CSE 332: Data Abstractions

Lecture 6: Dictionaries; Binary Search Trees

Ruth Anderson

Winter 2015

Announcements

- **Homework 1** – due TONIGHT at 11pm via catalyst
- **Homework 2** – due next Friday

- **Project 2** –
  – Coming soon!
  – Can work with a partner

Today

- Dictionaries
- Trees

Where we are

Studying the absolutely essential ADTs of computer science and classic data structures for implementing them

ADTs so far:

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

Next:

4. Dictionary (a.k.a. Map): associate keys with values
   - probably the most common, way more than priority queue

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

Data:

- *set* of (key, value) *pairs*
- keys must be *comparable*

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)`**
  - …

insert (rea, Ruth Anderson)

find (anied)

Conrad Nied,…

- rea
  Ruth
  Anderson
  …
- anied
  Conrad
  Nied
  …

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

A Modest Few Uses for Dictionaries

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

Simple implementations

For dictionary with *n* key/value pairs

	`insert`	`find`	`delete`
• Unsorted linked-list			
• Unsorted array			
• Sorted linked list			
• Sorted array			

We'll see a Binary Search Tree (BST) probably does better, but not in the worst case unless we keep it balanced

Simple implementations

For dictionary with *n* key/value pairs

	`insert`	**`find`**	**`delete`**
• Unsorted linked-list	O(n) *	$O(n)$	$O(n)$
• Unsorted array	O(n)*	$O(n)$	$O(n)$
• Sorted linked list	$O(n)$	$O(n)$	$O(n)$
• Sorted array	$O(n)$	$O(\mathtt{log}\ n)$	$O(n)$

We'll see a Binary Search Tree (BST) probably does better, but not in the worst case unless we keep it balanced

*Note: If we allow duplicates values to be inserted, you could do these in O(1) because you do not need to check for a key's existence before insertion

Lazy Deletion (e.g. in a sorted array)

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
- No need to shift values, etc.

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*
- **`find`** *O*(**`log`** *m*) *time* where *m* is data-structure size (m >= n)
- May complicate other operations

Better Dictionary data structures

Will spend the next several lectures looking at dictionaries with three different 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)

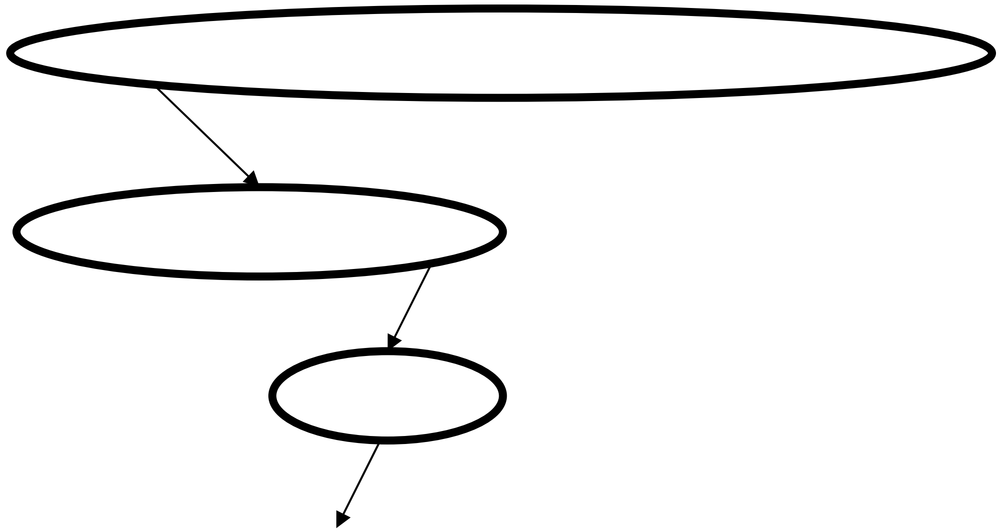
Why Trees?

Trees offer speed ups because of their branching factors

- Binary Search Trees are structured forms of *binary search*

Binary Search

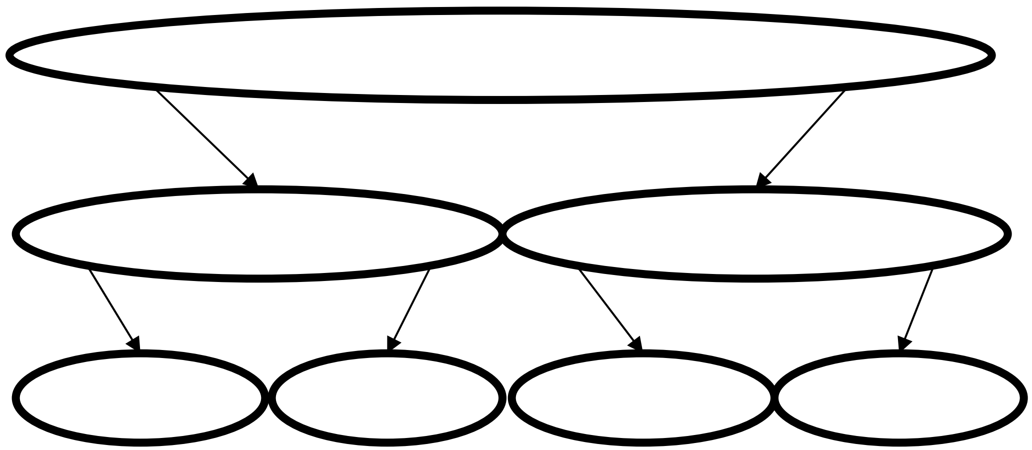
find(4)



1	**3**	**4**	**5**	**7**	**8**	**9**	**10**

Binary Search Tree

Our goal is the performance of binary search in a tree representation



1	**3**	**4**	**5**	**7**	**8**	**9**	**10**

Why Trees?

