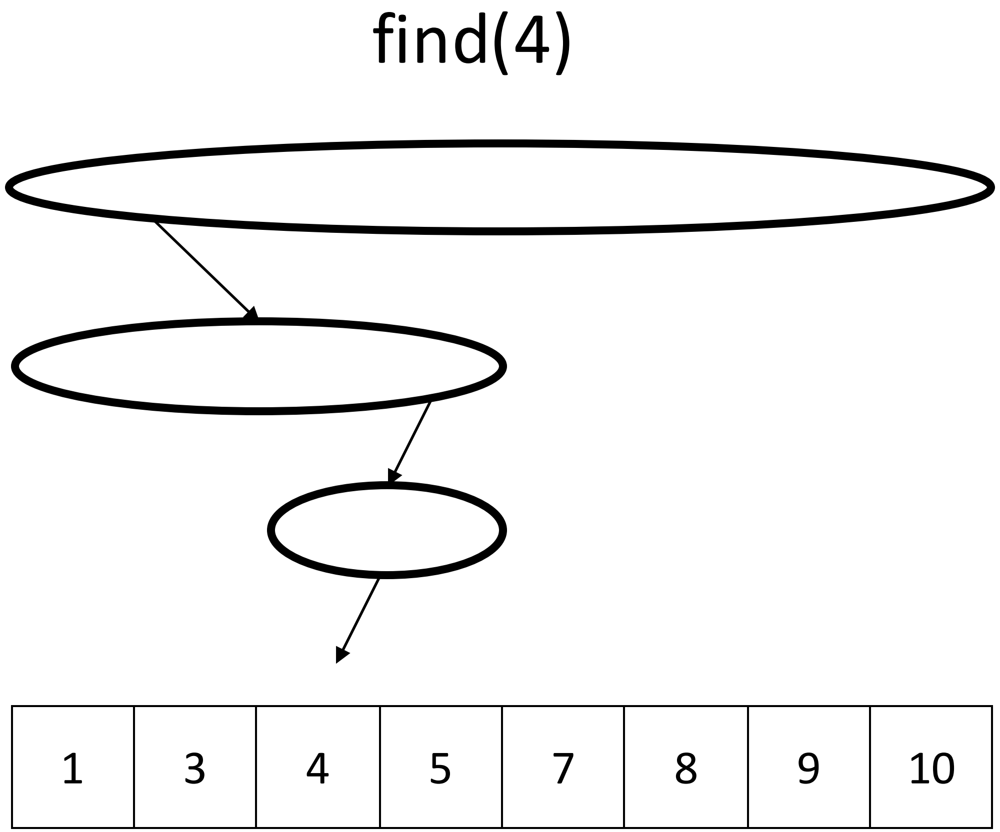
Trees offer speed ups because of their branching factors

- Binary Search Trees are structured forms of binary search

Even a Dictionary as basic as a BST is fairly good

| BST | `insert` | `find` | `delete` |
|---|---|---|---|
| **Worse-Case** | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| **Average-Case** | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ |

# Review: Binary Search



# Review: Binary Tree

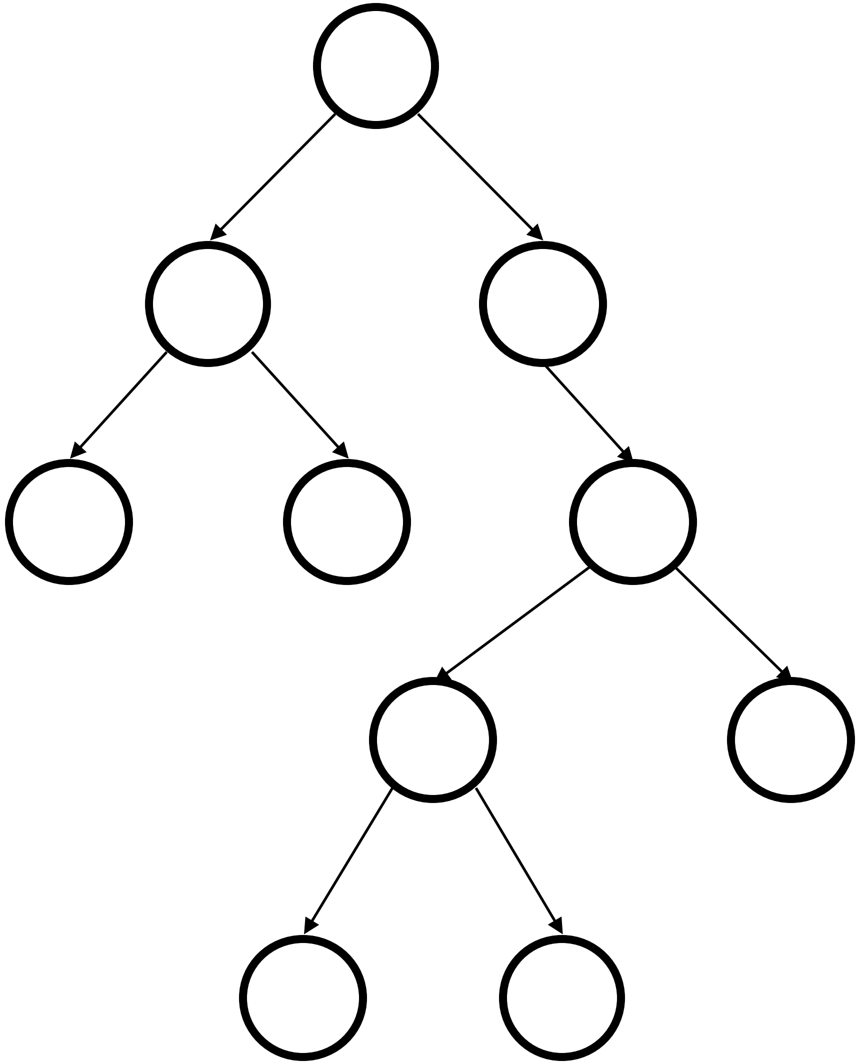
Binary Tree is either:

