CSE373: Data Structures and Algorithms

# Binary Search Trees

Steve Tanimoto
Winter 2016

This lecture material represents the work of multiple instructors at the University of Washington. Thank you to all who have contributed!

## Recall ....

- Dictionary ADT
  - stores (key, value) pairs
  - `find`, `insert`, `delete`
- Trees
  - Terminology
  - Binary Trees



## Reminder: Tree terminology

Node / Vertex
Root
Left subtree
Right subtree
Edges
Leaves

A B C D E F G H I J K L M N



## Example Tree Calculations

**Recall: Height of a tree is the maximum number of edges from the root to a leaf.**

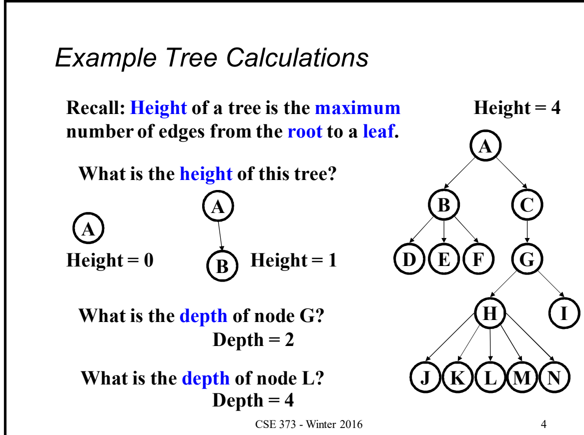
**What is the height of this tree?**

A — Height = 0

A → B — Height = 1

**What is the depth of node G?**
Depth = 2

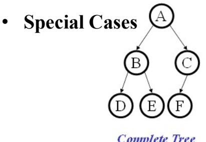
**What is the depth of node L?**
Depth = 4






## Binary Trees

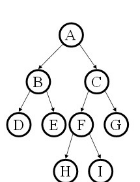
- **Binary tree: Each node has at most 2 children (branching factor 2)**
- **Binary tree is**
  - **A root *(with data)***
  - **A left subtree *(may be empty)***
  - **A right subtree *(may be empty)***
- **Special Cases**




*Complete Tree*

A, B, C, D, E, F, G

*Perfect Tree*




*Full Tree*



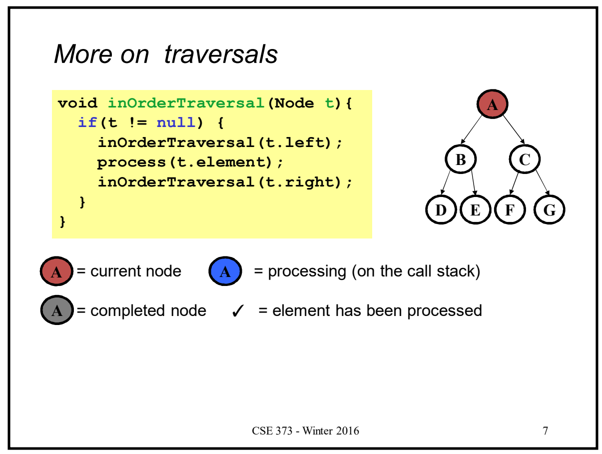
## Tree Traversals

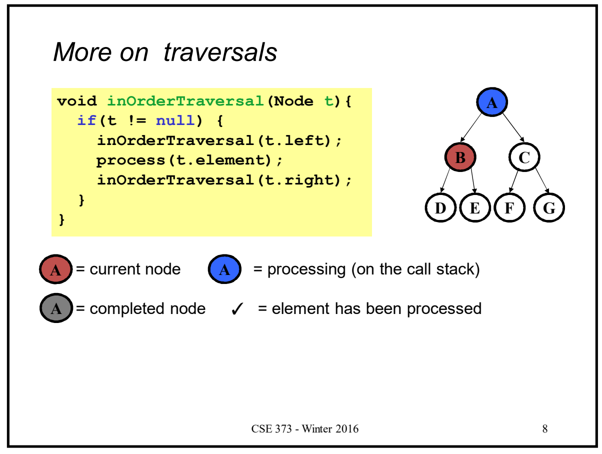
A *traversal* is an order for visiting all the nodes of a tree

