UNIVERSITY OF WASHINGTON · 1861

CSE332: Data Abstractions

Lecture 6: Dictionaries; Binary Search Trees

Tyler Robison

Summer 2010

Where we are

ADTs so far:

1. Stack: `push`, `pop`, `isEmpty` **LIFO**
2. Queue: `enqueue`, `dequeue`, `isEmpty` **FIFO**
3. Priority queue: `insert`, `deleteMin` **Min**

Next:

4. Dictionary: associate keys with values
    - probably the most common, way more than priority queue
    - Ex: Binary Search Tree, HashMap

The Dictionary (a.k.a. Map, a.k.a. Associative Array) ADT

- Data:
  - *set* of (key, value) *pairs*
  - keys must be *comparable (< or > or =)*
- Primary Operations:
  - `insert(key,val): places (key,val) in map`
    - **`If key already used, overwrites existing entry`**
  - `find(key): returns val associated with key`
  - `delete(key)`

Comparison: Set ADT vs. Dictionary ADT

The *Set* ADT is like a Dictionary without any values

- A key is *present* or not (no repeats)

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`**, **`is_subset`**
- notice these are operators on 2 sets

Dictionary data structures

Will spend the next week or two looking at three important dictionary data structures:

1. AVL trees
   - Binary search trees with *guaranteed balancing*
2. B-Trees
   - Also always balanced, but different and shallower
   - B!=Binary; B-Trees generally have large branching factor
3. Hashtables
   - Not tree-like at all

Skipping: Other balanced trees (red-black, splay)

A Modest Few Uses

Any time you want to store information according to some key and be able to retrieve it efficiently

- Lots of programs do that!

- Networks: router tables
- Compilers: symbol tables
- Databases, phone directories, associating username with profile, …

Some possible data structures

Worst case for dictionary with *n* key/value pairs

	`insert`	`find`	`delete`
Unsorted linked-list	$O(1)$*	$O(n)$	$O(n)$
Unsorted array	$O(1)$*	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$	$O(n)$

We'll see a Binary Search Tree (BST) probably does better…

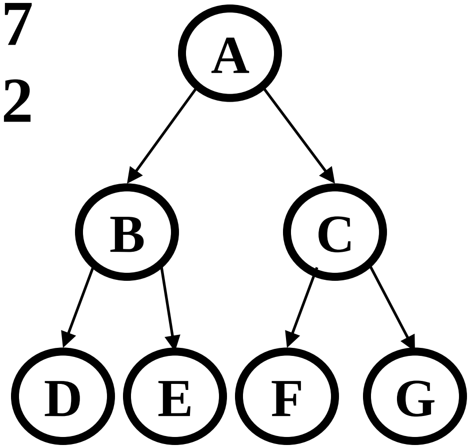
But not in the worst case unless we keep it balanced

*Correction: Given our policy of 'no duplicates', we would need to do O(n) work to check for a key's existence before insertion

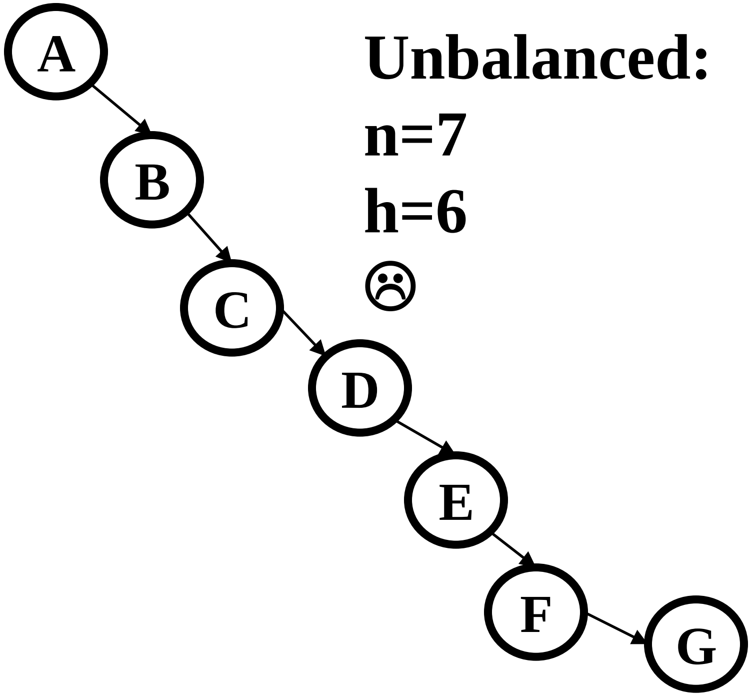
Some tree terms (review… again)

- A tree can be balanced or not
  - A balanced tree with *n* nodes has a height of *O*(`log` *n*)
  - Different tree data structures have different "balance conditions" to achieve this

Balanced:
n=7
h=2
☺



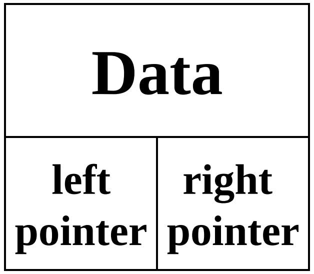
Unbalanced:
n=7
h=6
☹



Binary Trees

- Binary tree is empty or
  - a node *(with data)*, and with
    - a left subtree *(maybe empty)*
    - a right subtree *(maybe empty)*

- Representation:

