CSE 332
Data Structures & Parallelism

Dictionaries & Binary Search Trees

Melissa Winstanley
Spring 2024

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

Today

- Dictionaries
- Trees

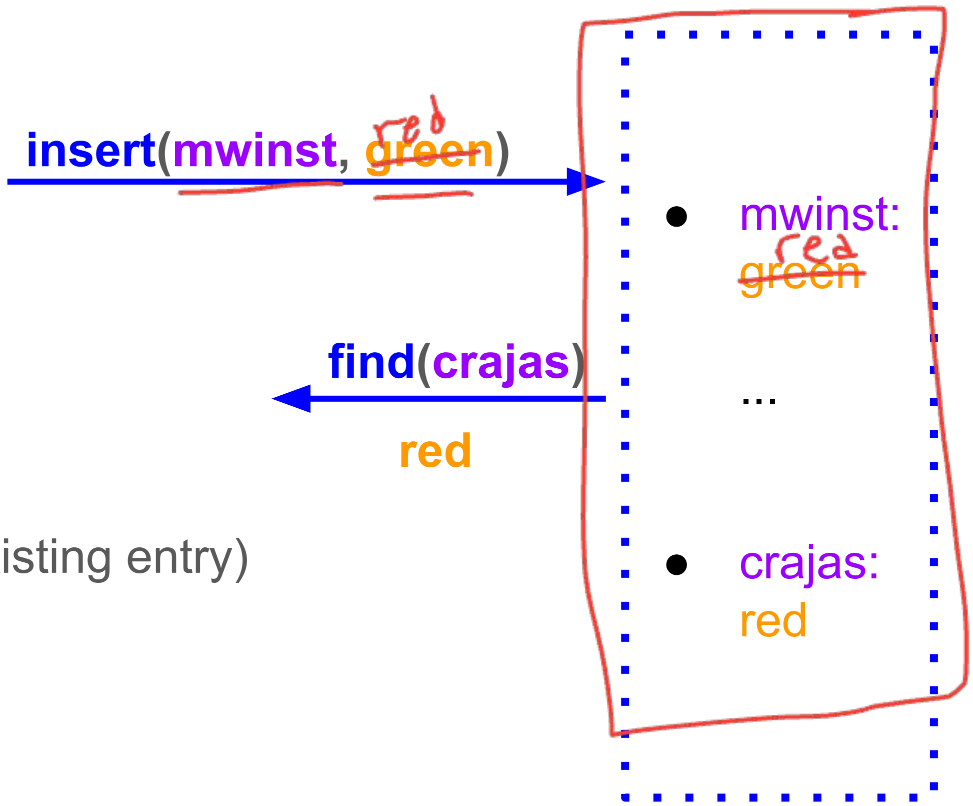
The Dictionary (aka Map) ADT

Data:

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

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`**, 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	~~$O(1)$~~ $O(n)$	$O(n)$	$O(n)$
Unsorted array	$O(n)$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$	$O(n)$

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

Lazy Deletion (e.g. in a sorted array) del(41)

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
- Deleted nodes waste 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