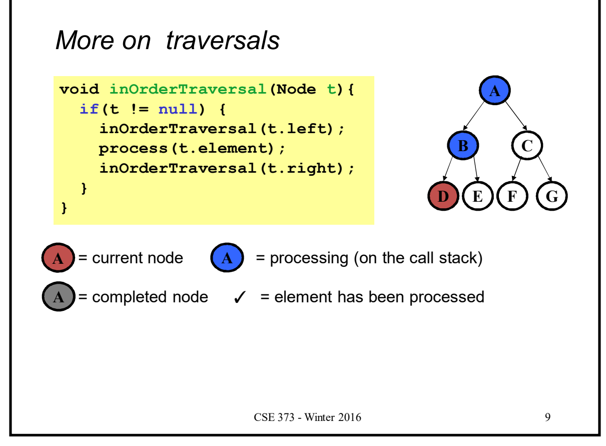
- *Pre-order*: root, left subtree, right subtree
  + * 2 4 5
- *In-order*: left subtree, root, right subtree
  2 * 4 + 5
- *Post-order*: left subtree, right subtree, root
  2 4 * 5 +

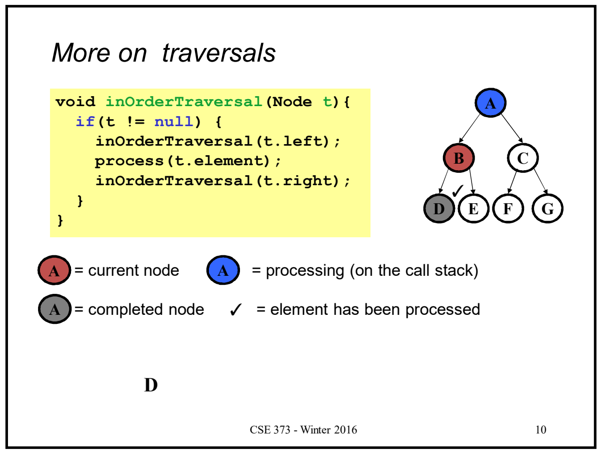
**(an expression tree)**

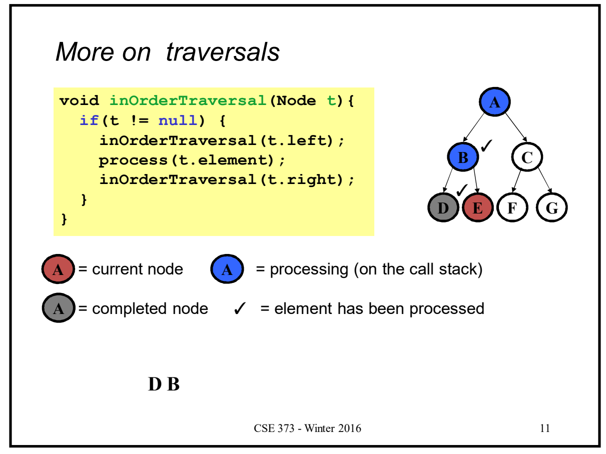
\+ , * , 5 , 2 , 4

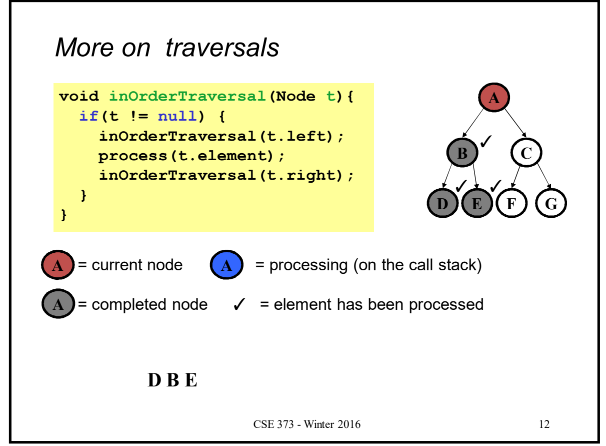

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
A
B
C
D
E
F
G
= current node
= processing (on the call stack)
= completed node
✓ = element has been processed
CSE 373 - Winter 2016
7