Trees offer speed ups because of their branching factors

- Binary Search Trees are structured forms of *binary search*

Even a basic BST is fairly good

	Insert	**Find**	**Delete**
Worse-Case	O(n)	O(n)	O(n)
Average-Case	O(log n)	O(log n)	O(log n)

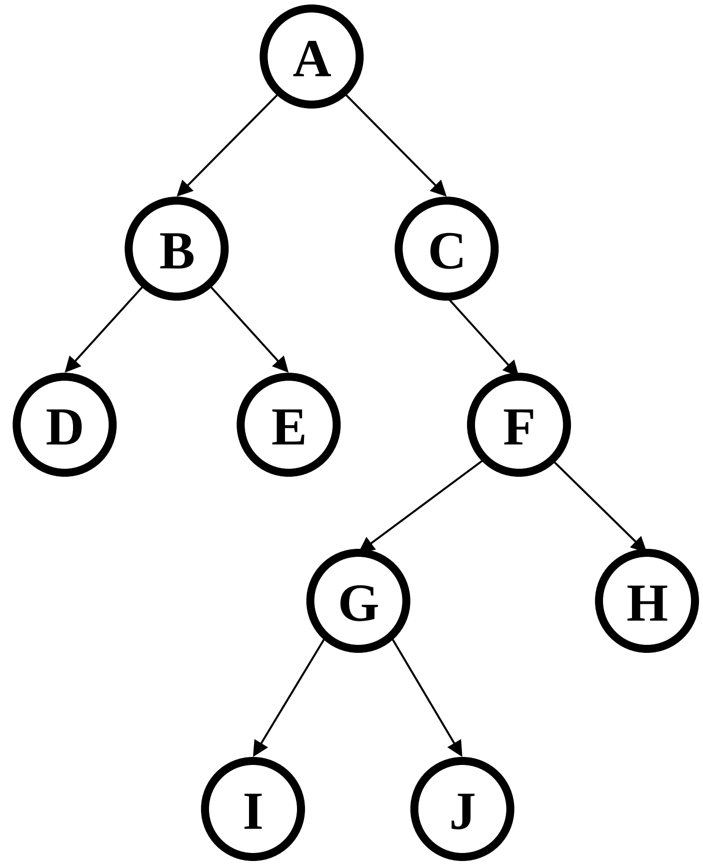
Binary Trees

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

- Representation:

<table>
<tr><th colspan="2">Data</th></tr>
<tr><td>left pointer</td><td>right pointer</td></tr>
</table>

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



Binary Tree: Some Numbers

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

For binary tree of height *h*:

– max # of leaves:

– max # of nodes:

– min # of leaves:

– min # of nodes:

Binary Trees: Some Numbers

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

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: $h + 1$

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

Calculating height

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

```
int treeHeight(Node root) {

              ???


}
```

