CSE 373: Data Structures and Algorithms

Lecture 9: Binary Search Trees

Instructor: Lilian de Greef
Quarter: Summer 2017

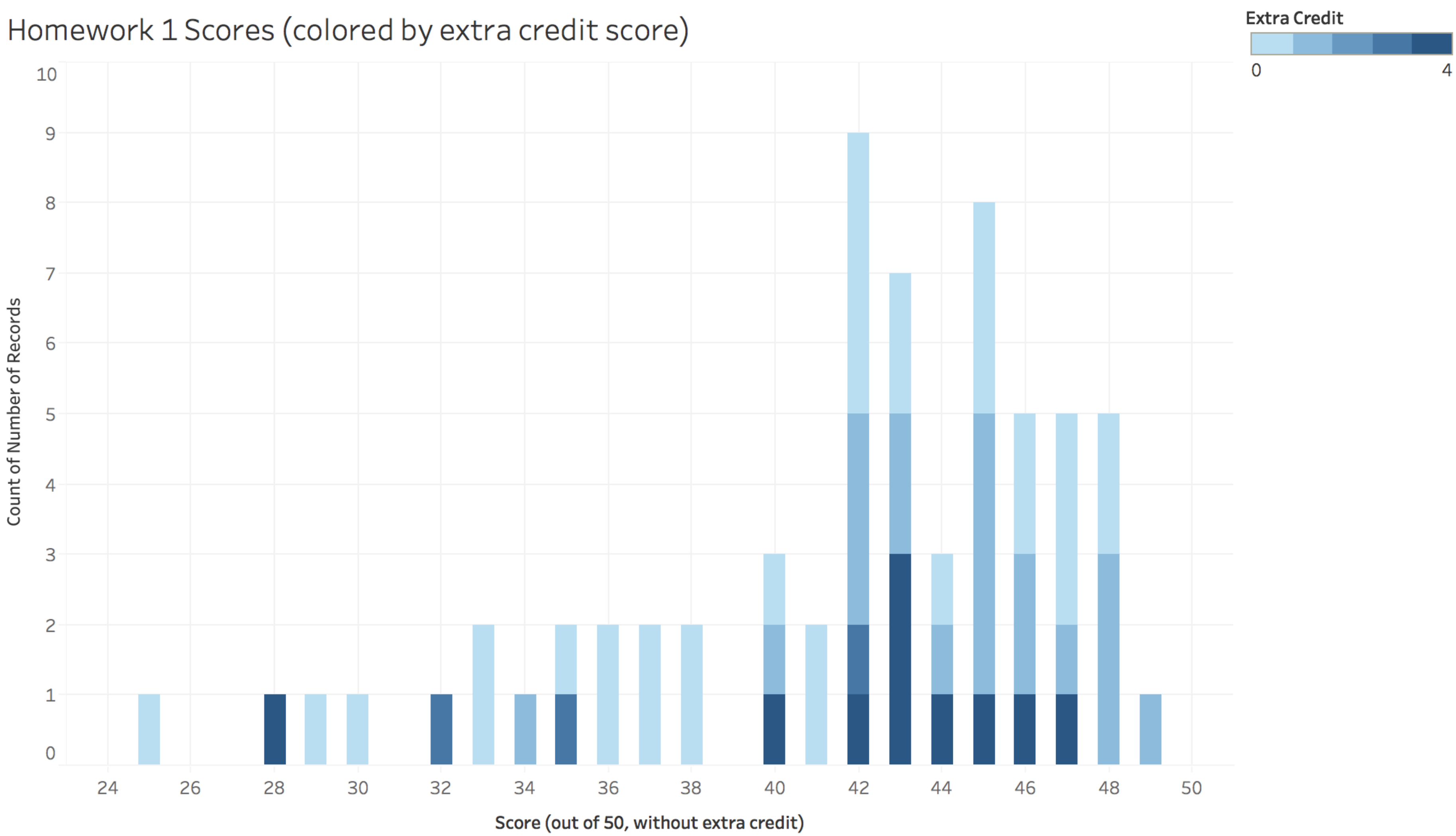
Today

- Announcements
- Binary Trees
  - Height
  - Traversals
- Binary Search Trees
  - Definition
  - `find`
  - `insert`
  - `delete`
  - `buildTree`

Announcements

- Change to office hours for just this week
  - Tuesday's "office" office hours / private office hours
    - 12:00pm – 12:30pm
    - (not at 1:30pm!)
  - Dorothy and I trading 2:00pm - 3:00pm office hours this week
    - Same time and location