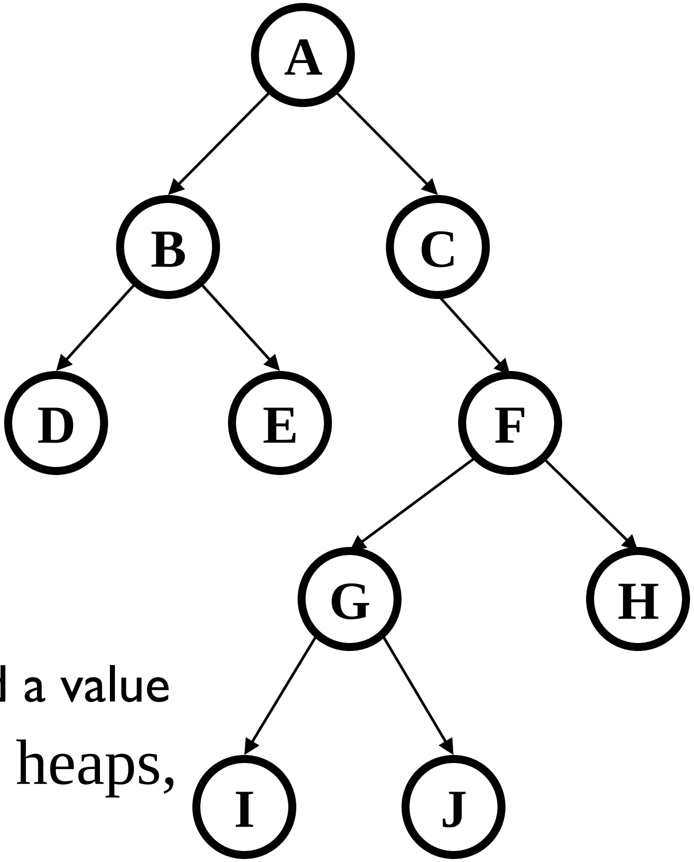


Data

left pointer | **right pointer**

- For a dictionary, data will include key and a value

Ditched this representation for binary heaps, but it's useful for BST



Binary Trees: Some Numbers

Recall: height of a tree = longest path from root to leaf (counting # of edges)

Operations tend to be a function of height

For binary tree of height *h*:

- max # of leaves: $\mathbf{2^{h}}$
- max # of nodes: $\mathbf{2^{(h+1)} - 1}$
- min # of leaves: **1**
- min # of nodes: **h+1**

For n nodes, we cannot do better than O(`log` n) height, and we want to avoid O(n) height

Calculating height

How do we find the height of a tree with root **`r`**?

```
int treeHeight(Node root) {

              ???



}
```

Calculating height

How do we find the height of a tree with root **`r`**?

```
int treeHeight(Node root) {
  if(root == null)
    return -1;
  return 1 + max(treeHeight(root.left),
                 treeHeight(root.right));
}
```

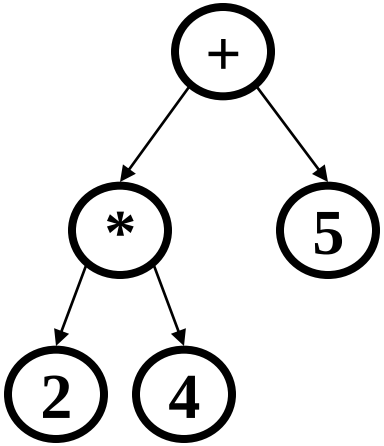
Running time for tree with *n* nodes: *O*(*n*) – single pass over tree

Note: non-recursive is painful – need your own stack of pending nodes; much easier to use recursion's call stack

Tree Traversals

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

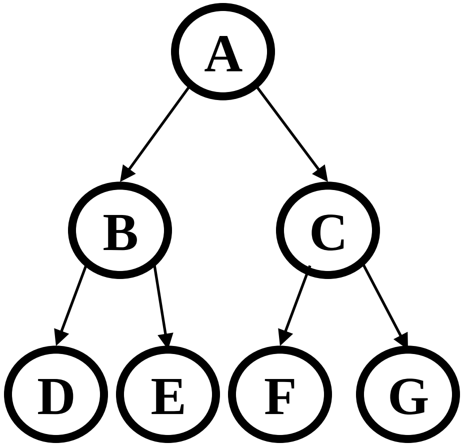
Expression tree



- *Pre-order*: root, left subtree, right subtree
  +*245
- *In-order*: left subtree, root, right subtree
  2*4+5
- *Post-order*: left subtree, right subtree, root
  24*5+

More on traversals

```
void inOrdertraversal(Node t){
  if(t != null) {
    traverse(t.left);
    process(t.element);
    traverse(t.right);
  }
}
```