- A root (with *data*)
- A left subtree (maybe *empty*)
- A right subtree (maybe *empty*)

Representation:

| Data | |
|---|---|
| Left Pointer | Right Pointer |

For a dictionary, *data* will be a `k-v` pair



# Review: Binary Tree Numbers

Remember: Height of a Tree = Longest path from root -> deepest descendent (count # arrows)

For Binary Tree of height $h$:

- Max # of leaves:
- Max # of nodes:
- Min # of leaves:
- Min # of nodes:

# Review: Binary Tree Numbers (Soln.)

Recall: Height of a Tree = Longest path from root -> deepest descendent (count # arrows)

For Binary Tree of height $h$:

- Max # of leaves: $2^h$
- Max # of nodes: $2^{(h+1)} - 1$
- Min # of leaves: $1$
- Min # of nodes: $h + 1$

# Review: Calculating Tree Height

What is the height of a tree with root `root`?

```
int treeHeight(Node root) {

    ???

}
```

Running time for tree with $n$ nodes: $\mathcal{O}(n)$ single pass over tree

# Review: Calculating Tree Height

What is the height of a tree with root `root`?

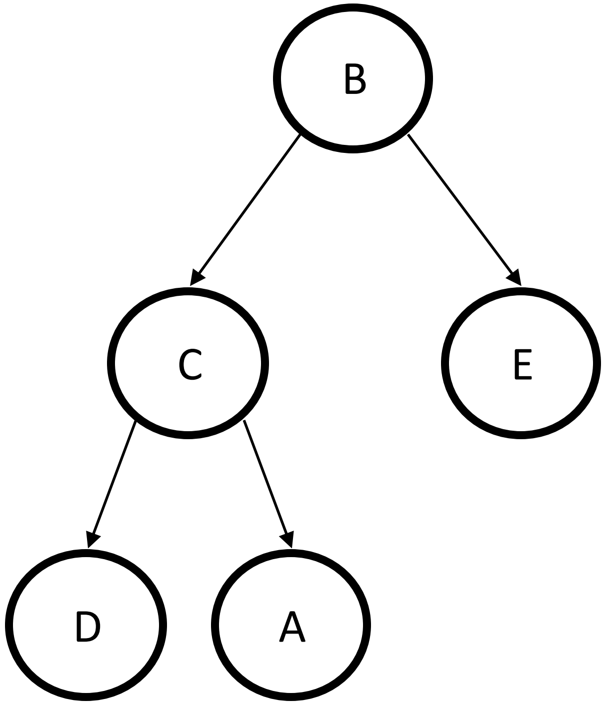
```
int treeHeight(Node root) {
  if(root == null)
      return -1;

  return 1 + max(treeHeight(root.left),
                 treeHeight(root.right));
}
```

Running time for tree with $n$ nodes: $O(n)$ single pass over tree

# Review: Binary Tree Traversals

A traversal is an order for visiting all the nodes of a tree

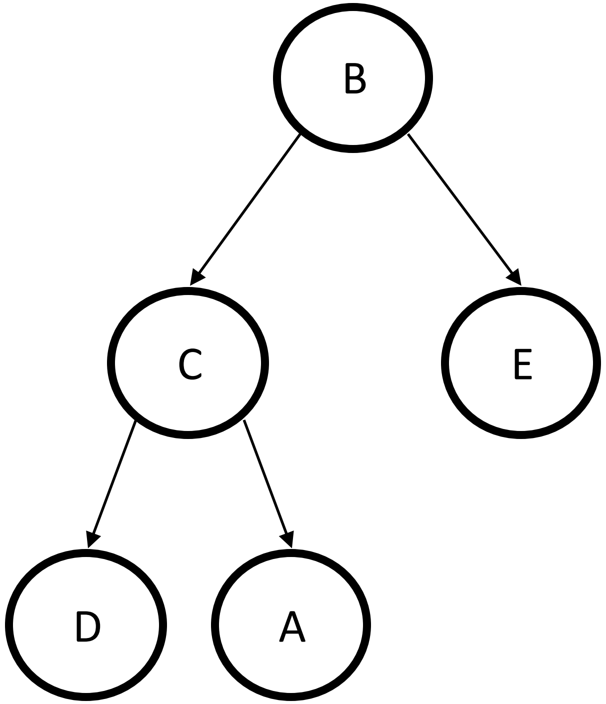
- Pre-order: root, left subtree, right subtree
- In-order: left subtree, root, right subtree
- Post-order: left subtree, right subtree, root



# Review: Binary Tree Traversals (Soln.)

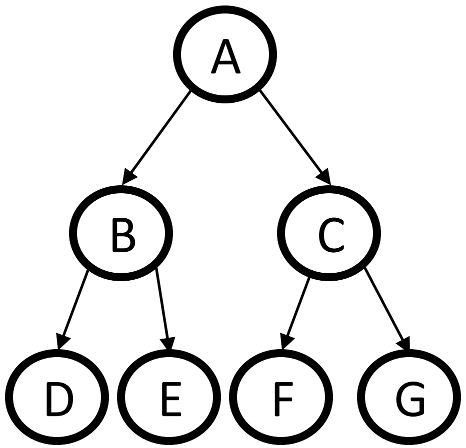
A traversal is an order for visiting all the nodes of a tree