Calculating height

What is 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));
}
```

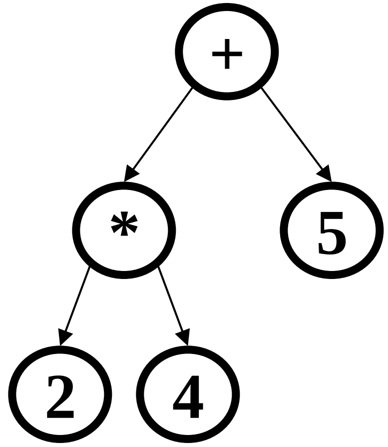
Running time for tree with n nodes: $O(n)$ – single pass over tree

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

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



(an expression tree)

Tree Traversals

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

- *Pre-order*: root, left subtree, right subtree

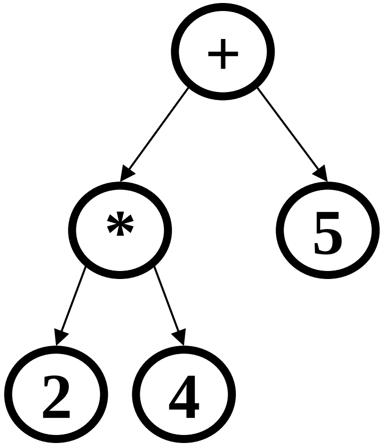
  + * 2 4 5

- *In-order*: left subtree, root, right subtree

  2 * 4 + 5

- *Post-order*: left subtree, right subtree, root

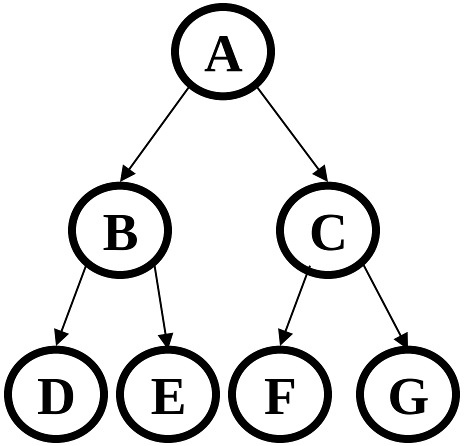
  2 4 * 5 +