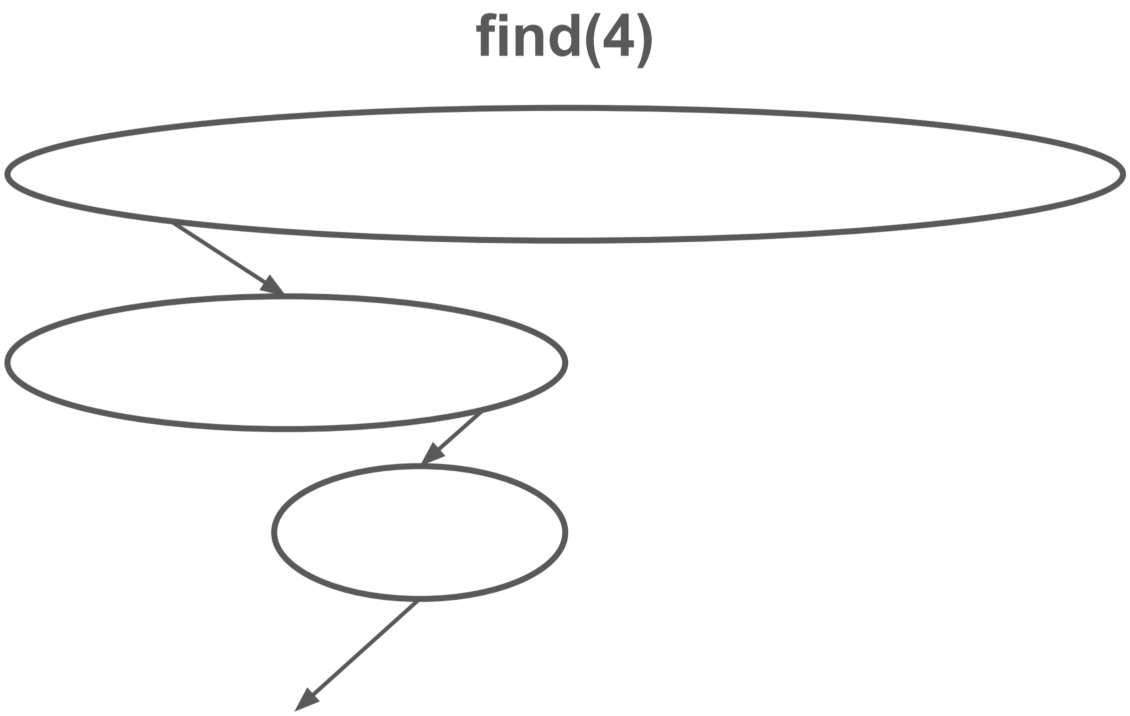
Skipping: Other balanced trees (red-black, splay)

Why Trees?

Trees offer speed ups because of their branching factors

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

Binary Search Tree



Why Trees?

Trees offer speed ups because of their branching factors

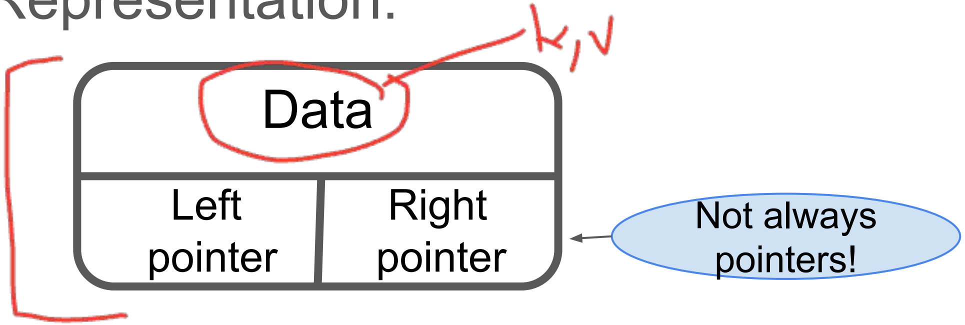
- Binary Search Trees are structured forms of *binary search*

Even a basic BST is fairly good

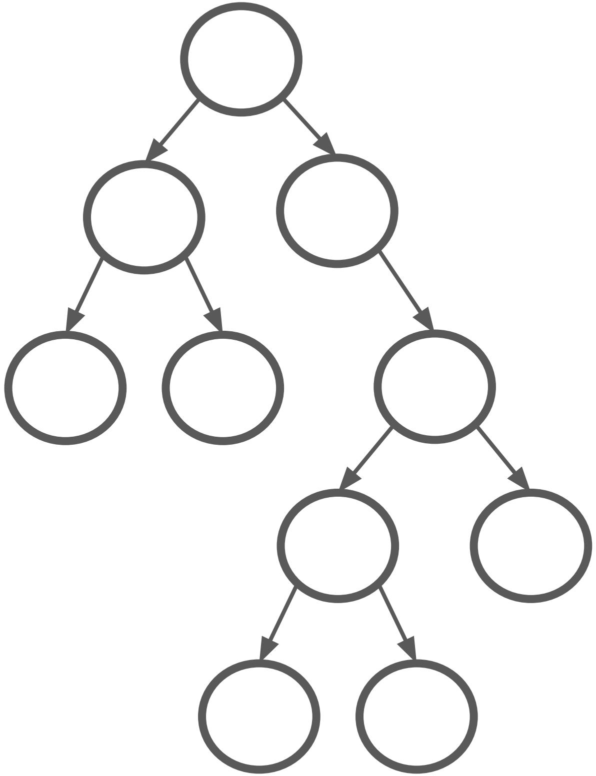
	Insert	Find	Delete
Worst-Case	O(n)	O(n)	O(n)
Average-Case	O(log n)	O(log n)	O(log n)