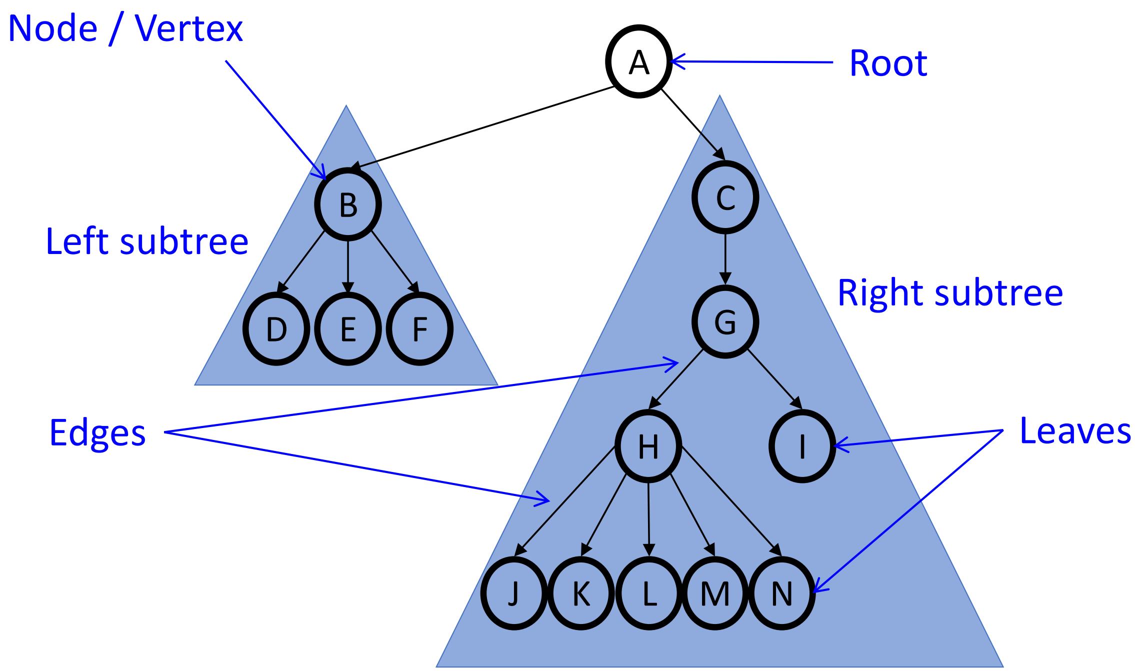
- Homework 1 Statistics
  - Mean: 39.7/50 (+1 extra credit)
  - Median: 42.5/50 (+0 extra credit)
  - Max: 49/50 (+1) or 47/50 (+4)
  - Standard Deviation: 10.18


Homework 1 Scores (colored by extra credit score)
Extra Credit
0
4
Count of Number of Records
10
9
8
7
6
5
4
3
2
1
0
24
26
28
30
32
34
36
38
40
42
44
46
48
50
Score (out of 50, without extra credit)

Binary Trees

Continued – part 2!

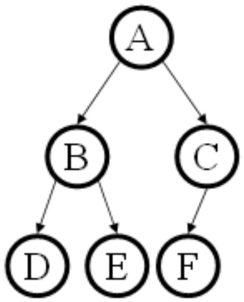
Reminder: Tree terminology



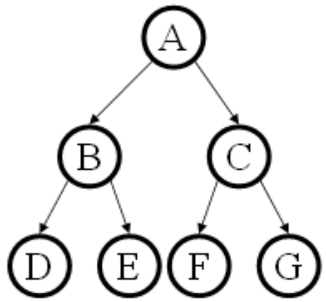
Binary Trees

- Binary tree: Each node has at most 2 children (branching factor 2)
- Binary tree is
  - A root *(with data)*
  - A left subtree *(may be empty)*
  - A right subtree *(may be empty)*

- Special Cases:



Complete Tree



Perfect Tree

(Last week's practice) What does the following method do?

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

A. It calculates the number of nodes in the tree.

B. It calculates the depth of the nodes.

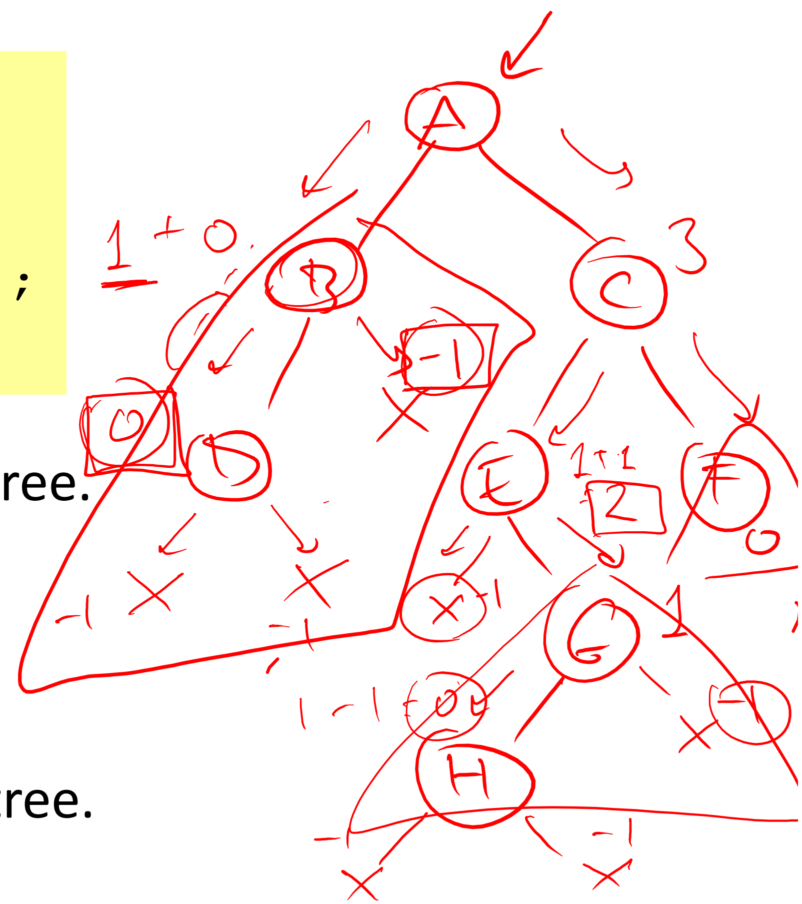
C. It calculates the height of the tree.