(an expression tree)

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

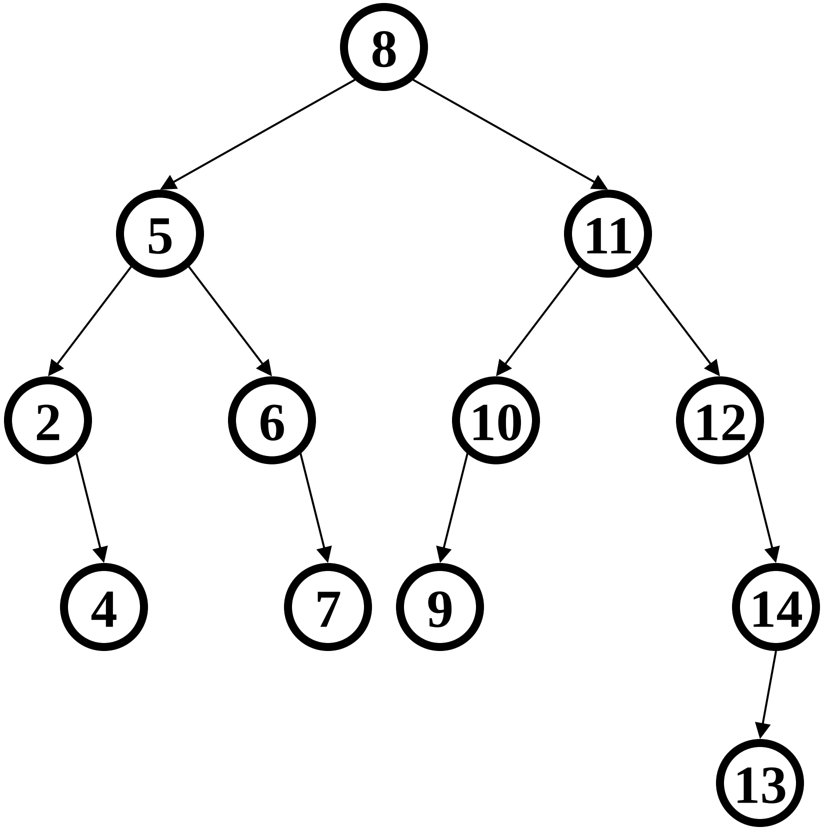
Sometimes order matters

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

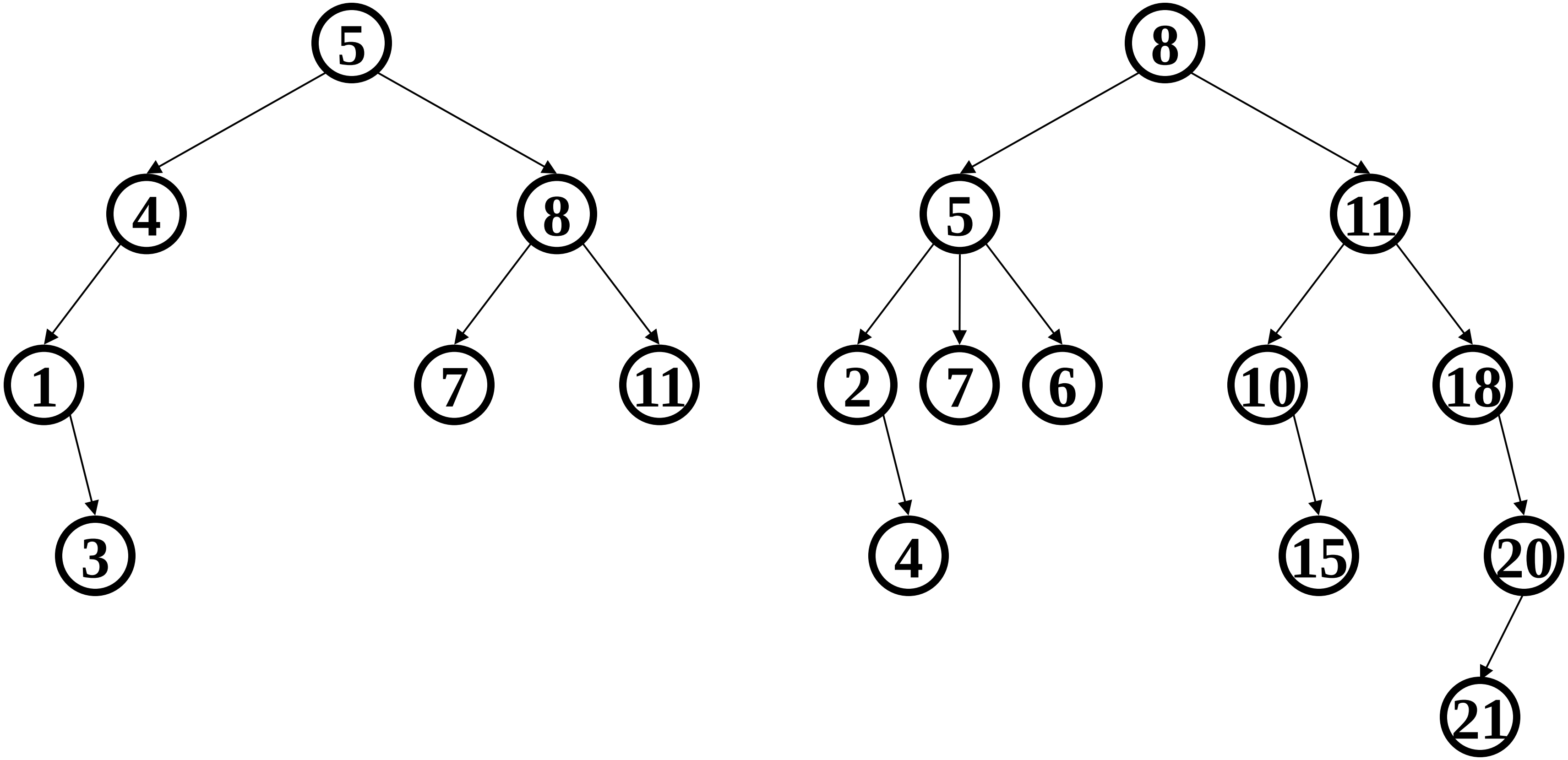
A
  B
    D
    E
  C
    F
    G

Binary Search Tree

- Structural property ("binary")
  - each node has ≤ 2 children
  - result: keeps operations simple
- 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

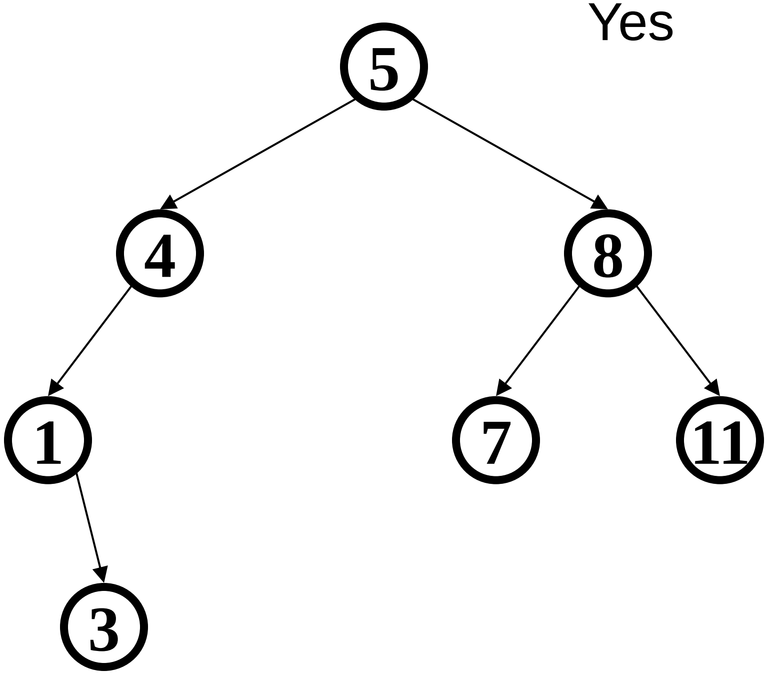


Are these BSTs?

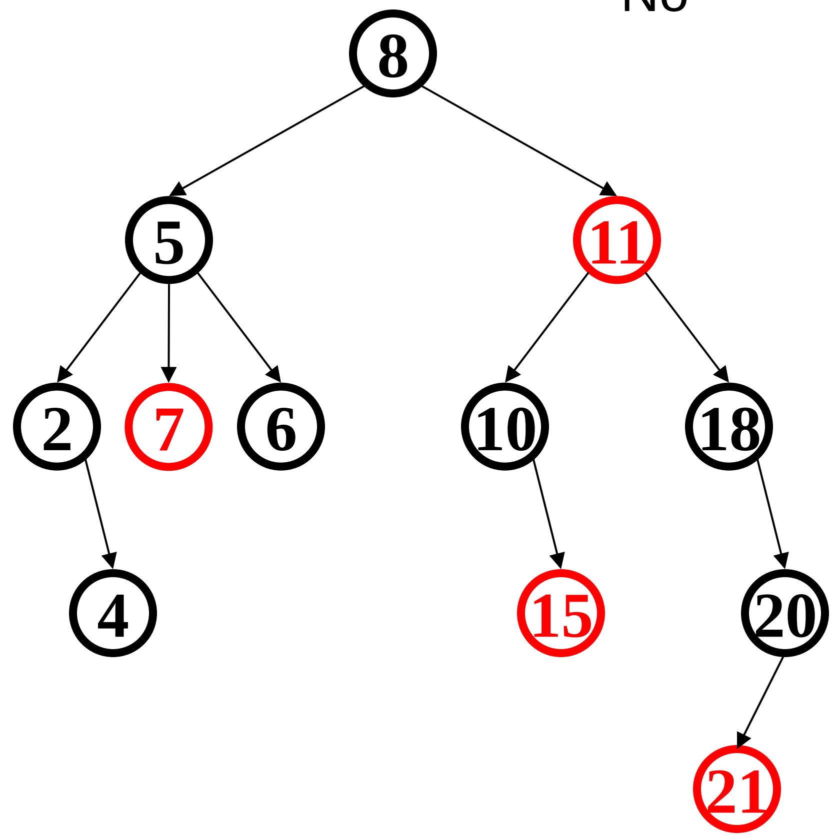


Are these BSTs?

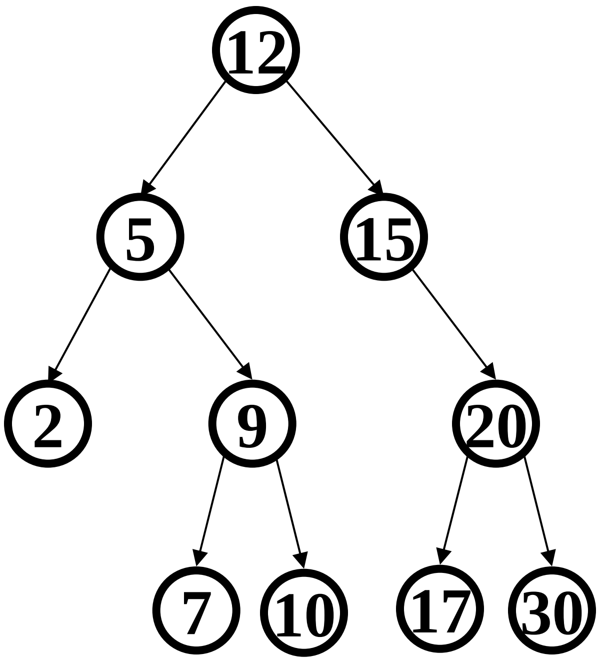
Yes



No

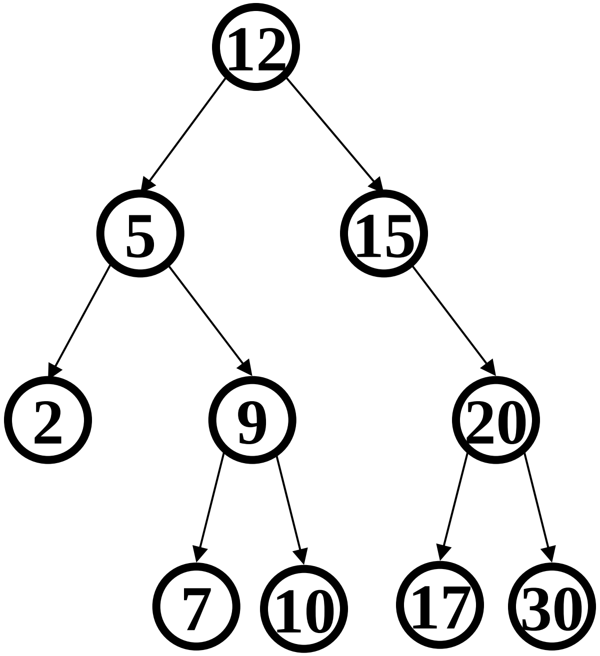


Find in BST, Recursive