- Pre-order: root, left subtree, right subtree
  - BCDAE
- In-order: left subtree, root, right subtree
  - DCABE
- Post-order: left subtree, right subtree, root
  - DACEB



# Review: More on Binary Tree Traversals

```
void inOrderTraversal(Node root){
  if(root != null) {
    traverse(root.left);
    print(root.data);
    traverse(root.right);
  }
}
```




Sometimes order doesn't matter

- Sum all elements

Sometimes order matters

- Example: print tree with parent above indented children (pre-order)
- Example: evaluate an expression tree (post-order)

# Today

- Recap: Dictionary ADT
- Review: Binary Search Trees
  - Trees
  - Basics, Properties, Operations
- Balanced BSTs?
- AVL Tree
  - Basics, Properties, Operations

# Binary Search Tree: Data Structure

**Structural Property**

- Each node has $\leq 2$ children

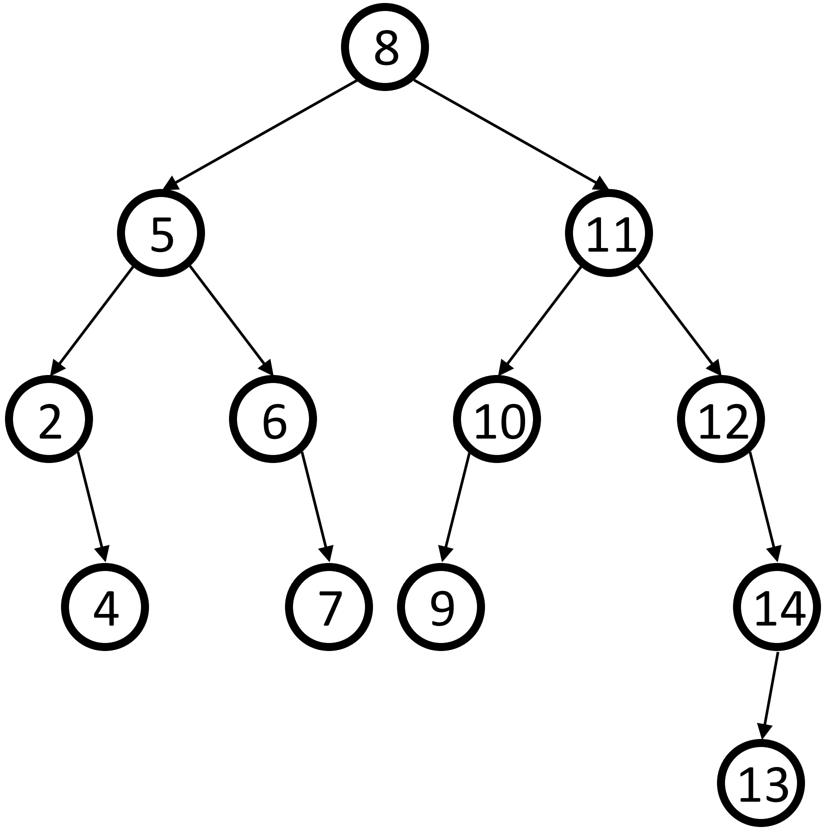
(makes operations simple)