D. It calculates the number of leaves in the tree.

$$\text{height} = 1 + \max(1, 3)$$
$$= 4$$

(Last week's practice) What does the following method do?

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



A. It calculates the number of nodes in the tree.

B. It calculates the depth of the nodes.

C. It calculates the height of the tree.

D. It calculates the number of leaves in the tree.

Binary Trees: Some Numbers

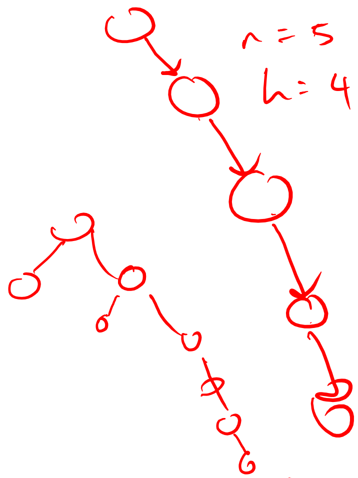
$1+2+4+8+\cdots 2^h = 2^{h+1}-1$

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

worst-case

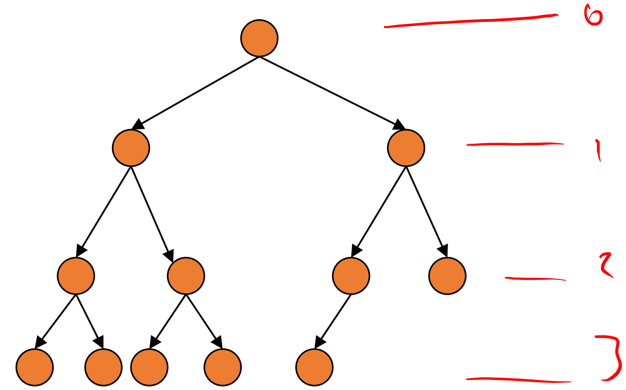
$n=5$

$h=4$



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: $h+1$



height = 3

For *n* nodes, the min height (best-case) is $O(\log n)$

the max height (worst-case) is $O(n)$

where n = # of nodes

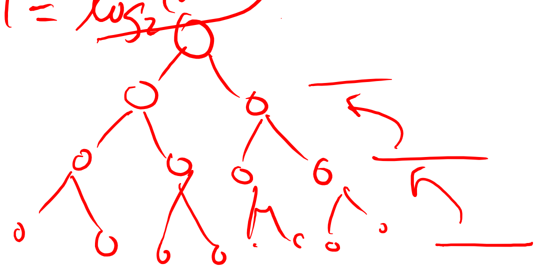
Tree Traversals

n nodes

$2^{h+1} - 1 = n$

$2^{h+1} = n+1$

$h+1 = \log_2(n+1)$



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

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

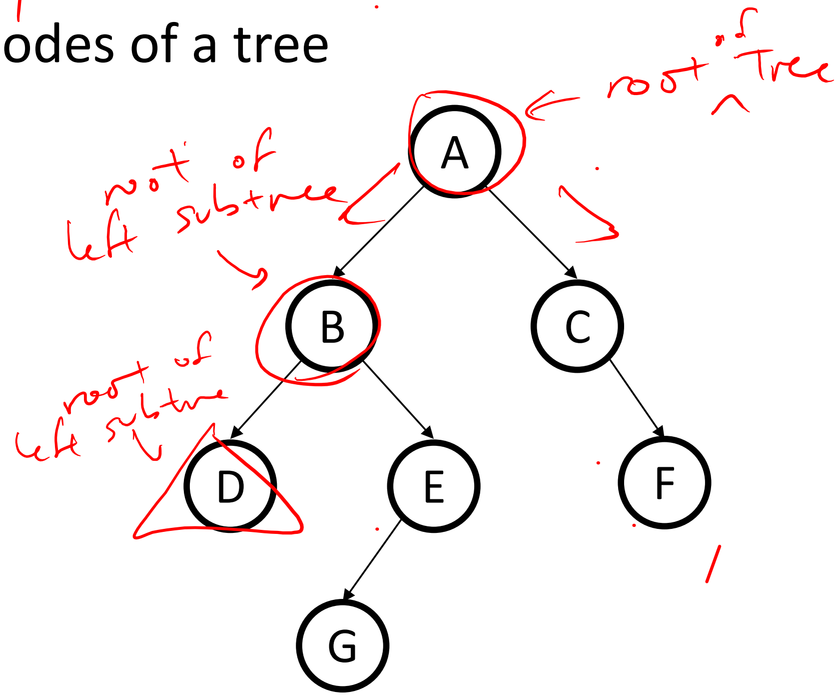
  A B D E G C F

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

  D B G E A C F

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

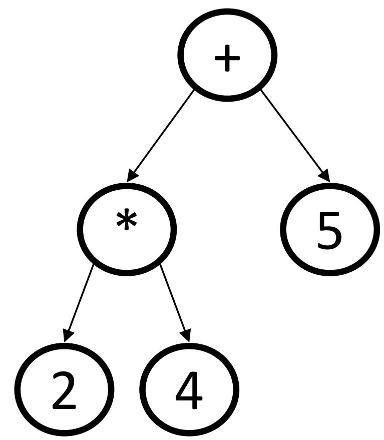
  D G E B F C A



Tree Traversals: Practice

Which one makes sense for evaluating this *expression 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 +



Binary Search Trees

A kind of binary tree!