Sometimes order doesn't matter

- Example: sum all elements

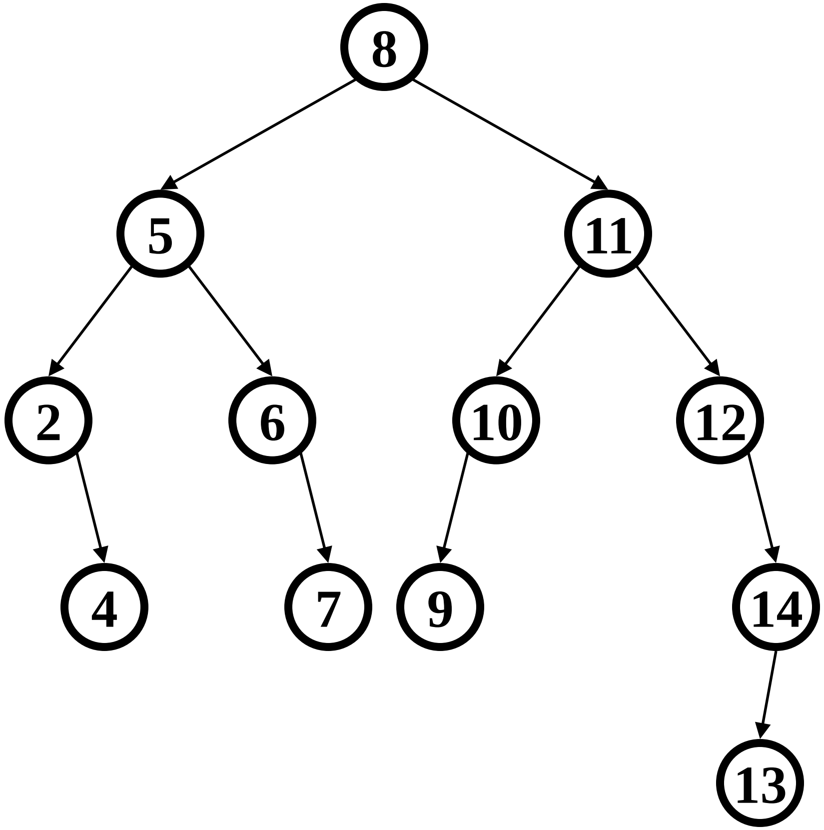
Sometimes order matters

- Example: print tree with parent above indented children (pre-order)
- Example: print BST values in order (in-order)

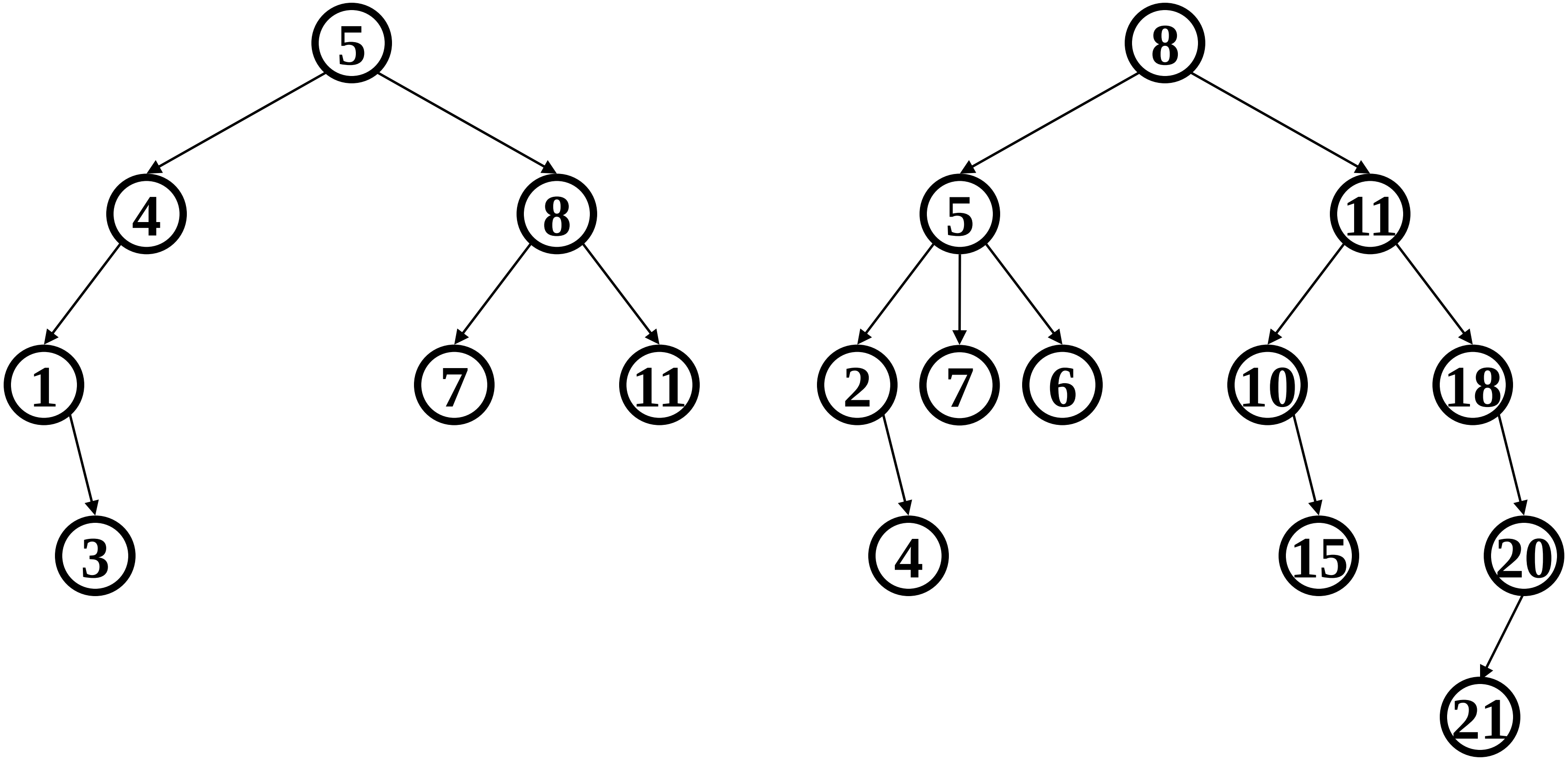
```
A
  B
    D
    E
  C
    F
    G
```