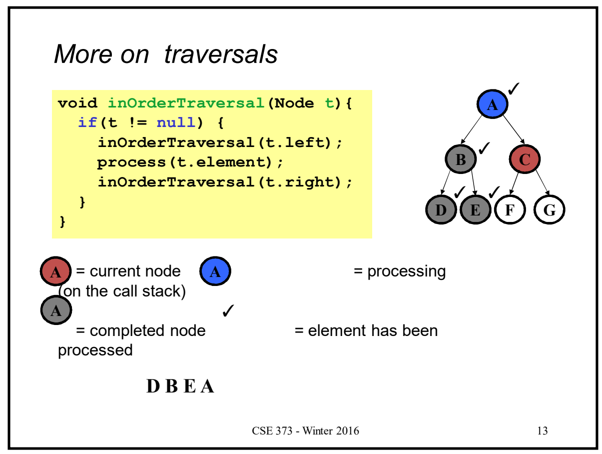

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
= current node
= processing (on the call stack)
= completed node
✓ = element has been processed
CSE 373 - Winter 2016
8

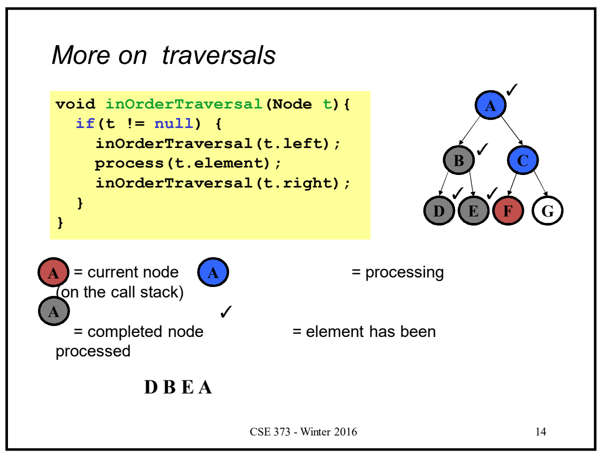

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
= current node
= processing (on the call stack)
= completed node
✓ = element has been processed
CSE 373 - Winter 2016
9

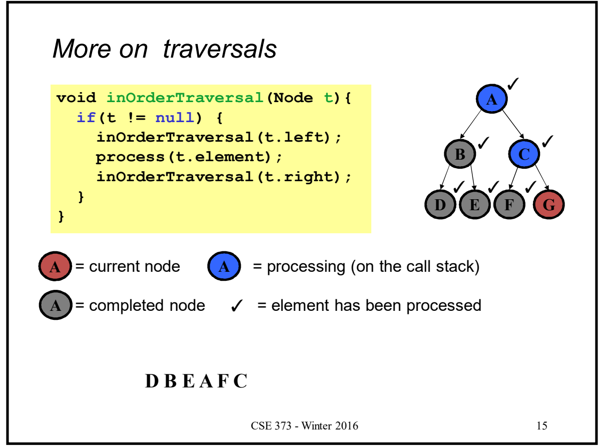

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
= current node
= processing (on the call stack)
= completed node
✓ = element has been processed
D
CSE 373 - Winter 2016
10

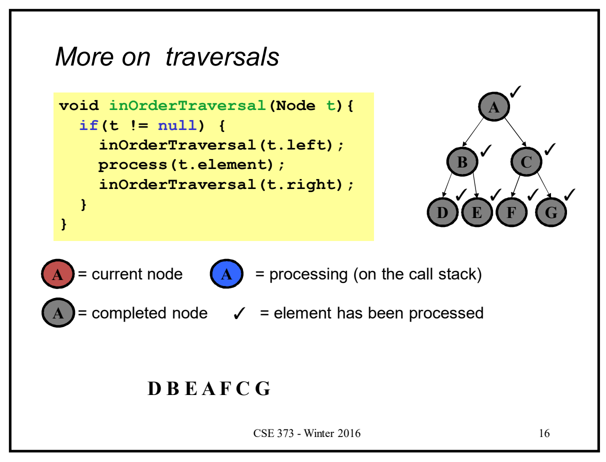

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
= current node
= processing (on the call stack)
= completed node
✓ = element has been processed
D B
CSE 373 - Winter 2016
11