Binary Trees

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



- 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: 2^h
- max # of nodes: $2^{h+1} - 1$
- min # of leaves: 1
- min # of nodes: $h+1$

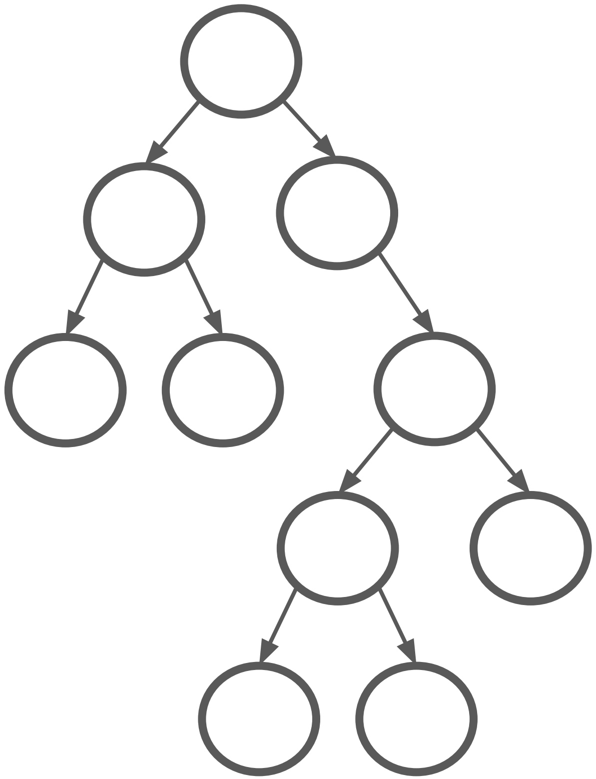
Calculating height

What is the height of a tree with root root?

```
int treeHeight(Node root) {

   ???

}
```



Calculating height

$T(n) = c + 2T(n/2)$

What is the height of a tree with root root?

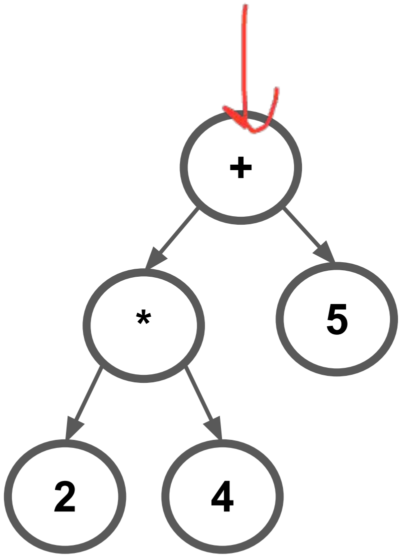
```
int treeHeight(Node root) {
  if(root == null)
    return -1
  return 1 + max(treeHeight(root.left),
                 treeHeight(root.right))
}
```

Running time for *n* nodes: O(*n*) (each node is visited once)

Note: non-recursive is painful - need your own stack of pending nodes. Much easier to use recursion