Binary Search Tree (BST) Data Structure

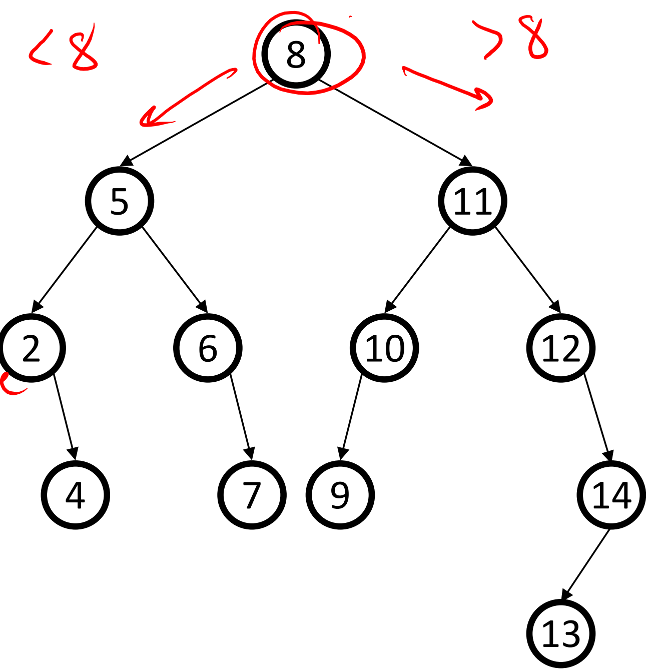
- Structure property (binary tree)
  - Each node has ≤ 2 children
  - Result: keeps operations simple

- Order property

  - All values in left subtree smaller than the node's value

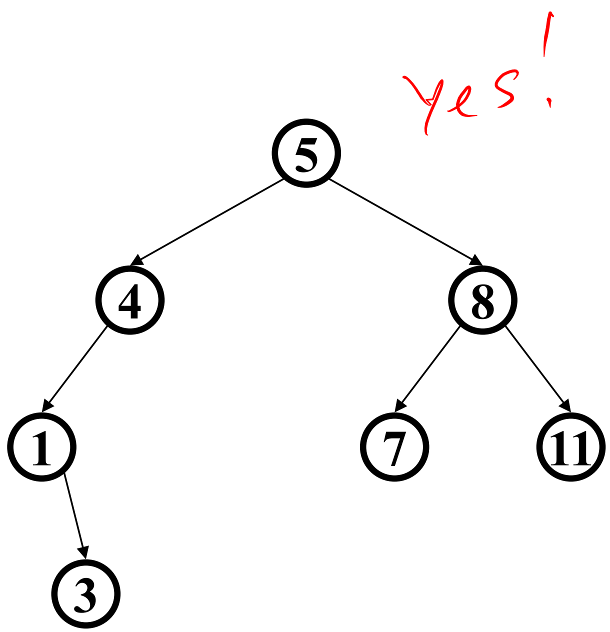
  - All values in right subtree greater than node's value

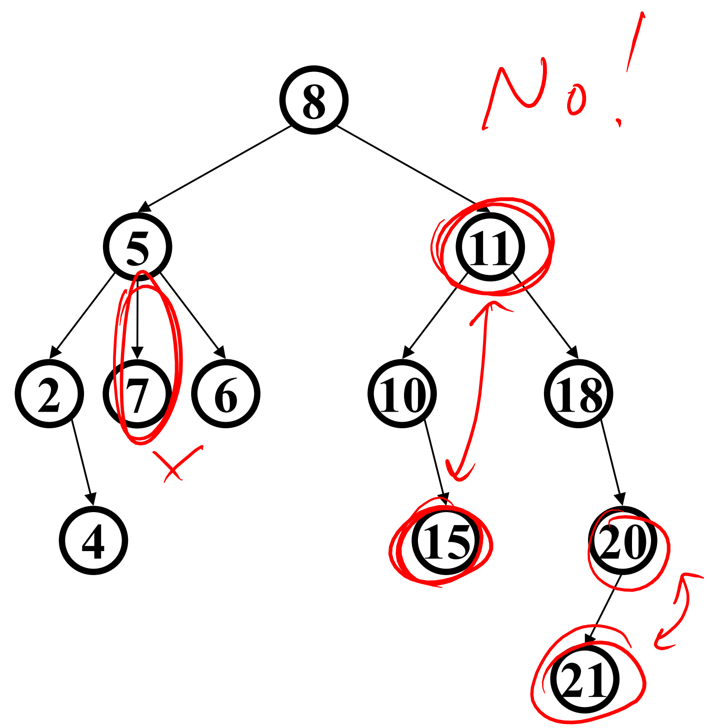
  - Result: straight-forward to find any given value



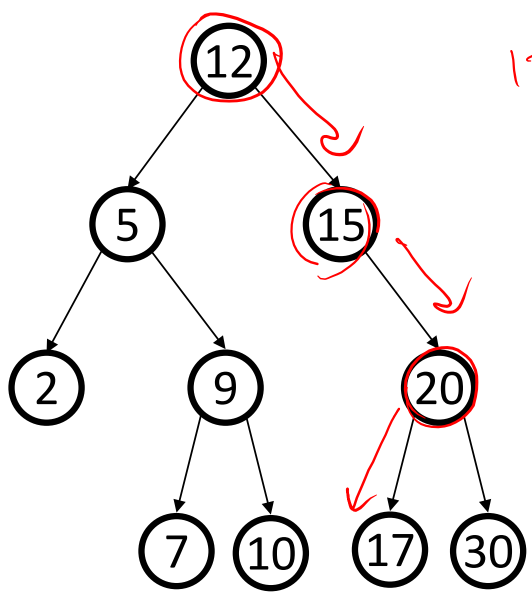
A binary *search* tree is a type of binary tree
(but not all binary trees are binary search trees!)