Binary Search Tree

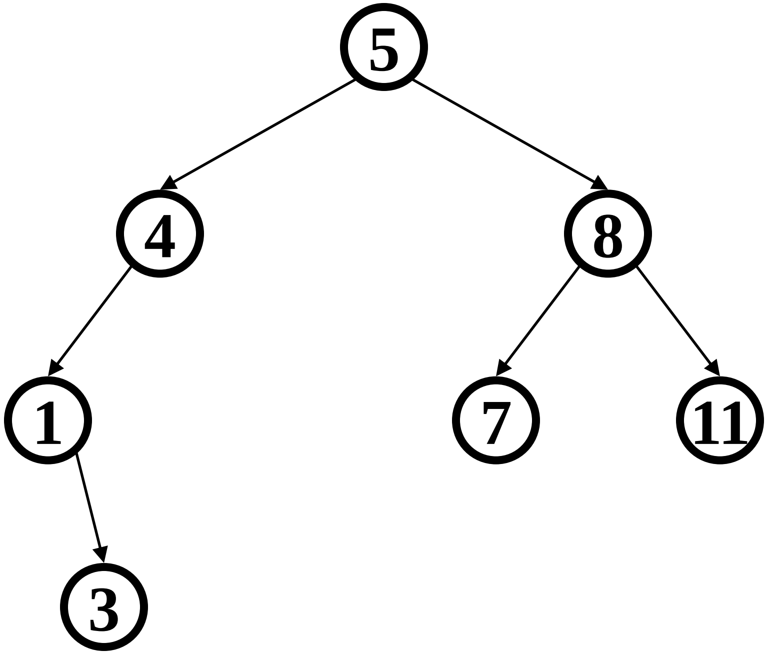
- Structural property ("binary")
  - each node has ≤ 2 children
- Order property
  - all keys in left subtree smaller than node's key
  - all keys in right subtree larger than node's key
  - result: easy to find any given key



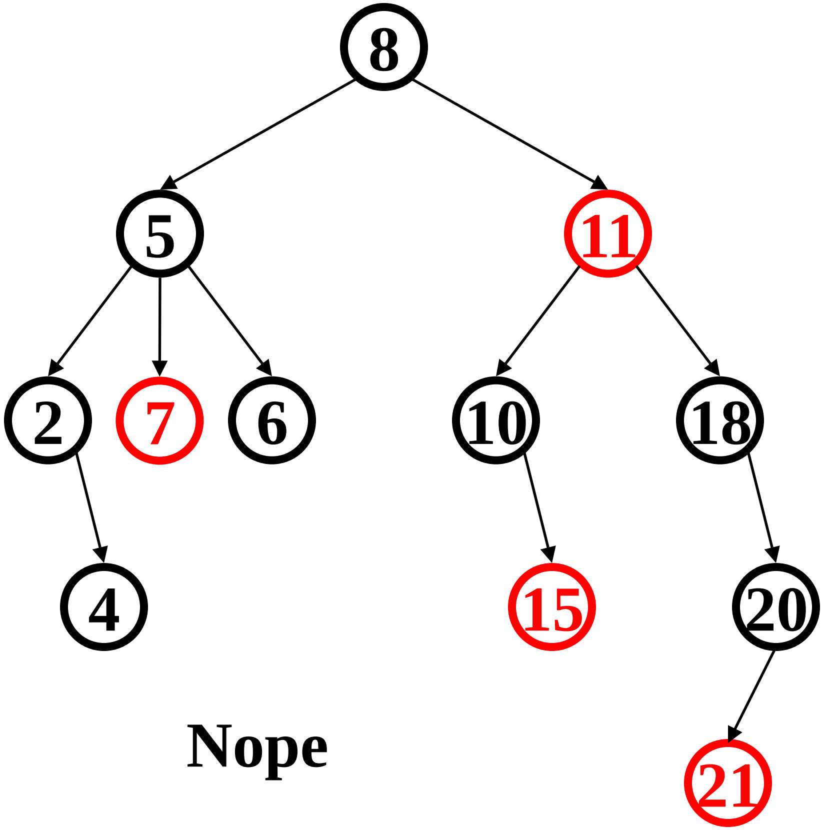
Are these BSTs?



Are these BSTs?

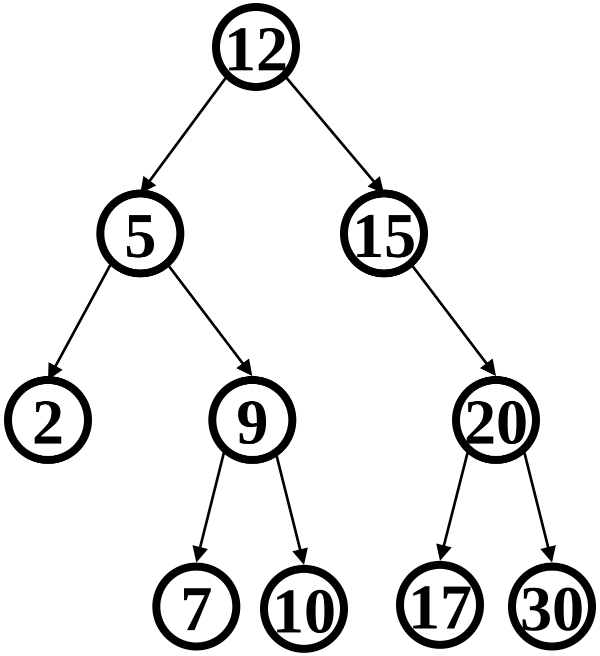


Yep



Nope

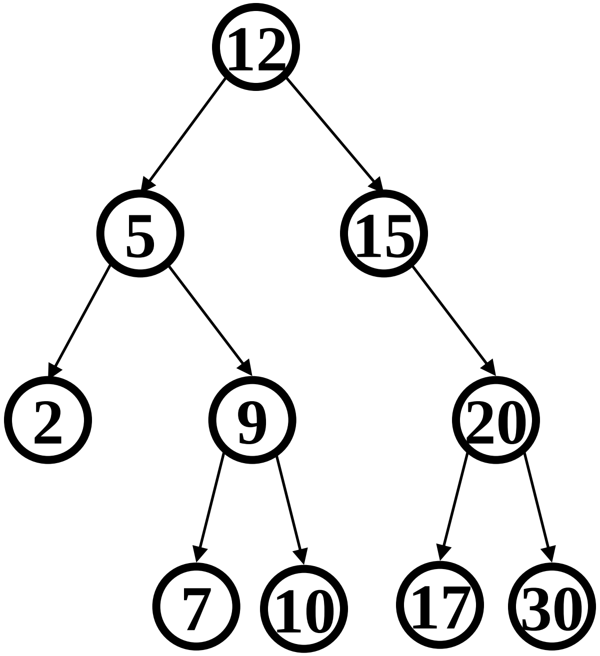
Find in BST, Recursive



```
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;
}
```

Run-time (for worst-case)?