- Result: Simple operations

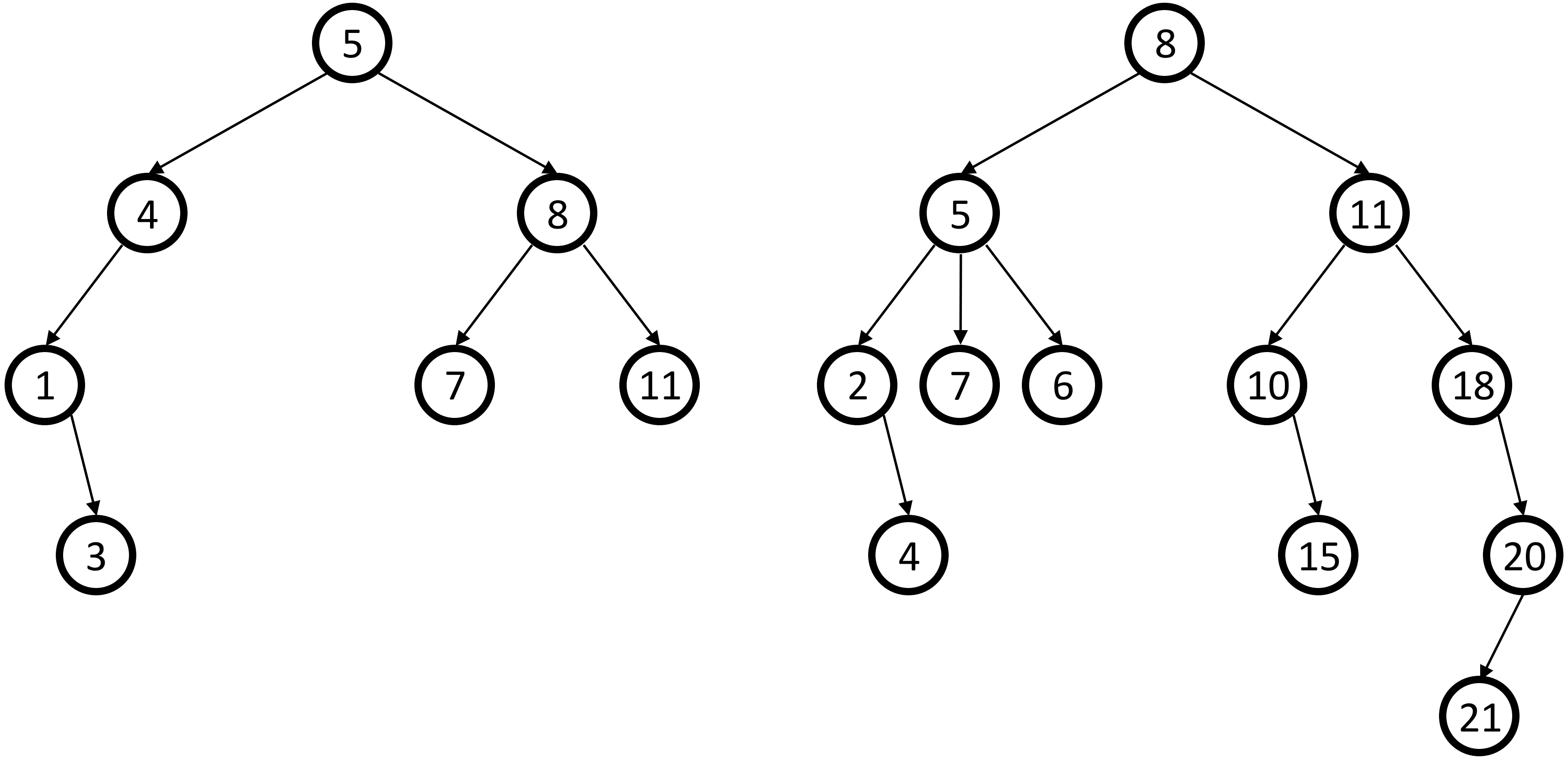
**Order Property**

- All keys in left subtree $<$ node's key
- All keys in right subtree $>$ node's key
- Result: Easy to find a key