```
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;
}
```

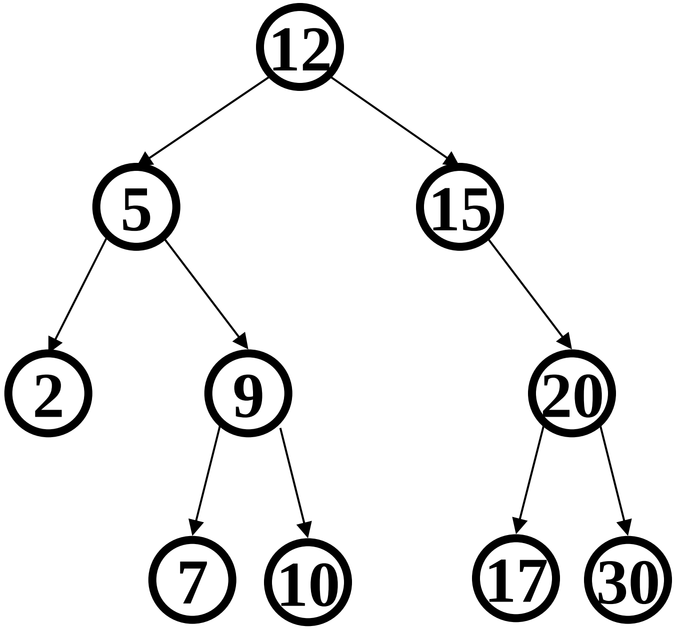
Find in BST, Iterative



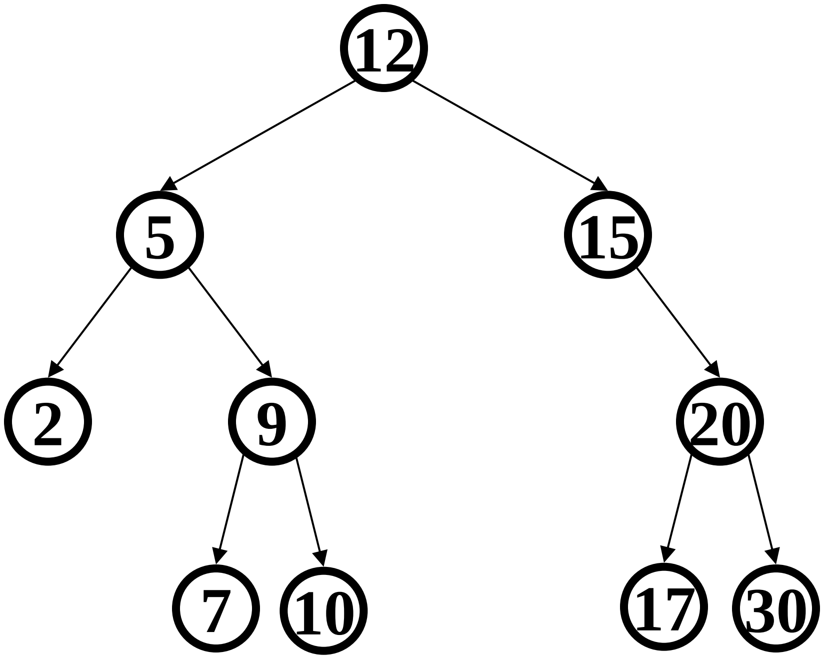