Find in BST, Iterative



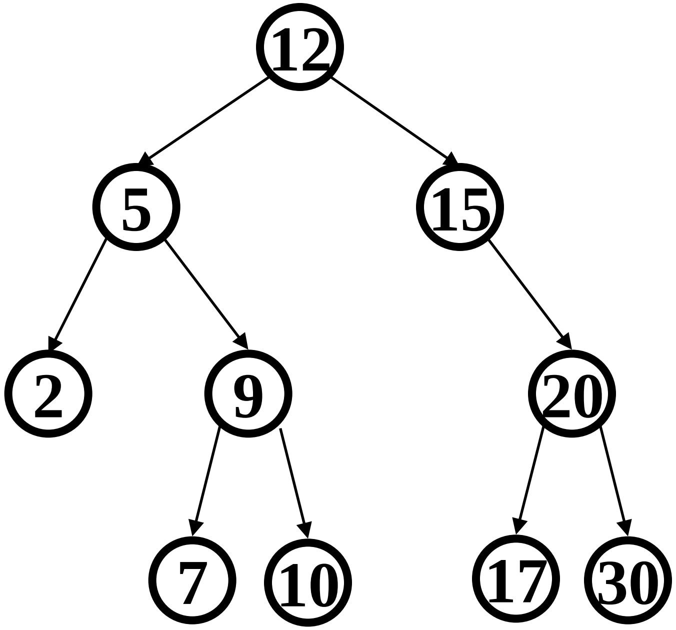
```
Data find(Key key, Node root){
 while(root != null
        && root.key != key) {
  if(key < root.key)
    root = root.left;
  else if(key > root.key)
    root = root.right;
 }
 if(root == null)
    return null;
 return root.data;
}
```

For iteratively calculating height & doing traversals, we needed a stack. Why do we not need one here?

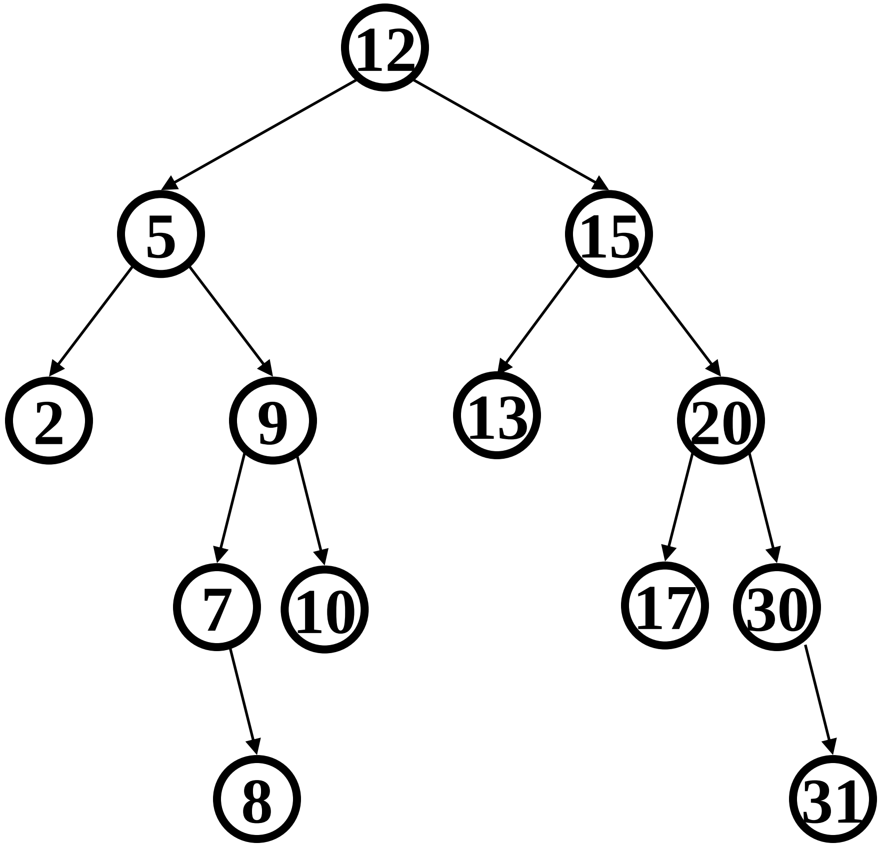
Other "finding operations"

- Find *minimum* node
- Find *maximum* node

- Find *predecessor*
- Find *successor*



Insert in BST



```
insert(13)
insert(8)
insert(31)
```

How do we insert k elements to get a completely unbalanced tree?

How do we insert k elements to get a balanced tree?