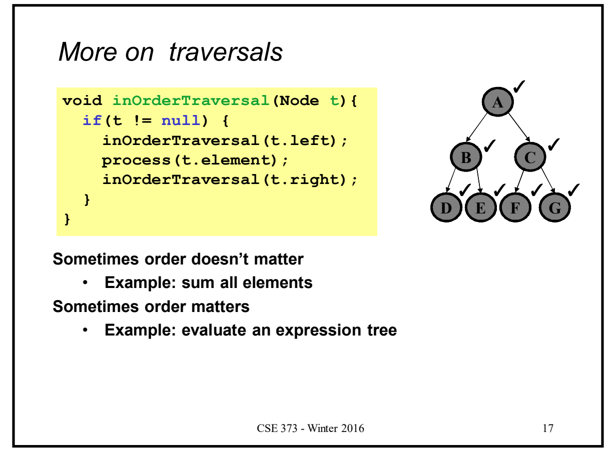

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
= current node
= processing (on the call stack)
= completed node
✓ = element has been processed
D B E
CSE 373 - Winter 2016
12

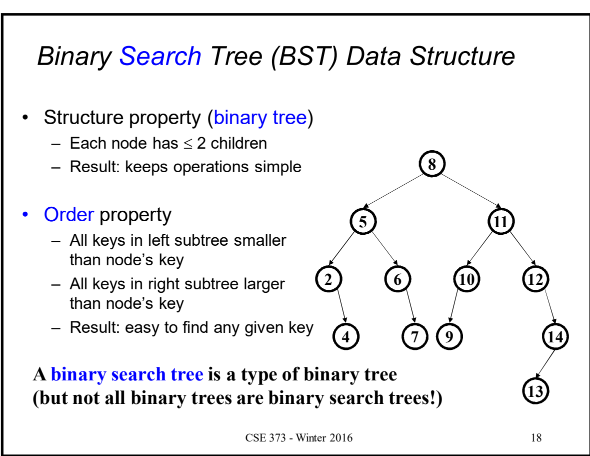

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
A
B
C
D
E
F
G
= current node
(on the call stack)
= processing
= completed node
= element has been processed
D B E A
CSE 373 - Winter 2016
13

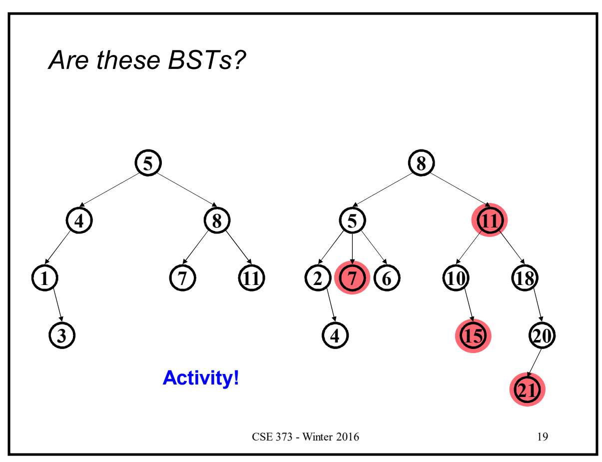

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
A
B
C
D
E
F
G
= current node
(on the call stack)
= processing
= completed node
= element has been processed
D B E A
CSE 373 - Winter 2016
14

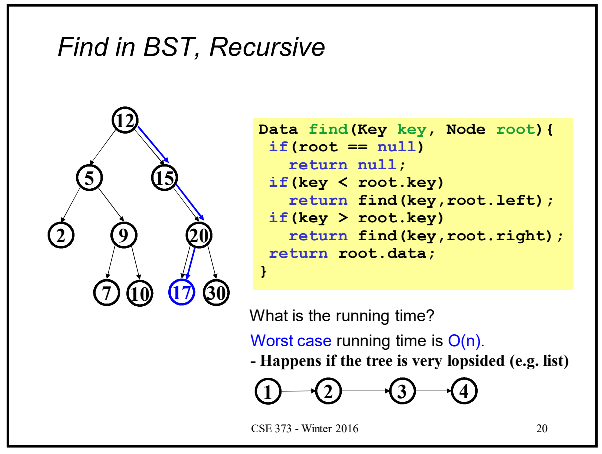

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
A
B
C
D
E
F
G
= current node
= processing (on the call stack)
= completed node
✓ = element has been processed
D B E A F C
CSE 373 - Winter 2016
15