```
Data find(Key key, Node root){
 while(root != null
        && root.key != key) {
  if(key < root.key)
    root = root.left;
  else(key > root.key)
    root = root.right;
 }
 if(root == null)
    return null;
 return root.data;
}
```

Other “finding operations”

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



Insert in BST

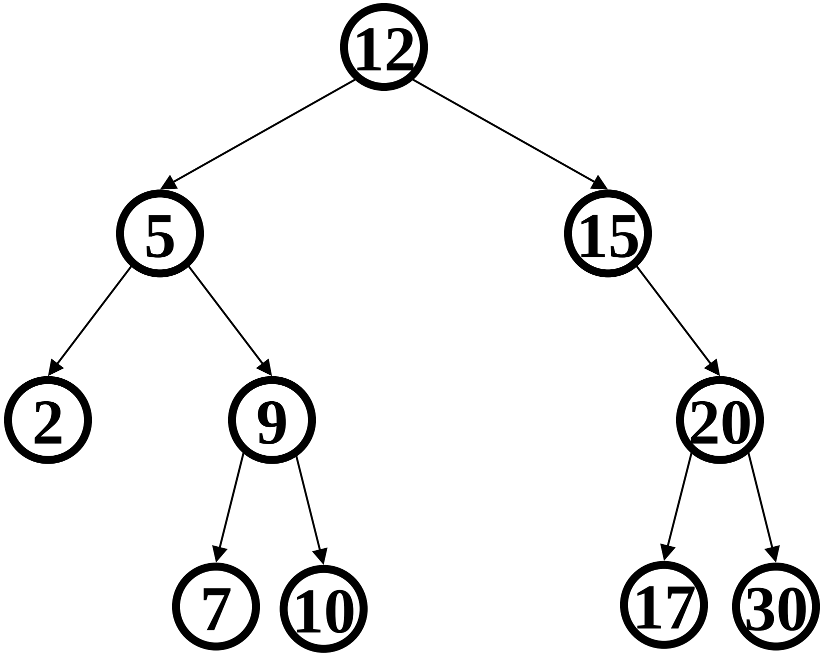