Lazy Deletion

10	**12**	**24**	**30**	**41**	**42**	**44**	**45**	**50**
✓	✗	✓	✓	✓	✓	✗	✓	✓

A *general technique* for making **`delete`** as fast as **`find`**:

- Instead of actually removing the item just mark it deleted

"Uh, I'll do it later"

Plusses:

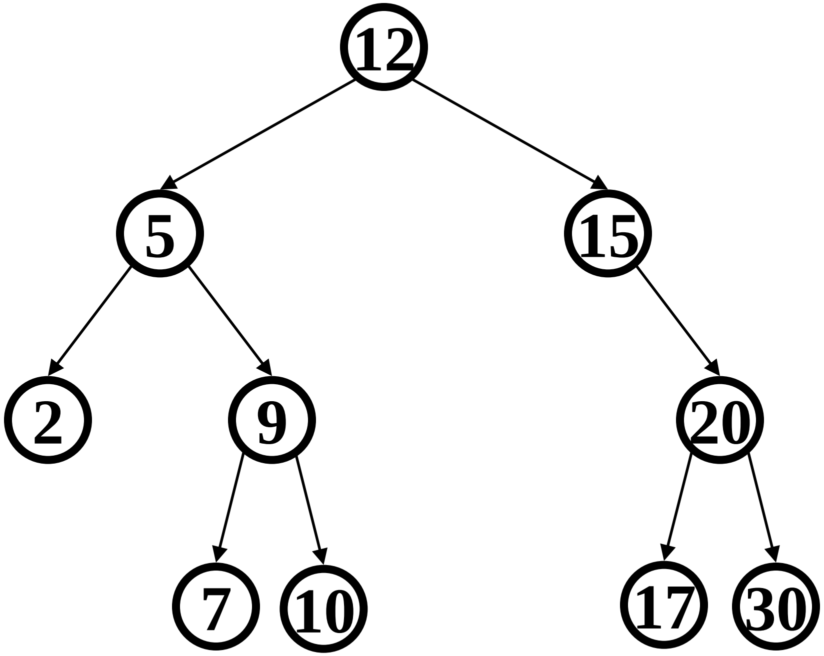
- Simpler
- Can do removals later in batches
- If re-added soon thereafter, just unmark the deletion

Minuses:

- Extra *space* for the "is-it-deleted" flag
- Data structure full of deleted nodes wastes *space*
- Can hurt run-times of other operations

We'll see lazy deletion in use later

(Non-lazy) Deletion in BST



Why might deletion be harder than insertion?

Deletion

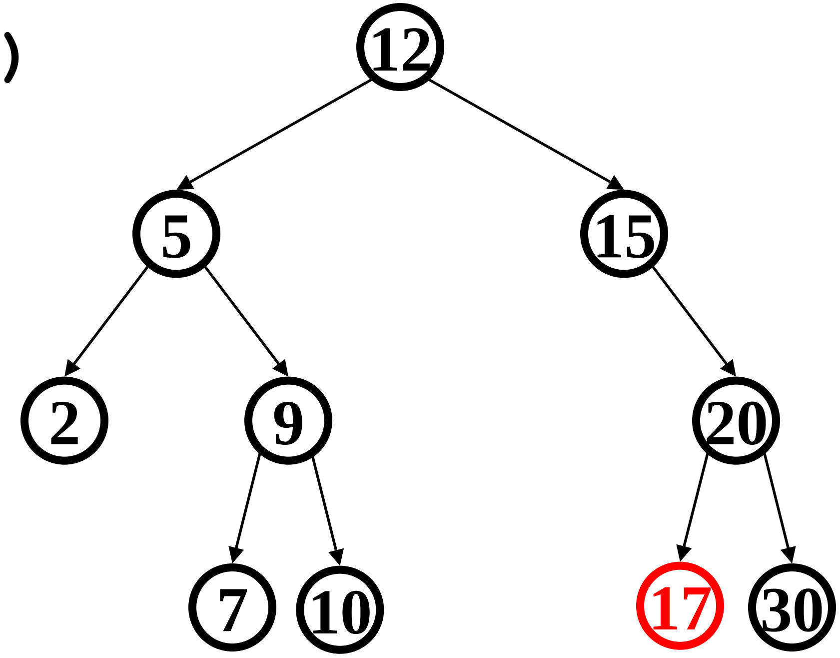
- Removing an item disrupts the tree structure

- Basic idea: `find` the node to be removed, then “fix” the tree so that it is still a binary search tree

- Three cases:
  - node has no children (leaf)
  - node has one child
  - node has two children