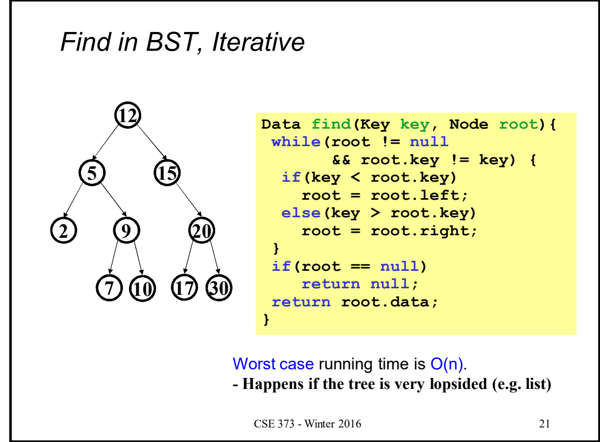

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
A
B
C
D
E
F
G
= current node
= processing (on the call stack)
= completed node
✓ = element has been processed
D B E A F C G
CSE 373 - Winter 2016
16

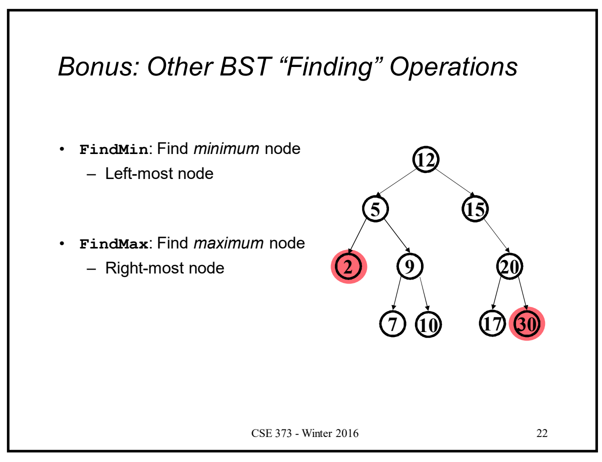

More on traversals
void inOrderTraversal(Node t){
if(t != null) {
inOrderTraversal(t.left);
process(t.element);
inOrderTraversal(t.right);
}
}
A
B
C
D
E
F
G
Sometimes order doesn't matter
Example: sum all elements
Sometimes order matters
Example: evaluate an expression tree
CSE 373 - Winter 2016
17

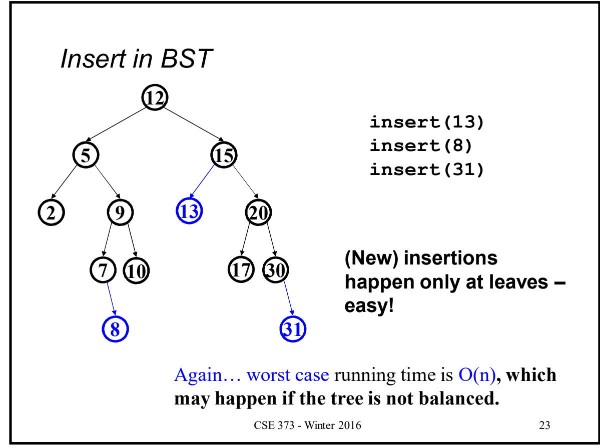

Binary Search Tree (BST) Data Structure
Structure property (binary tree)
Each node has ≤ 2 children
Result: keeps operations simple
Order property
All keys in left subtree smaller than node's key
All keys in right subtree larger than node's key
Result: easy to find any given key
8
5
11
2
6
10
12
4
7
9
14
13
A binary search tree is a type of binary tree
(but not all binary trees are binary search trees!)
CSE 373 - Winter 2016
18