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 +

More on traversals

Sometimes order doesn't matter

- Example: sum all elements

Sometimes order matters

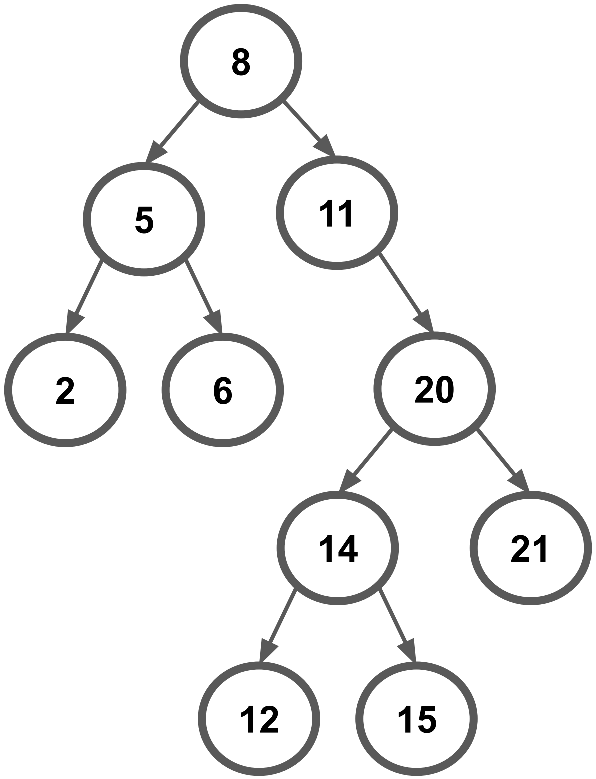
- Example: print tree with parent above indented children (pre-order)
- Example: evaluate an expression tree (post-order)

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

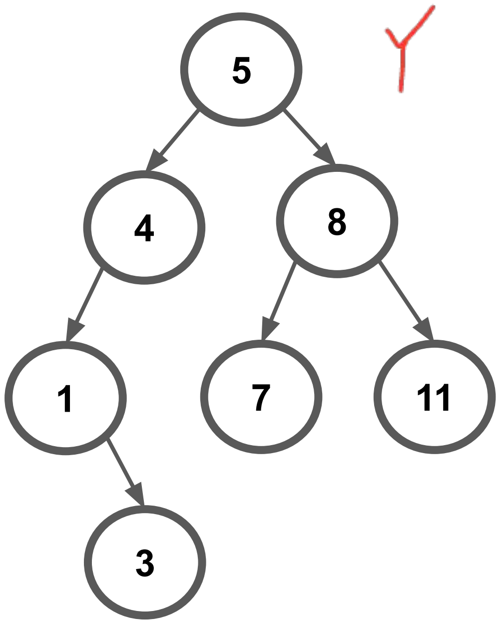
Binary Search Tree

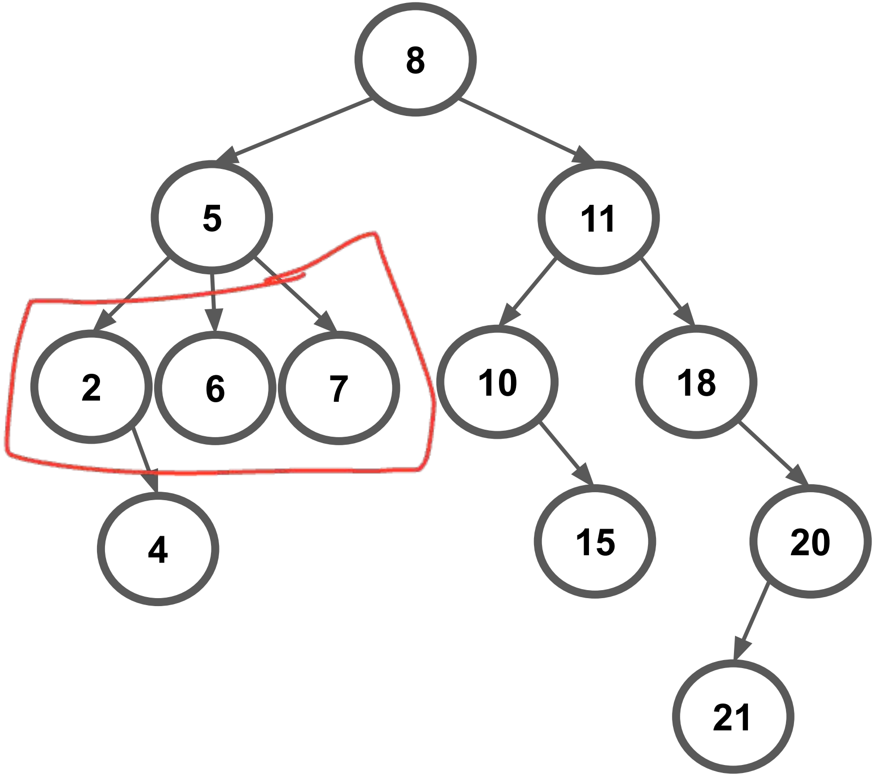
- Structural property ("binary")
  - each node has 2 children
  - result: keeps operations simple
- Order property
  - all keys in left subtree smaller than node's key
  - all keys in right subtree larger than node's key
  - result: easy to find any given key