Deletion – The Leaf Case

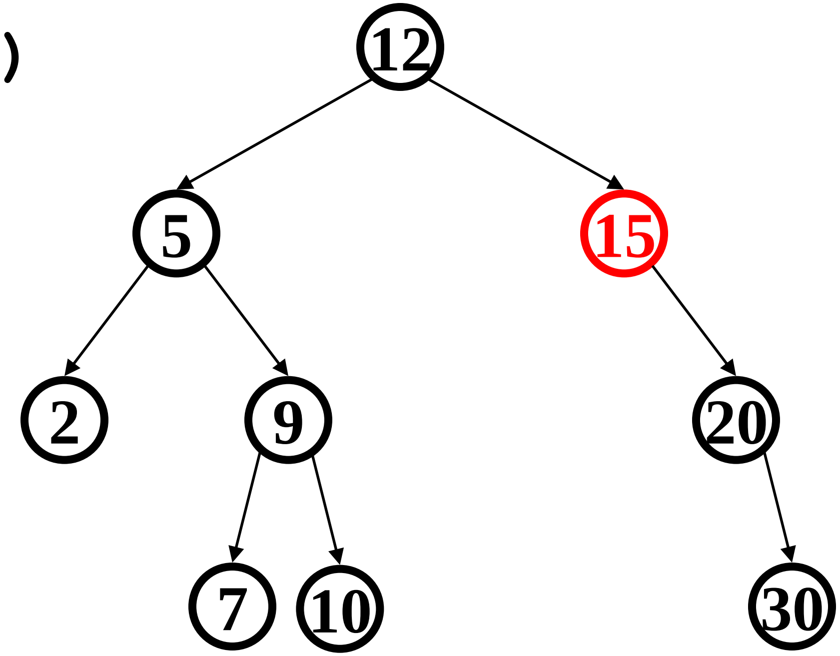
`delete(17)`



Just remove it

Deletion – The One Child Case

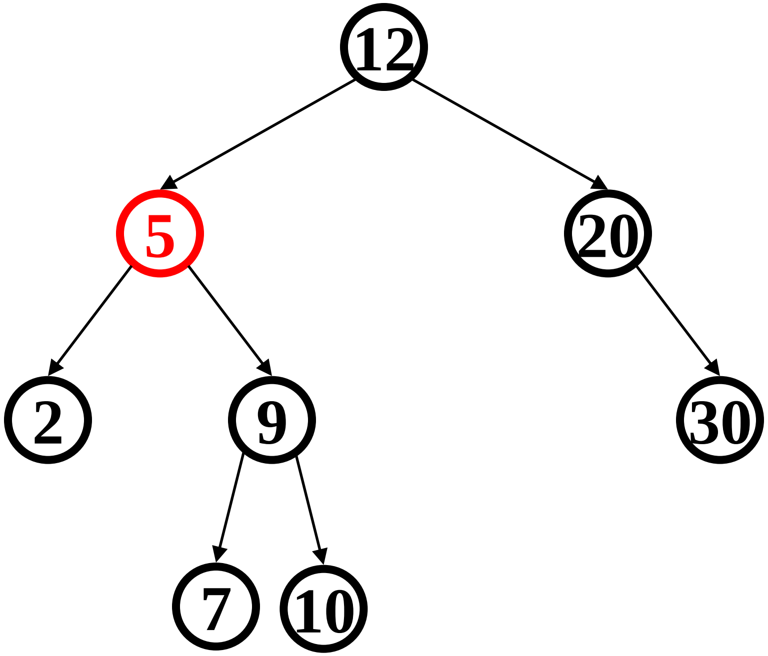
`delete(15)`



Replace it with its child

Deletion – The Two Child Case

`delete(5)`



What can we replace 5 with?

Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees

Options:

- *successor* from right subtree: **`findMin(node.right)`**
- *predecessor* from left subtree: **`findMax(node.left)`**
  - These are the easy cases of predecessor/successor

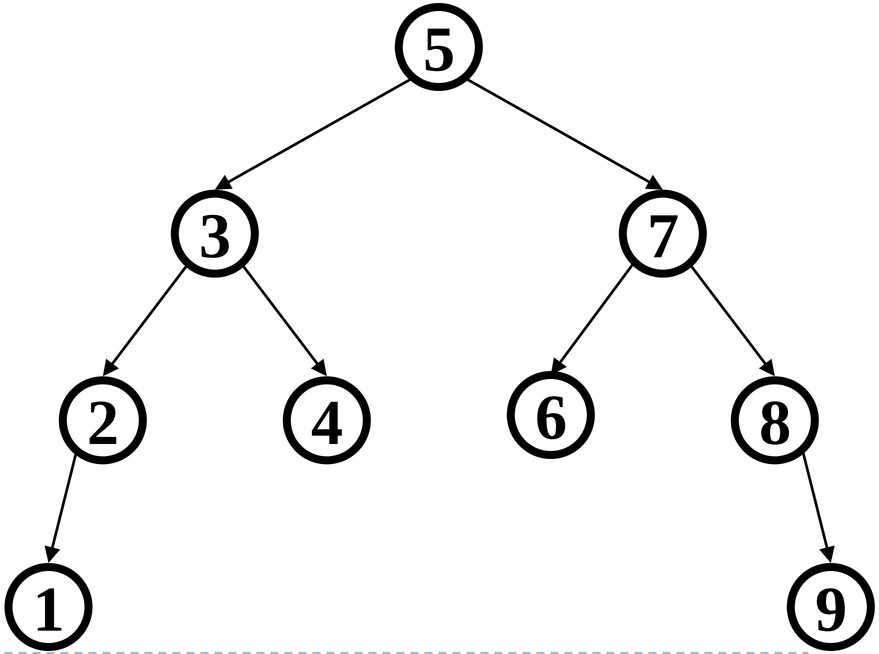
Now delete the original node containing *successor* or *predecessor*

- Leaf or one child case – easy cases of delete!

BuildTree for BST

- BuildHeap equivalent for trees
- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
- In order (and reverse order) not going to work well
- Try a different ordering
  - median first, then left median, right median, etc.
  - 5, 3, 7, 2, 1, 4, 8, 6, 9
  - What tree does that give us?
  - What big-O runtime?

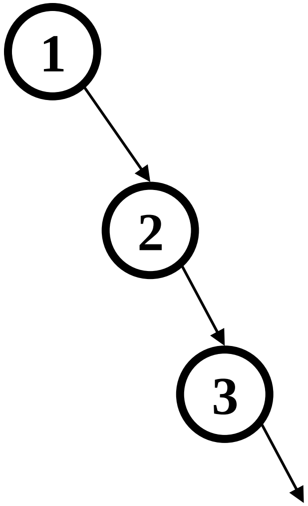
O(n log n), definitely better



Unbalanced BST

- Balancing a tree at build time is insufficient, as sequences of operations can eventually transform that carefully balanced tree into the dreaded list

- At that point, everything is *O*(*n*) ☹
  - `find`
  - `insert`
  - `delete`



Balanced BST

Observation

- BST: the shallower the better!
- For a BST with *n* nodes inserted in arbitrary order
  - Average height is $O(\log n)$ – see text for proof
  - Worst case height is $O(n)$
- Simple cases such as inserting in key order lead to the worst-case scenario