```
insert(13)
insert(8)
insert(31)
```

(New) insertions happen only at leaves – easy!

1. Find
2. Create a new node

Deletion in BST



Why might deletion be harder than insertion?

Deletion

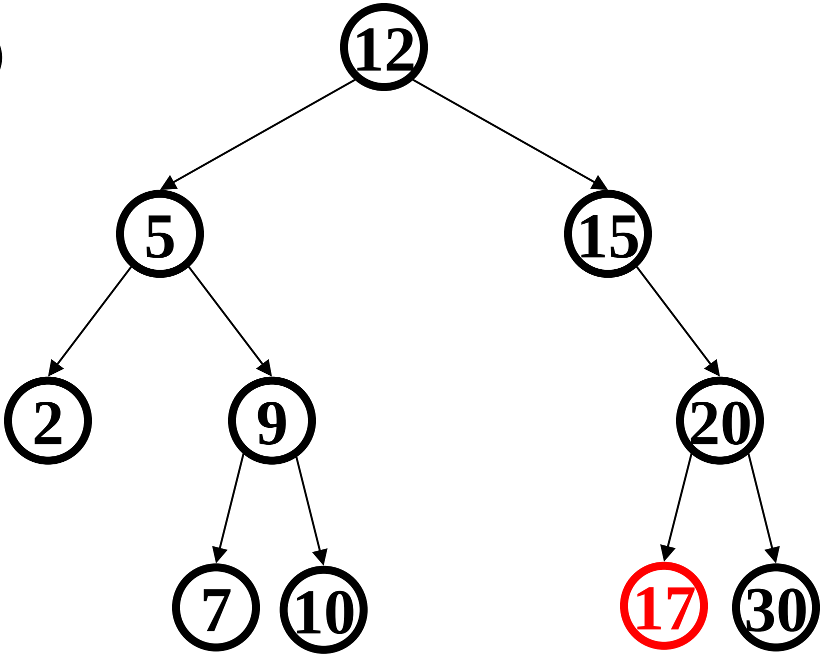
- Removing an item disrupts the tree structure

- Basic idea:
  - **`find`** the node to be removed,
  - Remove it
  - “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

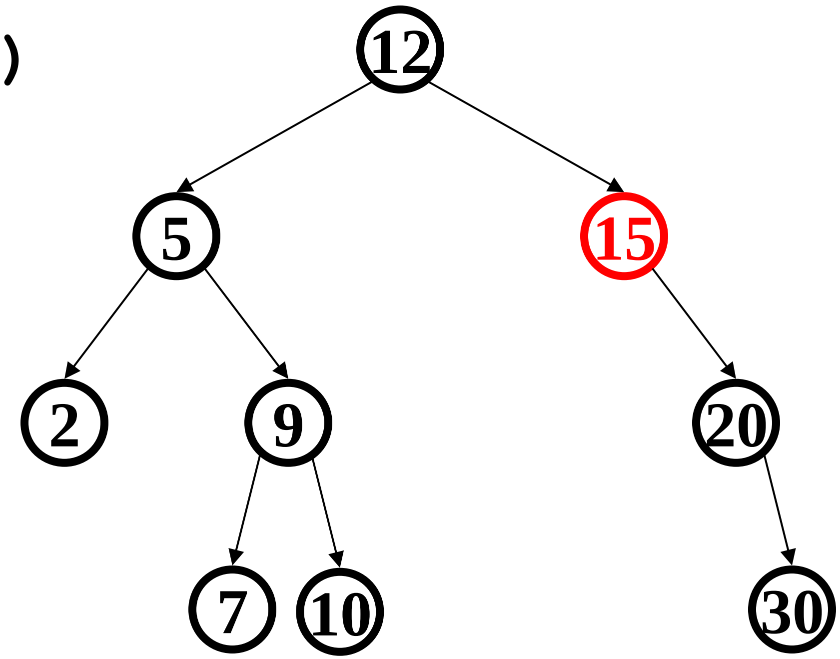
Deletion – The Leaf Case

`delete(17)`



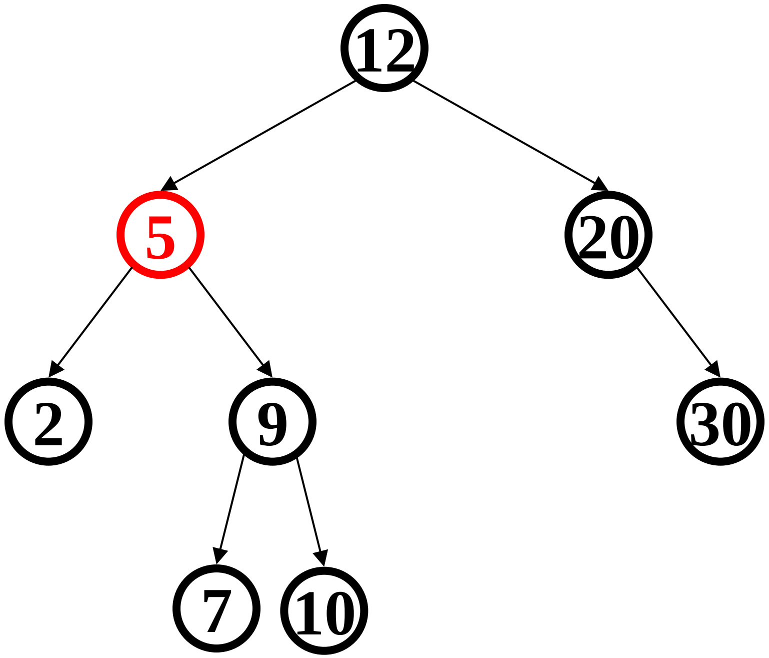
Deletion – The One Child Case

`delete(15)`



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

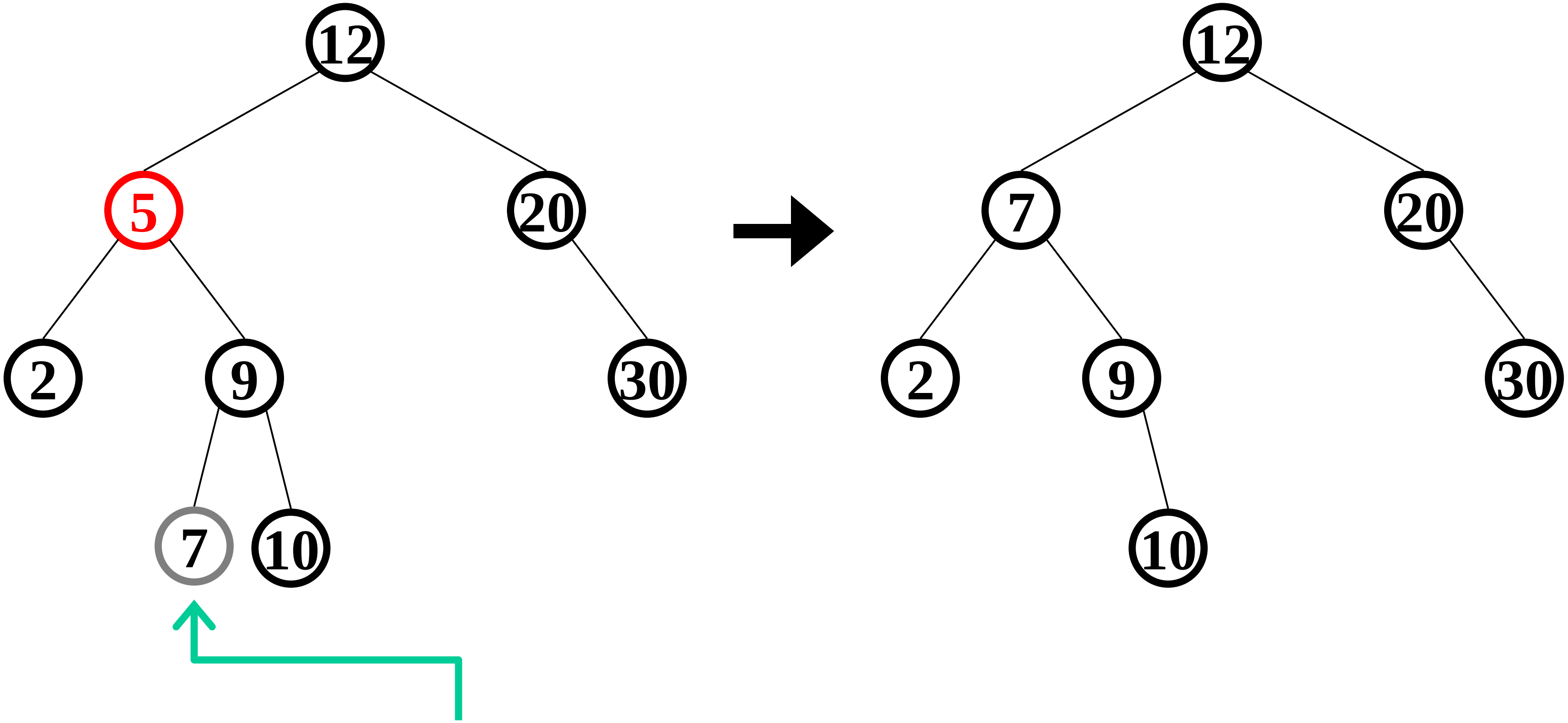
Options:

- *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!

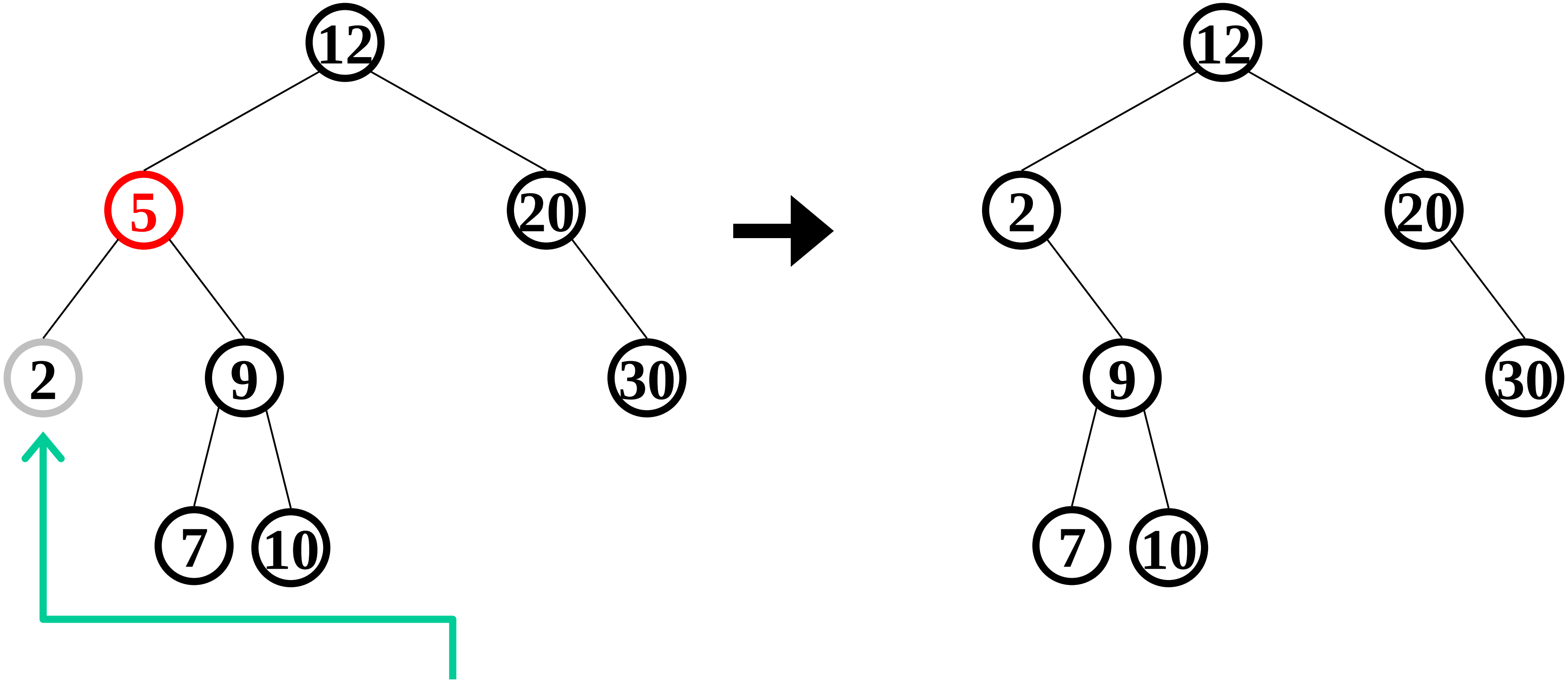
Delete Using Successor



findMin(right sub tree) → 7

`delete(5)`

Delete Using Predecessor



findMax(left sub tree) → 2

`delete(5)`

BuildTree for BST

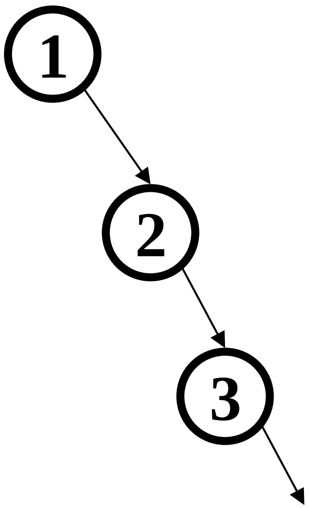
- We had **`buildHeap`**, so let's consider **`buildTree`**

- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST

  - If inserted in given order, what is the tree?

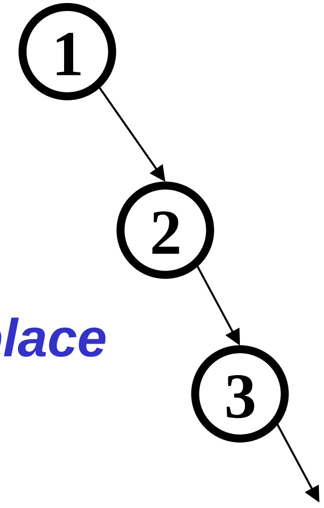
  - What big-O runtime for this kind of sorted input?