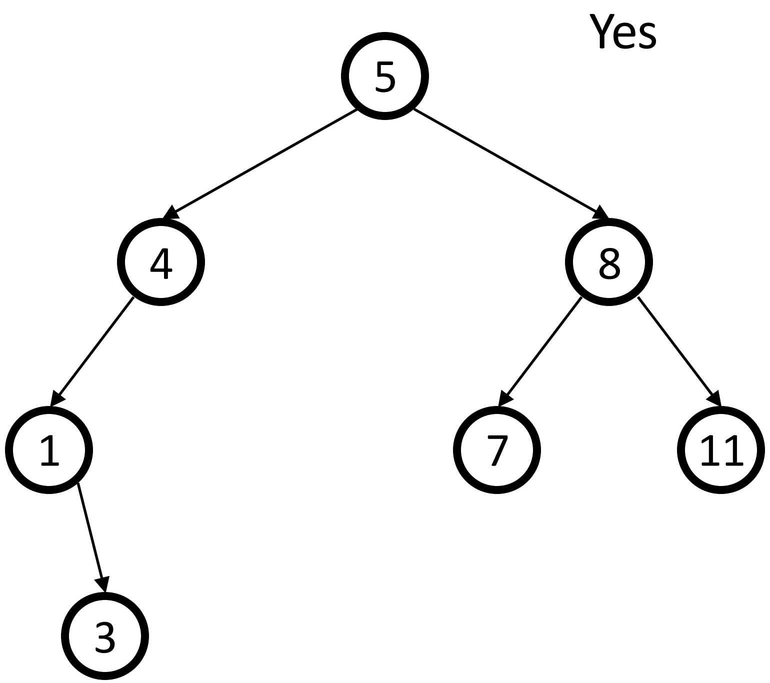
Note: No duplicates

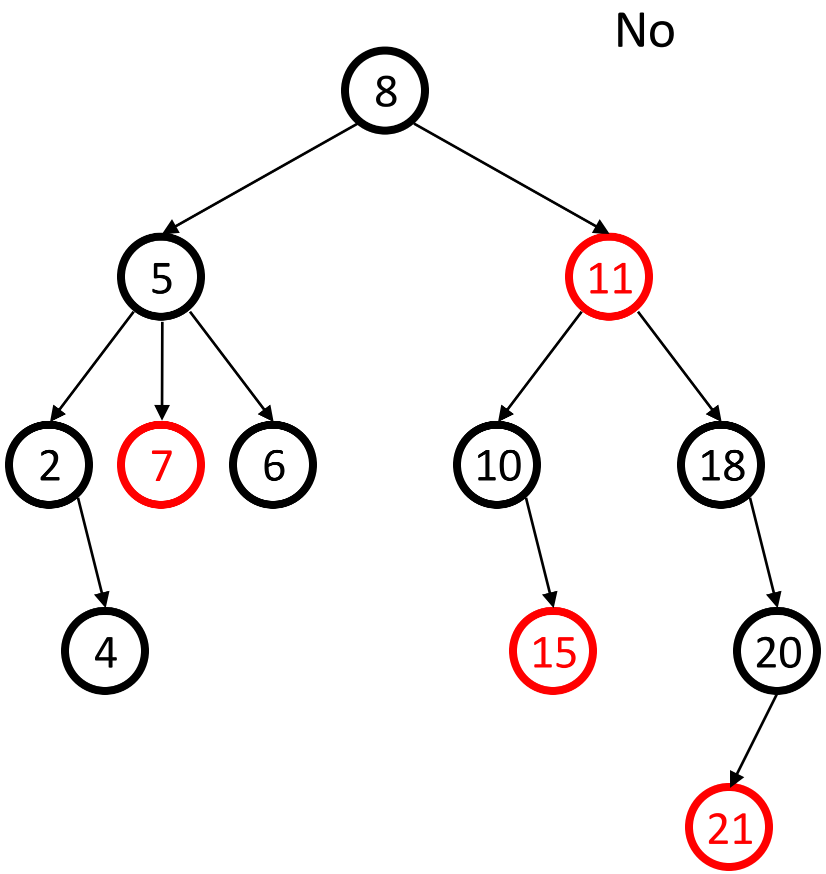


# Are these BSTs?

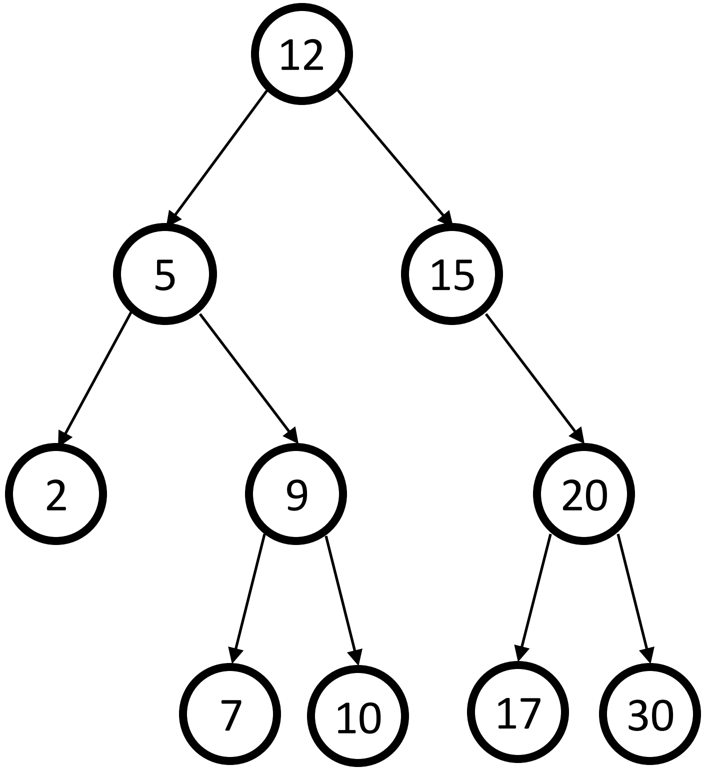


# Are these BSTs? (Soln.)



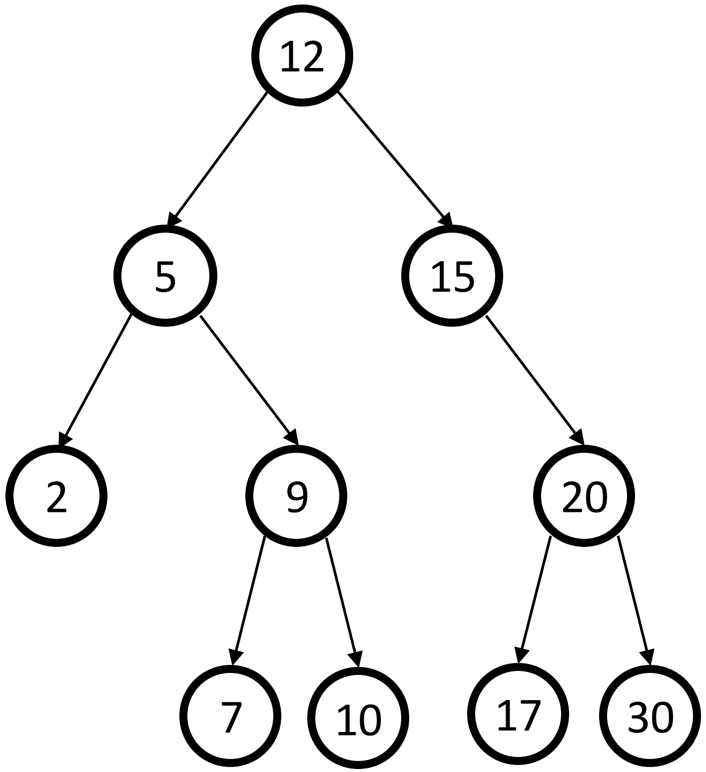



# BSTs: `find`, Recursive




```
Data find(Key key, Node root){
 if(root == null)
   return null;
 if(key < root.key)
   return find(key,root.left);
 if(key > root.key)
   return find(key,root.right);
 return root.data;
}
```

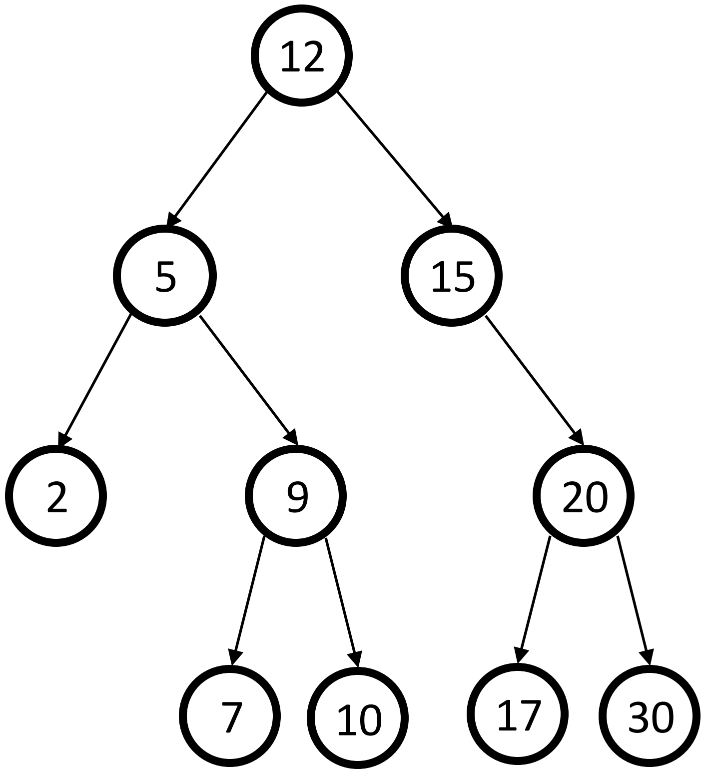
# BSTs: `find`, Iterative ("Harder")