Practice: are these BSTs?





How do we `find(value)` in BST's?



find(17)

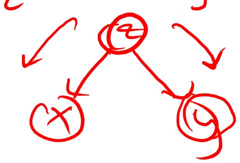
$12 < 17$

$15 < 17$

$20 > 17$

`find` in BST: Recursive Version

$x < z$ $y > z$

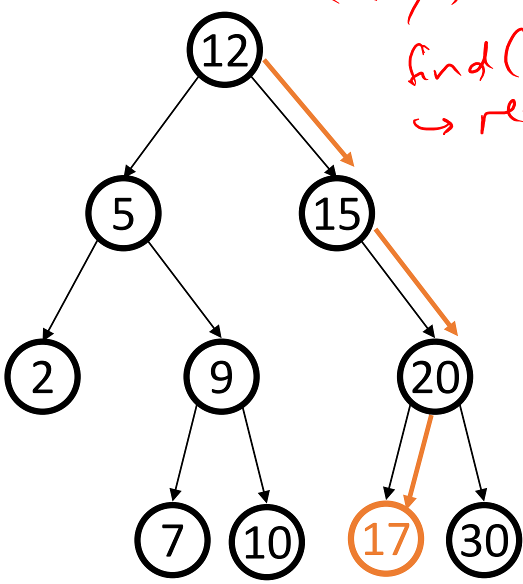


$x == y$?

(key, data)

find(key)

→ return data




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

Data

value

value

value

What is the running time?

Balanced tree : $O(\log n)$

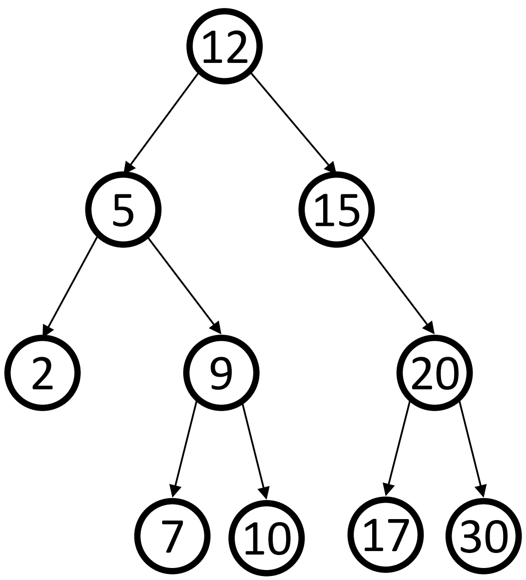
n = # nodes

worst-case : $O(n)$

Happens for very lopsided tree!