  - Is inserting in the reverse order any better?



BuildTree for BST

- We had **`buildHeap`**, so let's consider **`buildTree`**
- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
  - If inserted in given order, what is the tree?
  - What big-O runtime for this kind of sorted input?
  - Is inserting in the reverse order any better?



$O(n^2)$

Not a happy place

Balanced BST

Observation

- BST: the shallower the better!
- For a BST with n nodes inserted in arbitrary order
  - Average height is $O(\texttt{log}\ n)$ – 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 $O(\texttt{log}\ n)$ – 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

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

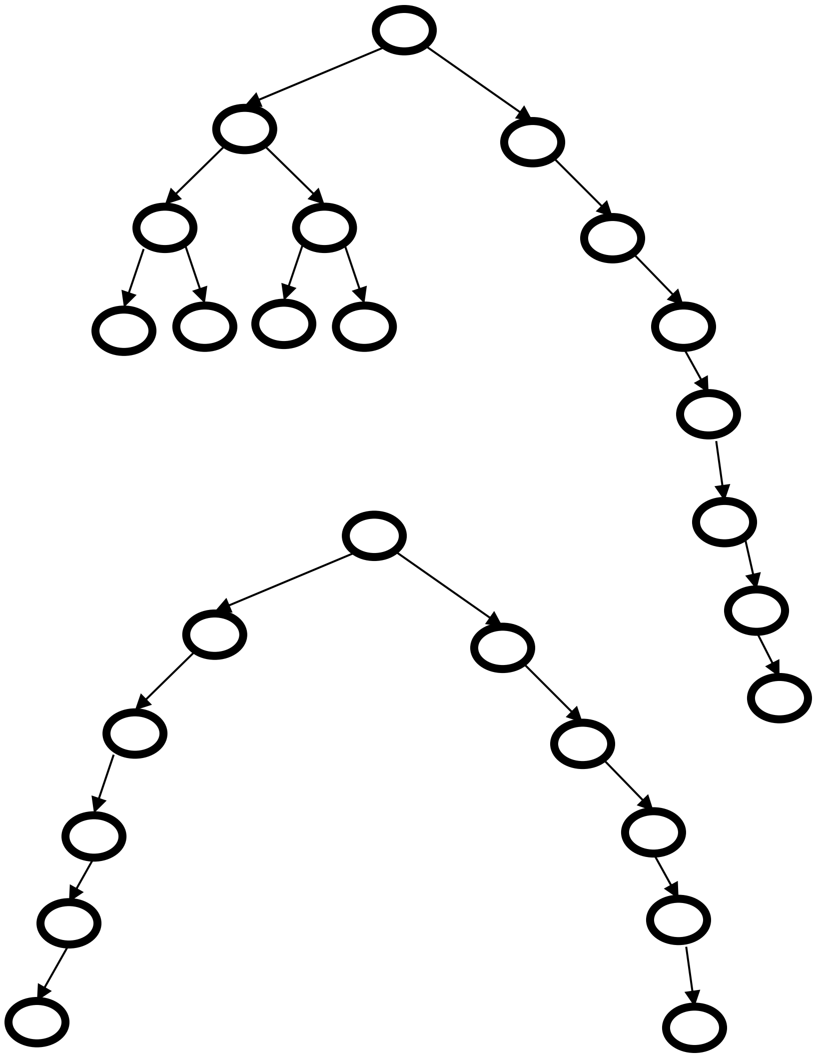
Potential Balance Conditions

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

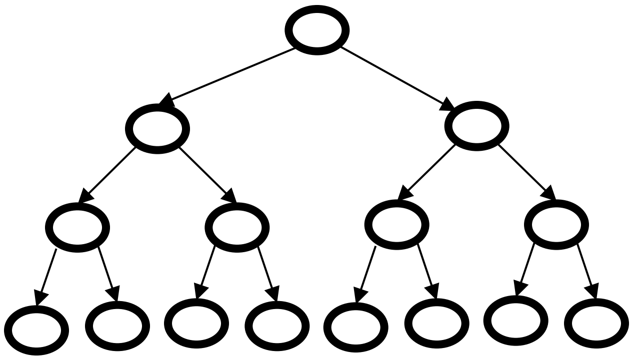
3. Left and right subtrees of every node have equal number of nodes

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

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 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*, $-1 \leq$ balance(*x*) ≤ 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