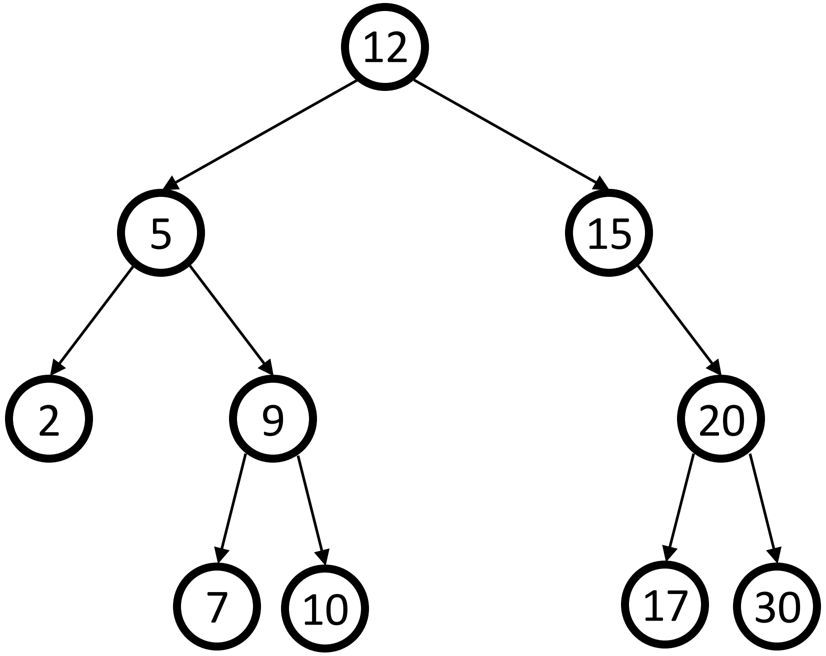

```
Data find(Key key, Node root){
 while(root != null
        && root.key != key) {
  if(key < root.key)
    root = root.left;
  else(key > root.key)
    root = root.right;
 }
 if(root == null)
    return null;
 return root.data;
}
```