No duplicates this time!

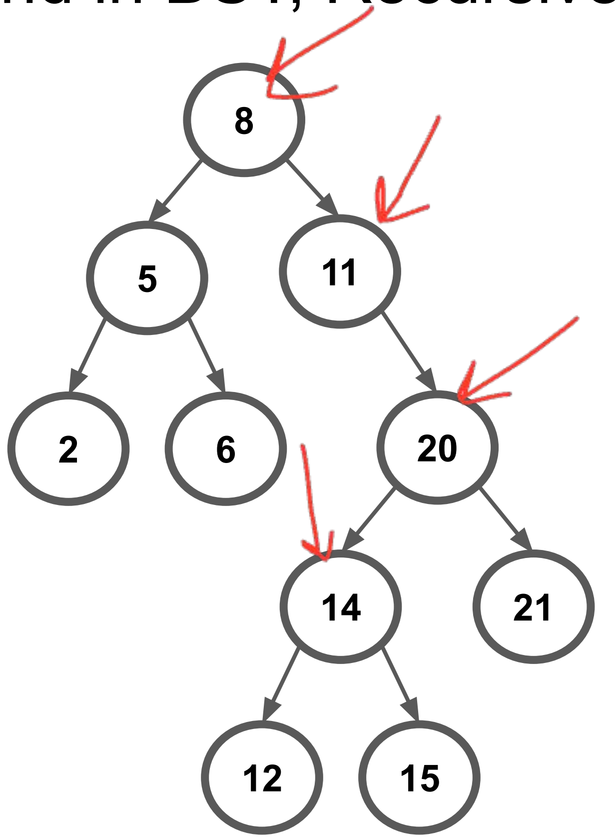


Are these BSTs?