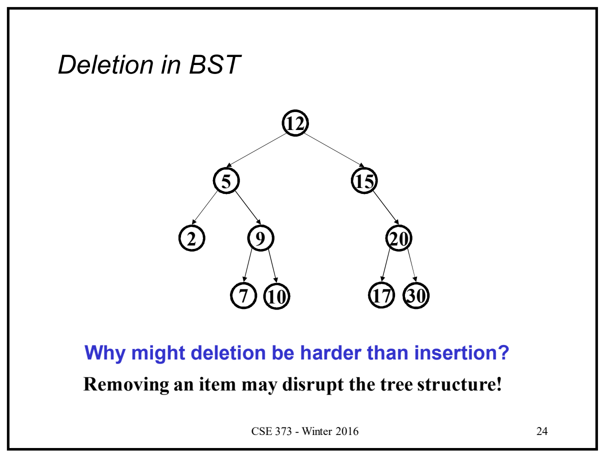

Are these BSTs?
5
4
8
1
7
11
3
8
5
11
2
7
6
10
18
4
15
20
21
Activity!
CSE 373 - Winter 2016
19


Find in BST, Recursive
12
5
15
2
9
20
7
10
17
30
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;
}
What is the running time?
Worst case running time is O(n).
- Happens if the tree is very lopsided (e.g. list)
1
2
3
4
CSE 373 - Winter 2016
20


Find in BST, Iterative
12
5
15
2
9
20
7
10
17
30
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;
}
Worst case running time is O(n).
- Happens if the tree is very lopsided (e.g. list)
CSE 373 - Winter 2016
21


Bonus: Other BST “Finding” Operations
FindMin: Find minimum node
– Left-most node
FindMax: Find maximum node
– Right-most node
12
5
15
2
9
20
7
10
17
30
CSE 373 - Winter 2016
22


Insert in BST
12
5
15
2
9
13
20
7
10
17
30
8
31
insert(13)
insert(8)
insert(31)
(New) insertions happen only at leaves – easy!
Again… worst case running time is O(n), which may happen if the tree is not balanced.
CSE 373 - Winter 2016
23


Deletion in BST
12
5
15
2
9
20
7
10
17
30
Why might deletion be harder than insertion?
Removing an item may disrupt the tree structure!
CSE 373 - Winter 2016
24

## *Deletion in BST*

- Basic idea: `find` the node to be removed, then "fix" the tree so that it is still a binary search tree
- Three potential cases to fix:
  - Node has no children (leaf)
  - Node has one child
  - Node has two children