O → O → O → O → O

`find` in BST: Iterative Version




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

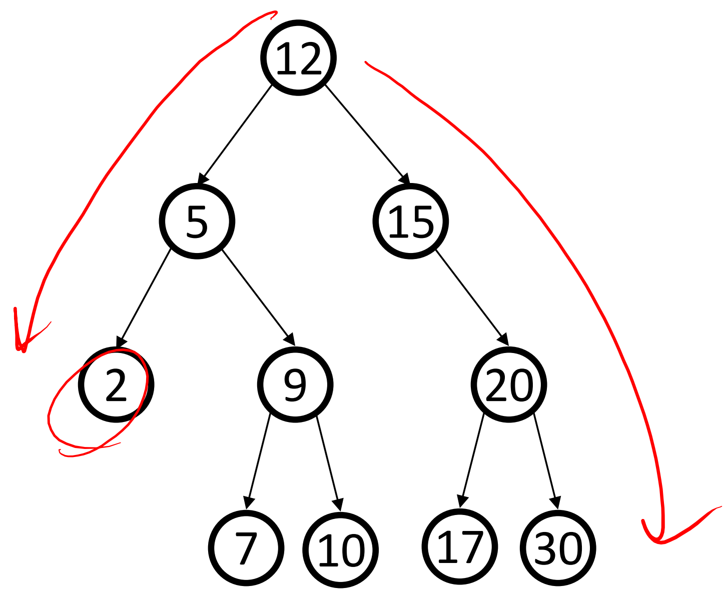
Other BST "Finding" Operations

`findMin`: Find *minimum* node

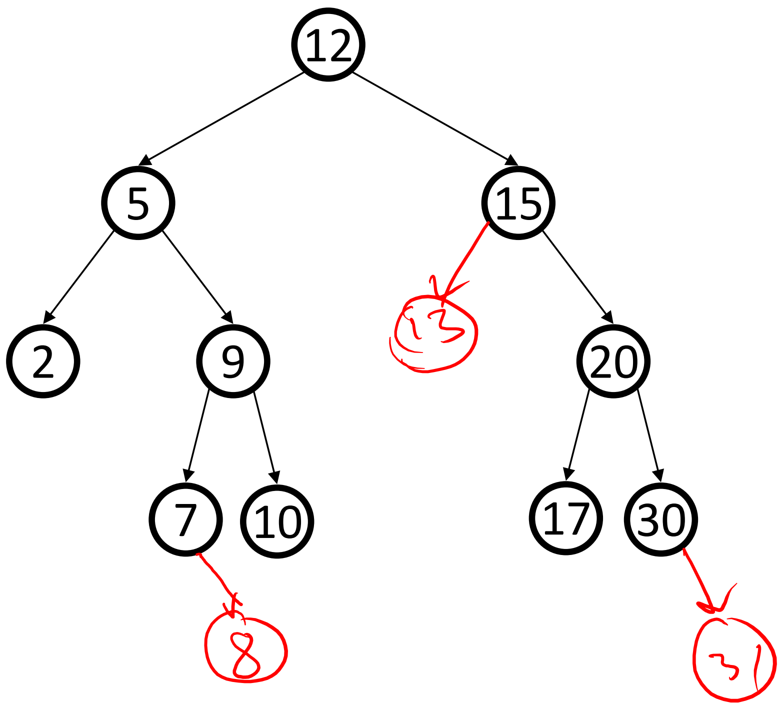
leftmost node

`findMax`: Find *maximum* node

rightmost node



`insert` in BST

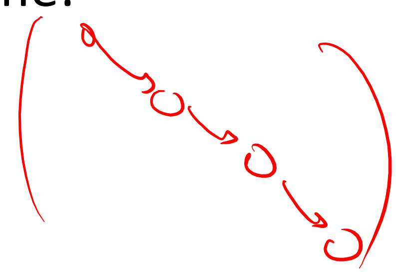


```
insert(13)
insert(8)
insert(31)
```

Insertions ~~happen at the leaves~~ become leaves

Worst-case running time:

$O(n)$



Practice with `insert`, primer for `delete`

Start with an empty tree. Insert the following values, in the given order:
`14, 2, 5, 20, 42, 1, 4, 16`

Then, changing as few nodes as possible, delete the following in order:
`42, 14`

What would the root of the resulting tree be?

A. `2`
B. `4`
C. `5`
D. `16`

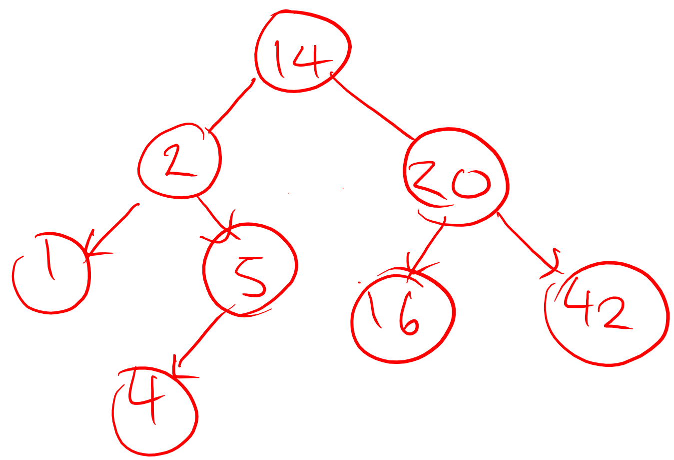
Practice with `insert`, primer for `delete`

Start with an empty tree. Insert the following values in the given order:

`14, 2, 5, 20, 42, 1, 4, 16`

Then, changing as few nodes as possible, delete the following in order:

`42, 14`



`delete` in BST

- Why might `delete` be harder than `insert`?

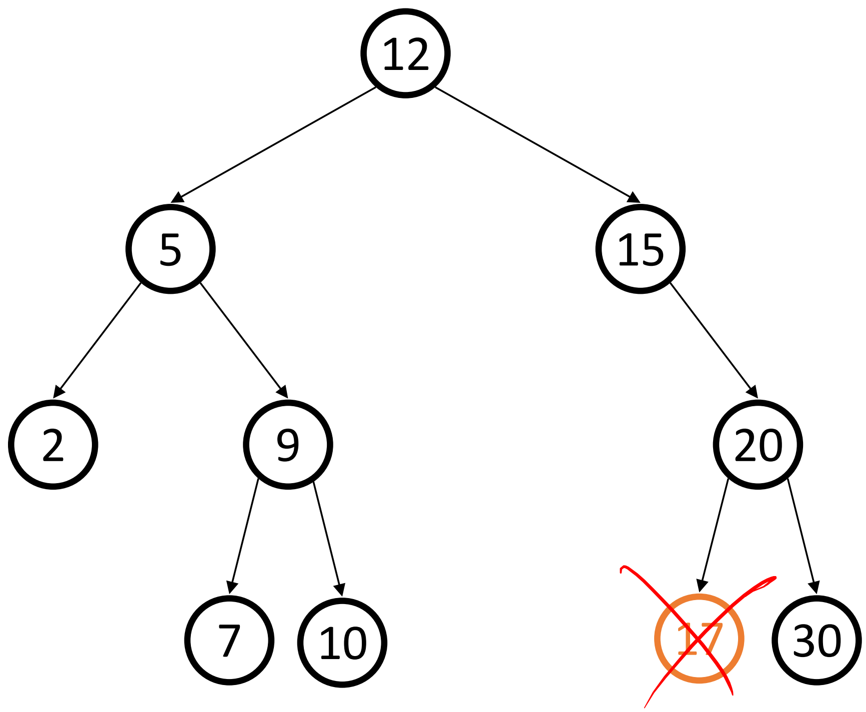
  You don't want to abandon your child nodes!