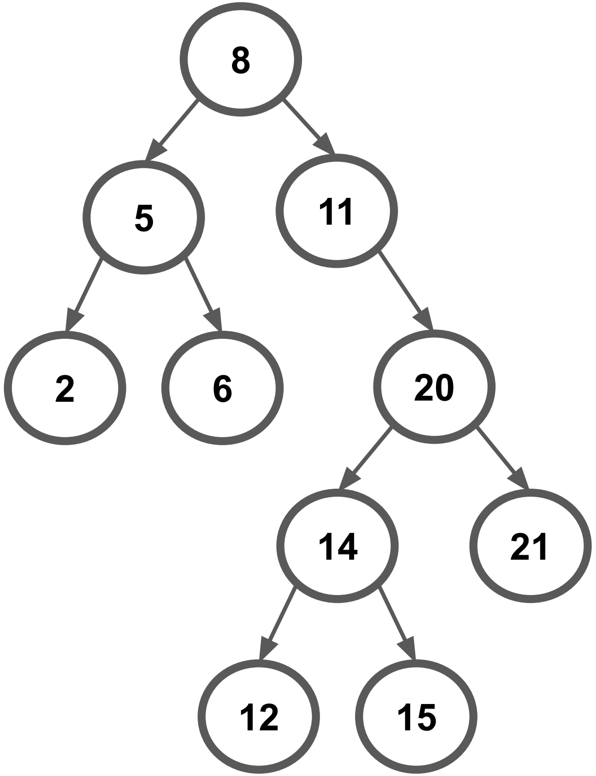
Find in BST, Recursive



find(14)

```
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

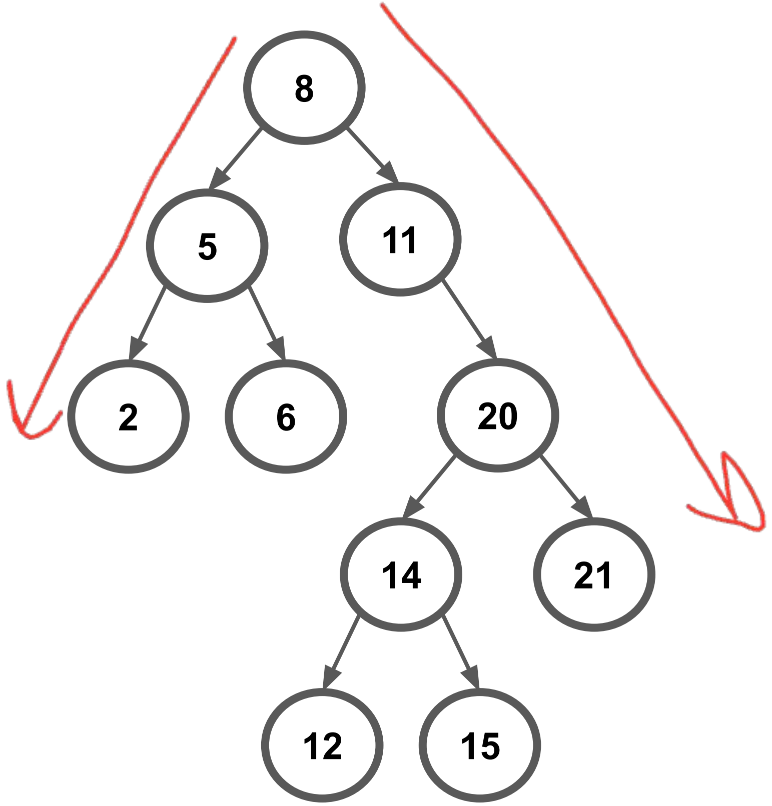



```
Data find(Key key, Node root) {
  while(root != null
        && root.key != key) {
    if(key < root.key)
      root = root.left;
    else
      root = root.right;
  }
  if(root == null)
    return null;
  return root.data;
}
```

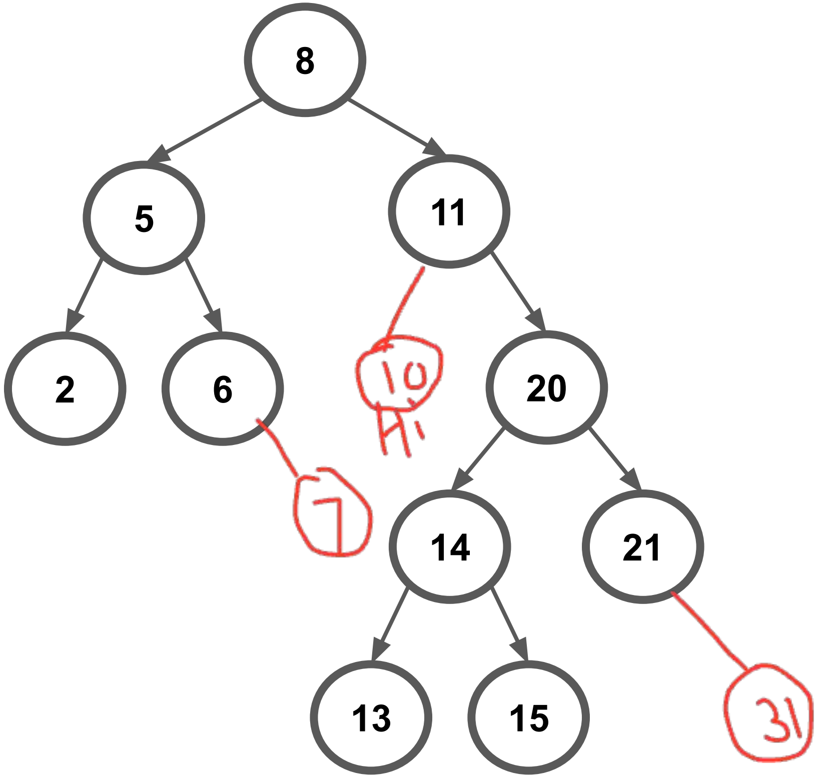
Other “finding operations”

- Find minimum node

- Find maximum node



Insert in BST

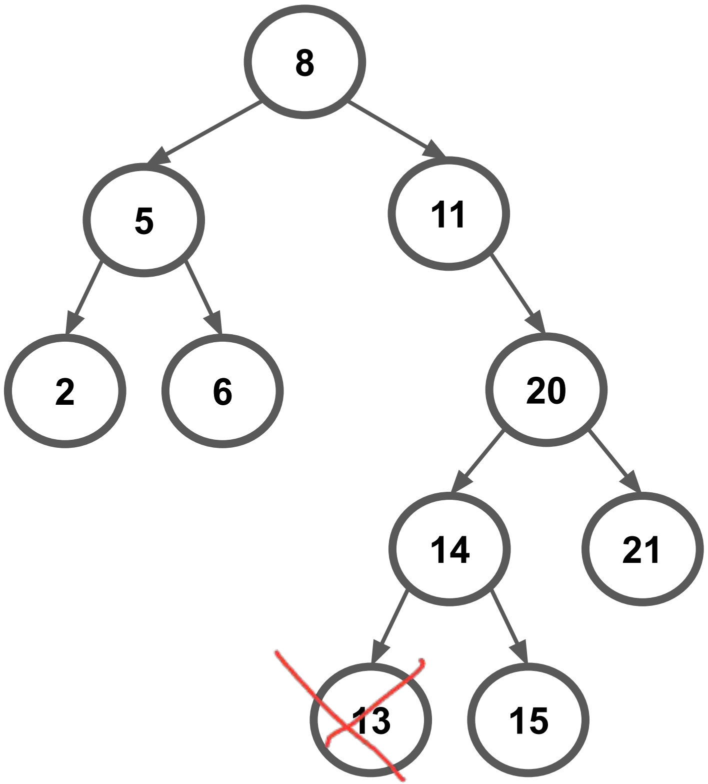