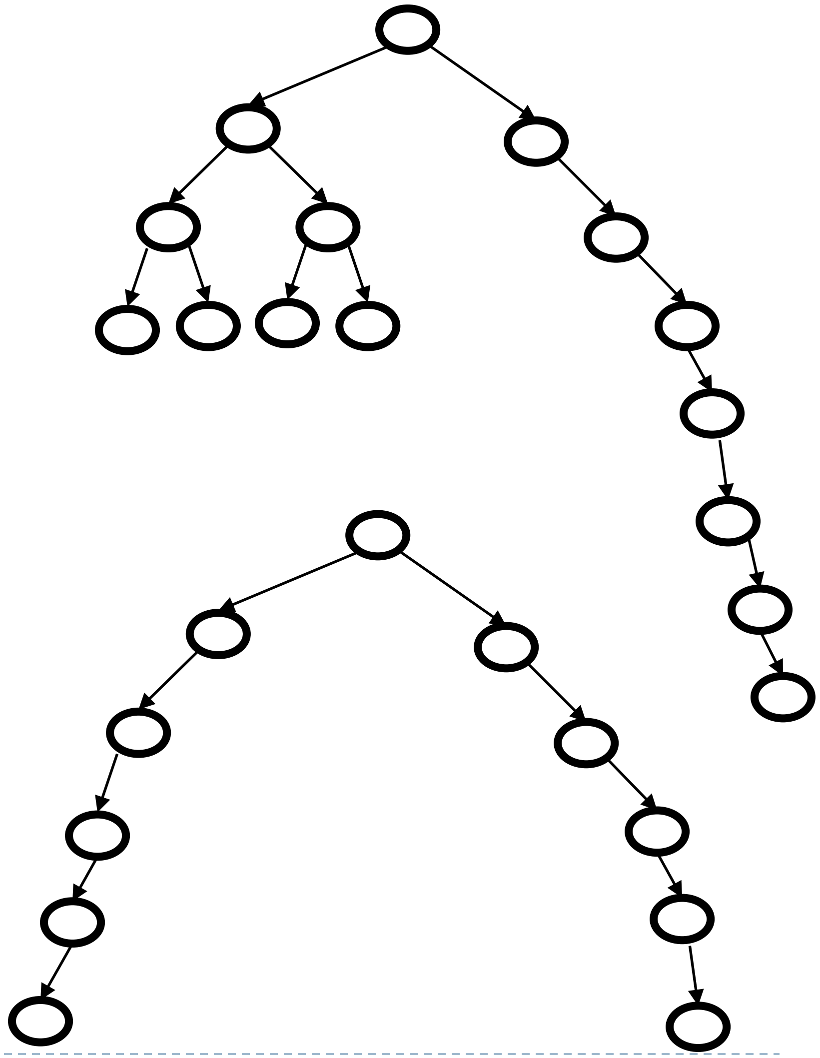
Solution: Require a **Balance Condition** that

1. ensures depth is always $O(\log n)$ – strong enough!
2. is easy to maintain – not too strong!

Potential Balance Conditions

1. Left and right subtrees of the *root* have equal number of nodes

   Too weak!
   Height mismatch example:



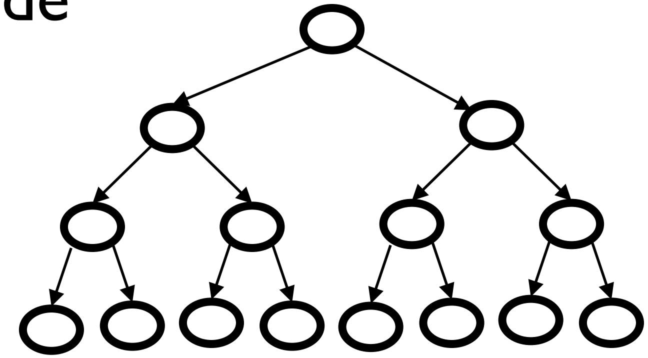
2. Left and right subtrees of the *root* have equal *height*

   Too weak!
   Double chain example:

Potential Balance Conditions

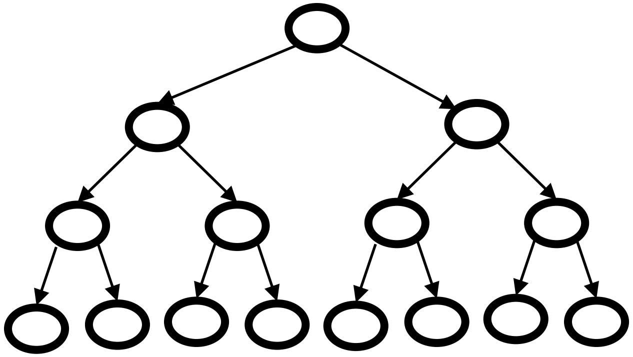
3. Left and right subtrees of every node have equal number of nodes

Too strong!
Only perfect trees ($2^n - 1$ nodes)



4. Left and right subtrees of every node have equal *height*

Too strong!
Only perfect trees ($2^n - 1$ nodes)



The AVL Tree Balance Condition

Left and right subtrees of *every node*
have *heights* **differing by at most 1**

Definition:

balance(*node*) = height(*node*.left) – height(*node*.right)

AVL *property*: **for every node *x*, –1 ≤ balance(*x*) ≤ 1**

That is, heights differ by at most 1

- Ensures small depth
  - Will prove this by showing that an AVL tree of height *h* must have a number of nodes *exponential* in *h*
- Easy (well, efficient) to maintain
  - Using single and double rotations
  - Perhaps not so easy to code….

Have fun on project 2!