- Basic idea: find the node to remove, the "fix" the tree so that it's still a BST

- Three potential cases to fix:
  - Node has no children (leaf)
  - Node has one child
  - Node has two children

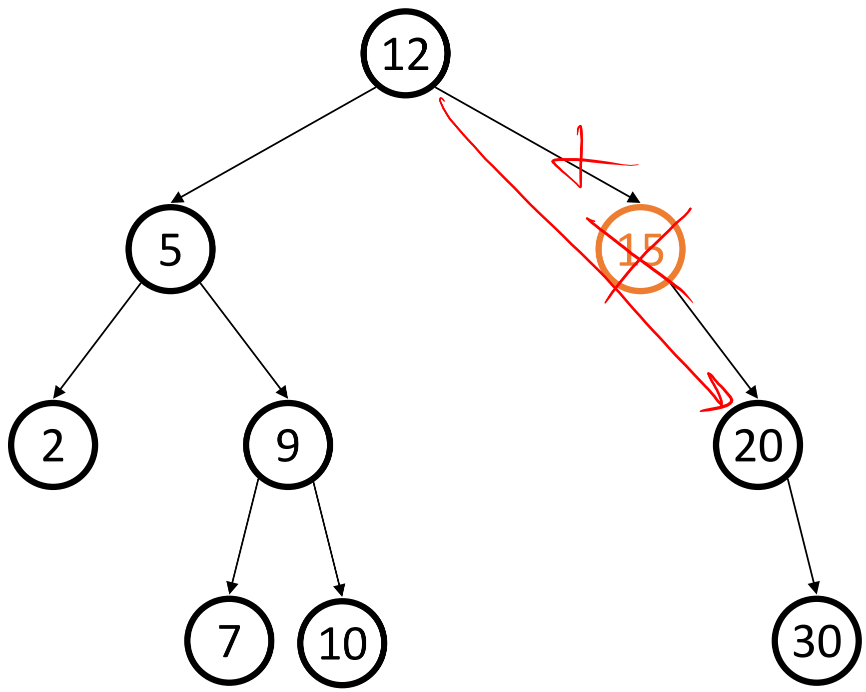
`delete` case: Leaf

`delete(17)`



`delete` case: One Child

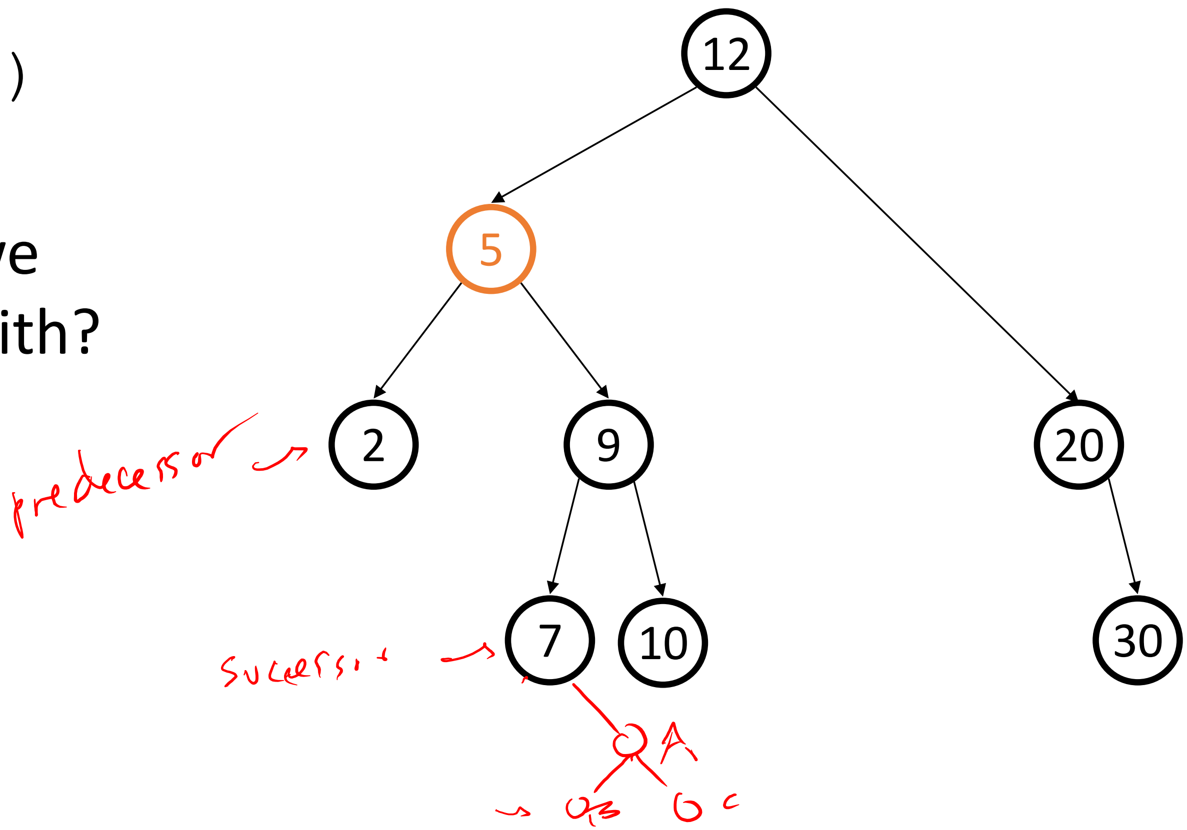
`delete(15)`



`delete` case: Two Children

`delete(5)`

What can we replace 5 with?



`delete` case: Two Children

What can we replace the node with?