## *Deletion – The Leaf Case*

`delete(17)`



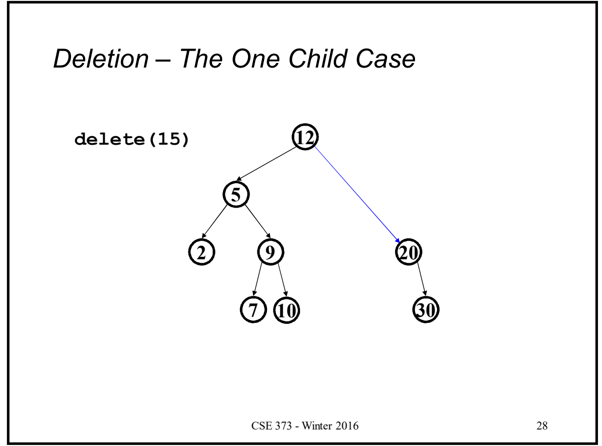
## *Deletion – The One Child Case*

`delete(15)`



## *Deletion – The One Child Case*

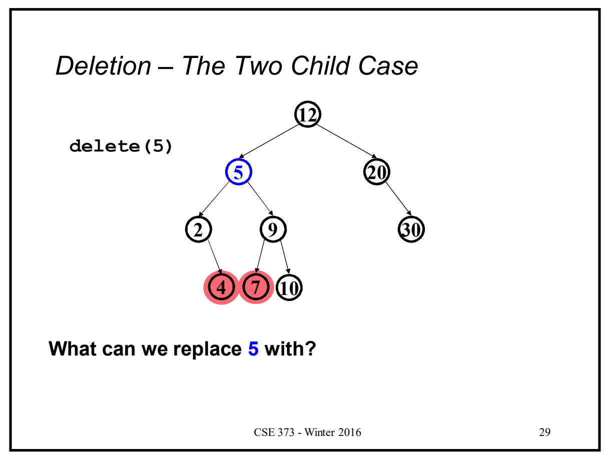
`delete(15)`






## *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

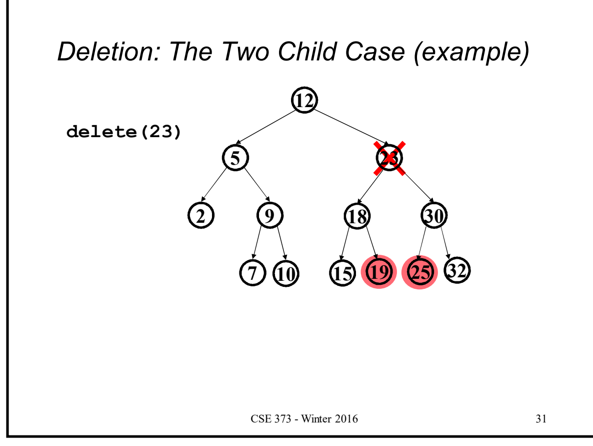
Options:

- *successor* minimum node from right subtree
  **`findMin(node.right)`**
- *predecessor* maximum node from left subtree
  **`findMax(node.left)`**

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

## Deletion: The Two Child Case (example)

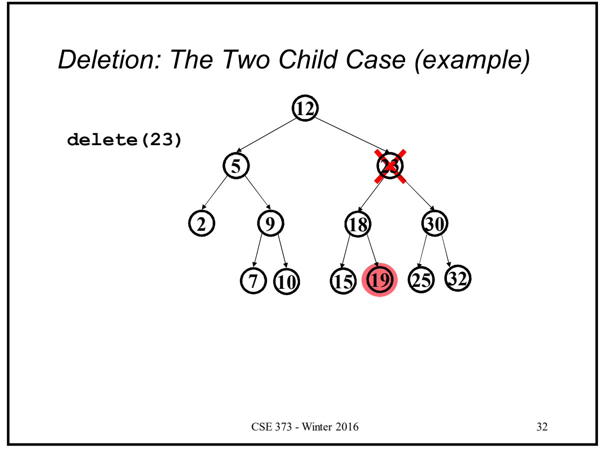
`delete(23)`






## Deletion: The Two Child Case (example)

`delete(23)`






## Deletion: The Two Child Case (example)

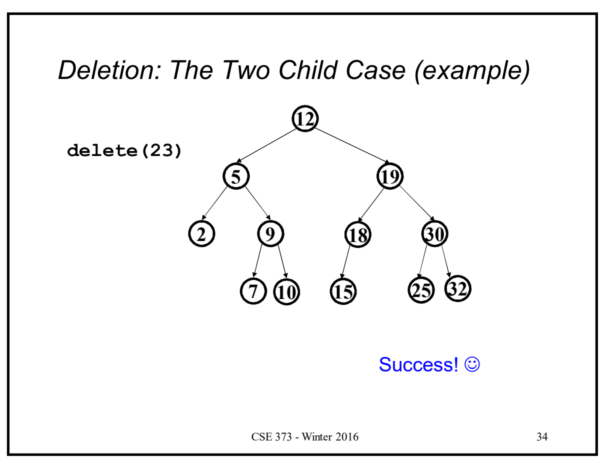
`delete(23)`

12
5 19
2 9 18 30
7 10 15 25 32



## Deletion: The Two Child Case (example)

`delete(23)`




Success! ☺



## 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 all, starting from an empty tree
- 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? ***$\Theta(n^2)$ Not a happy place***
  - Is inserting in the reverse order any better?

1
2
3

## *BuildTree for BST*

- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
- What we if could somehow re-arrange them
  - 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***