# BSTs: Other "find" operations

- Find minimum node?
  - Go left
- Find maximum node?
  - Go right



# BSTs: `insert`




`insert(13)`

`insert(8)`

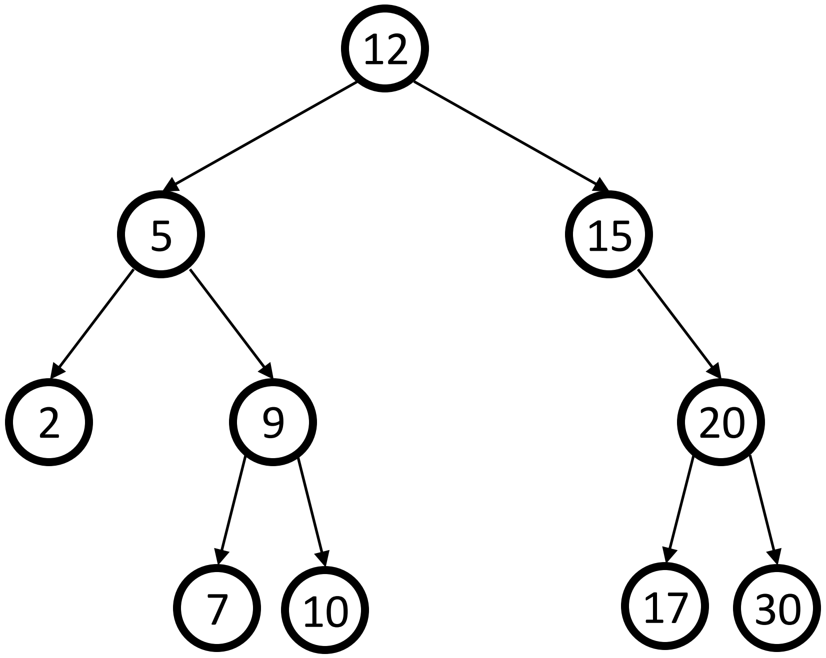
`insert(31)`

Each `insert` is inserting a leaf node

1. `find`
2. Create new (leaf) node

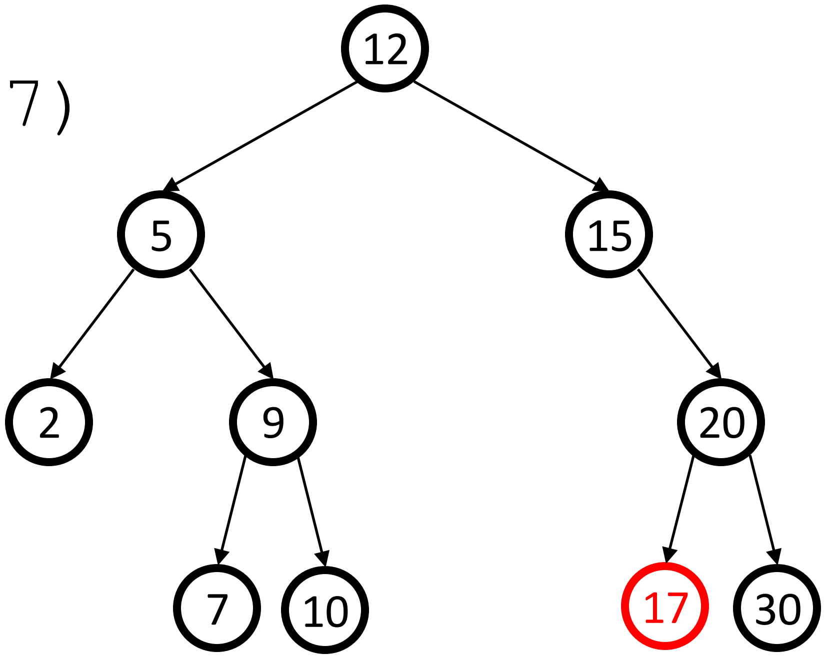
# BSTs: `delete`

How?



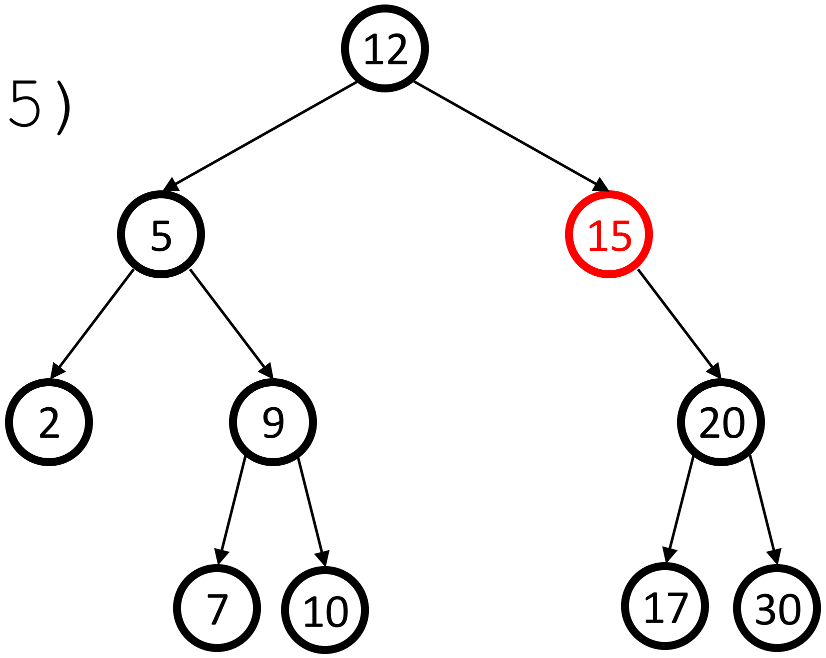
# BSTs: `delete` - Case 1: Leaf

`delete(17)`



# BSTs: `delete` - Case 2: One Child

`delete(15)`



# BSTs: `delete` - Case 3: Two Child

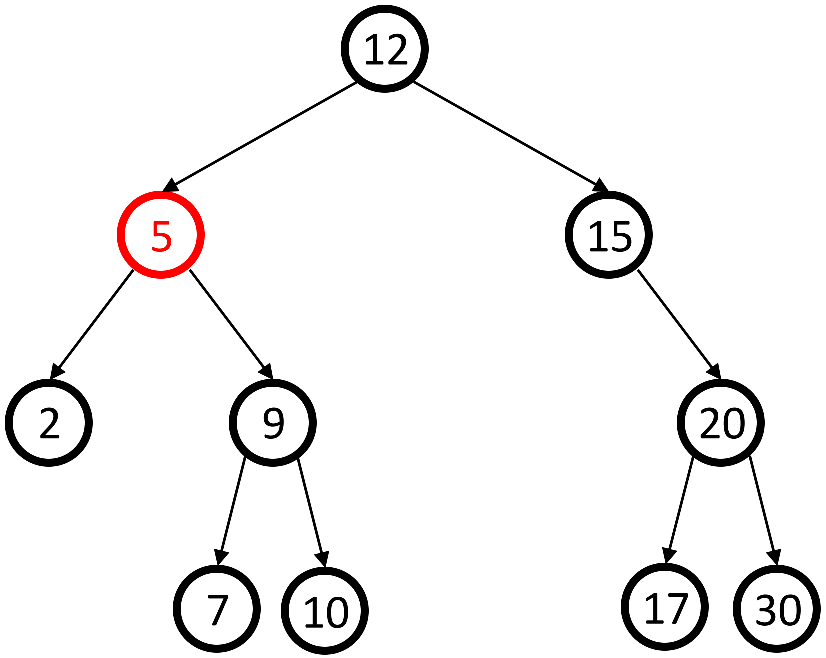
**`findMax(5.left)`**

**`swap and delete`**

or

**`findMin(5.right)`**

**`swap and delete`**




What can we replace 5 with?