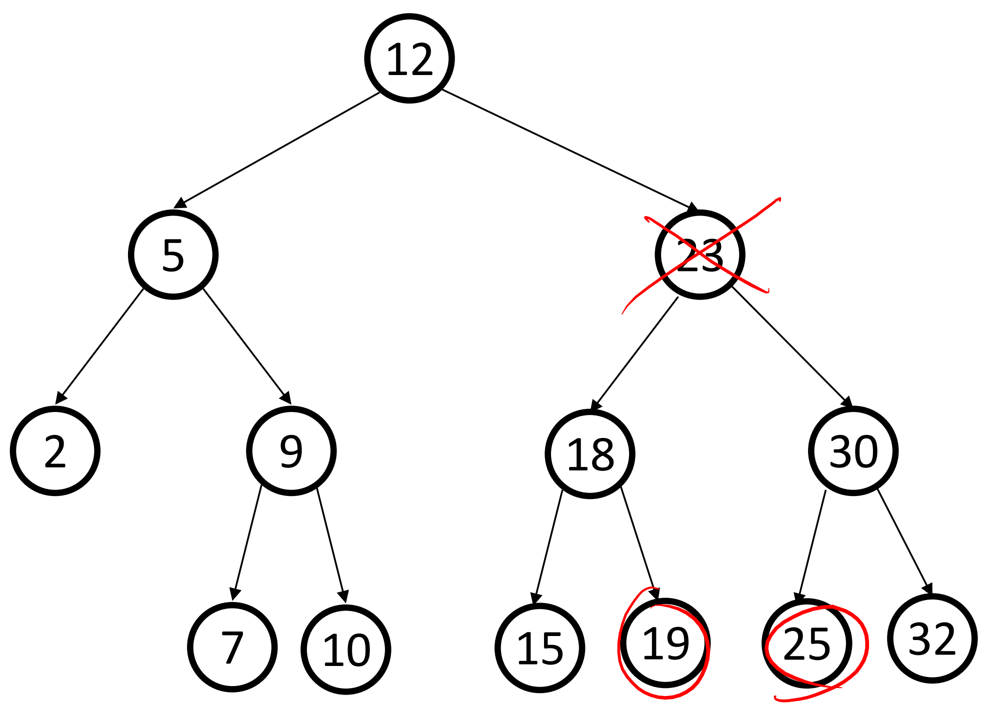
Options:

successor — minimum node from the right subtree

predecessor — maximum node from the left subtree

`delete` case: Two Children (example #2)

`delete(23)`

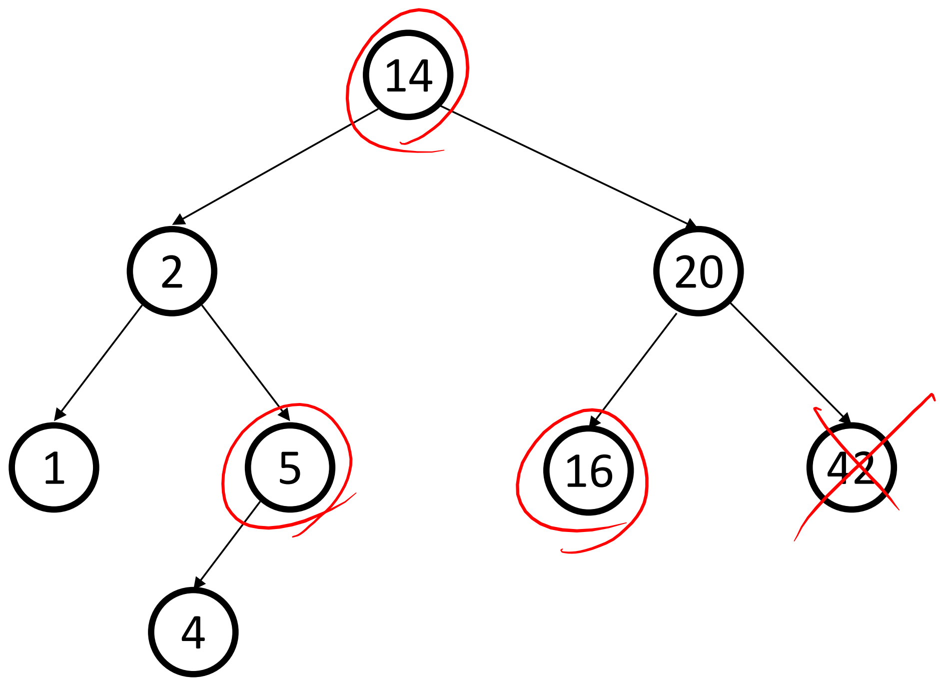


Practice with `insert`, primer for `delete`

REVISITED

Changing as few nodes as possible, delete the following in order:

~~42~~, 14



`delete` through Lazy Deletion

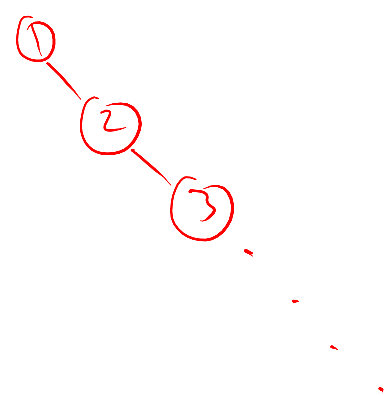
- Lazy deletion can work well for a BST
  - Simpler
  - Can do “real deletions” later as a batch
  - Some inserts can just “undelete” a tree node

- But
  - Can waste space and slow down find operations
  - Make some operations more complicated:
    - e.g., **`findMin`** and **`findMax`**?

`buildTree` for BST

Let's consider `buildTree` (insert values starting from an empty tree)

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



- If inserted in given order, what is the tree? stick!
- What big-O runtime for buildTree on this sorted input?
- Is inserting in the reverse order any better?