- Largest element on the left subtree (called predecessor)
- Smallest element on the right subtree (called successor)

# BSTs: `delete`

Basic Idea:

- `find`
- remove (which can break tree structure)
- "fix" Structure and Order

3 Cases:

1. Leaf
2. One Child
3. Two Child

# Any Questions?

# Today

- Recap: Dictionary ADT
- Review: Binary Search Trees
  - Trees
  - Basics, Properties, Operations
- Balanced BSTs?
- AVL Tree
  - Basics, Properties, Operations

# BSTs: Balancing?

Observation

- Worst case: $\Theta(n)$ (though "average" its $\Theta(\log n)$)
- Shorter = better runtime (i.e., Taller = worse runtime)

Solution: **Balancing**

- Always ensure `root` height is $\Theta(\log n)$
- Efficient to maintain

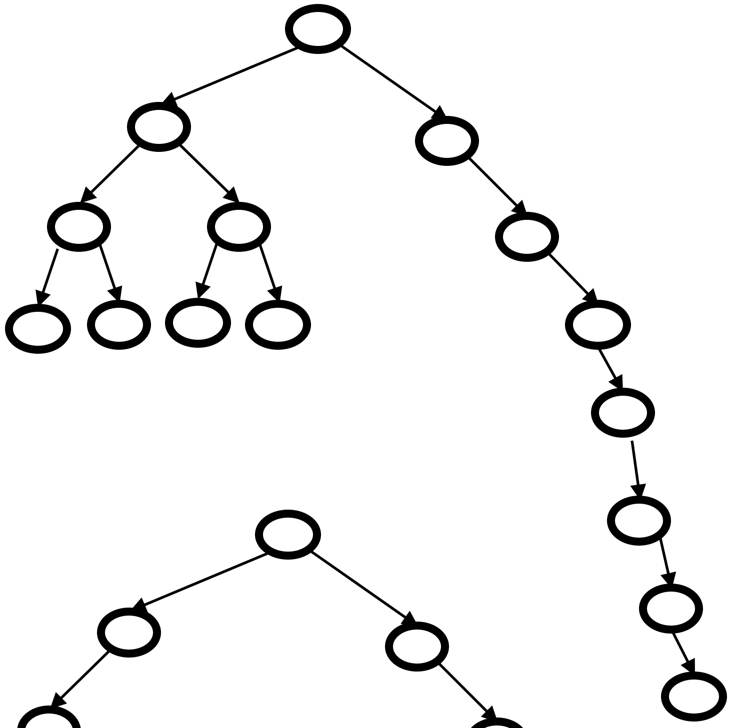
# BSTs: Potential Balance Conditions 1

1. Left subtree and right subtree of *just* `root` **same # of nodes**

2. Left subtree and right subtree of *just* `root` **same height**

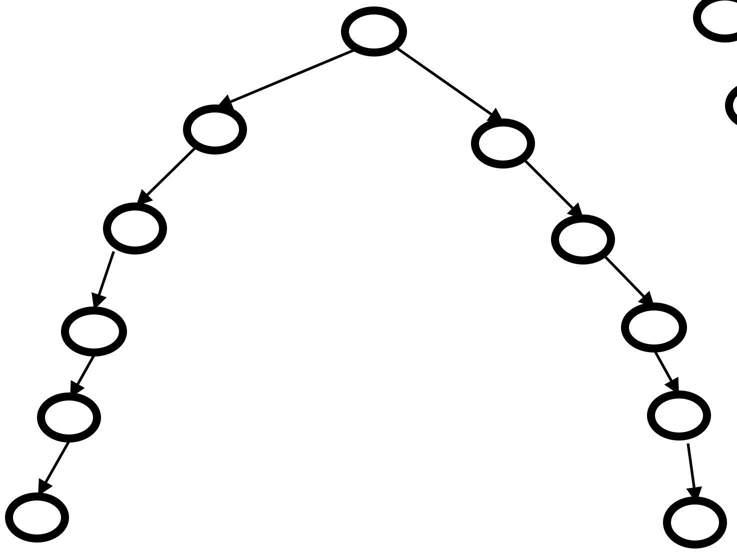
# BSTs: Potential Balance Conditions 1 (Soln.)

1. Left subtree and right subtree of *just* `root` **same # of nodes**



Too Weak!

2. Left subtree and right subtree of *just* `root` **same height**



Too Weak!

# BSTs: Potential Balance Conditions 2

1. Left subtree and right subtree of *every* `node` **same # of nodes**

2. Left subtree and right subtree of *every* `node` **same height**

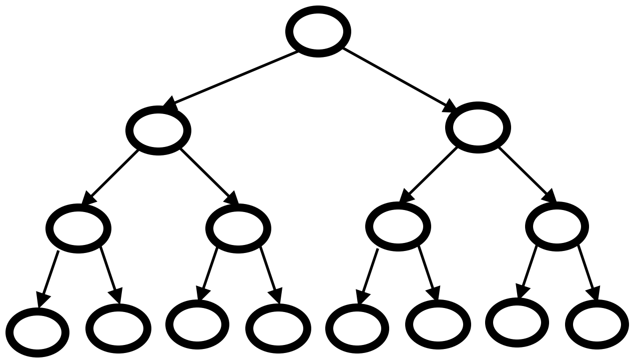
# BSTs: Potential Balance Conditions 2 (Soln.)

1. Left subtree and right subtree of *every* `node` **same # of nodes**

Too Strong!

2. Left subtree and right subtree of *every* `node` **same height**

Too Strong!



Only Perfect Trees Allowed :(

# BSTs: AVL Balance Condition

Left subtree and right subtree of *every* `node` **heights differ by at most 1**

**AVL Balance Property:**

`balance(node)=height(node.left)-height(node.right)`

For every node, $-1 \leq$ `balance(node)` $\leq 1$

- Always ensure `root` height is $\Theta(\log n)$
- Efficient to maintain
  - $\Theta(1)$ rotations

# Timeline

- Dictionary ADT
- Review: Binary Search Trees
  - Trees
  - Basics, Properties, Operations
- Balanced BSTs?
- AVL Tree
  - Basics, Properties, Operations
- AVL Tree `insert`
  - Single Rotation
  - Double Rotation
- AVL Tree Conclusions