```
insert(10)
insert(7)
insert(31)
```

(New) insertions happen only at leaves - easy!

1. Find
2. Create a new node

Deletion in BST



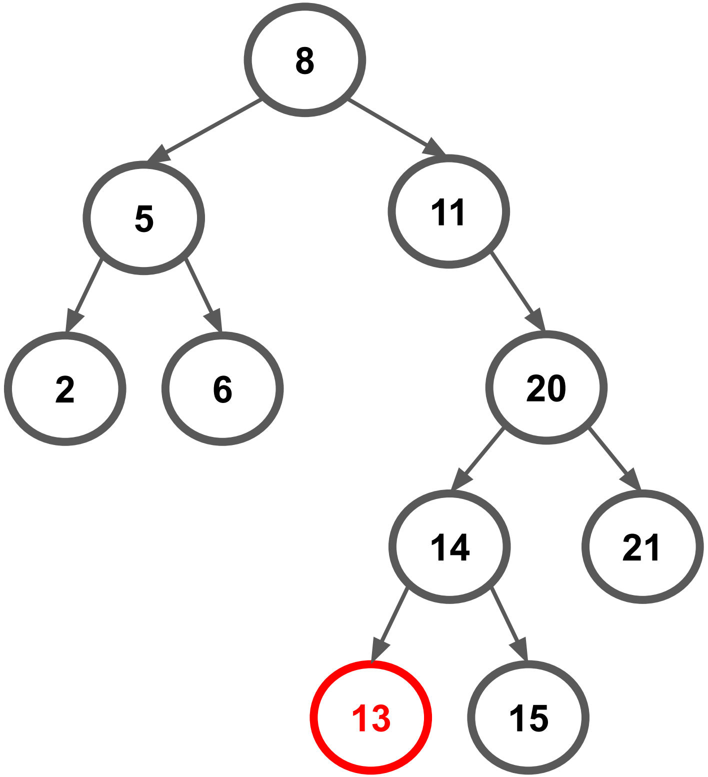
Why might deletion be harder than insertion?

delete(13)
delete(11)

Deletion

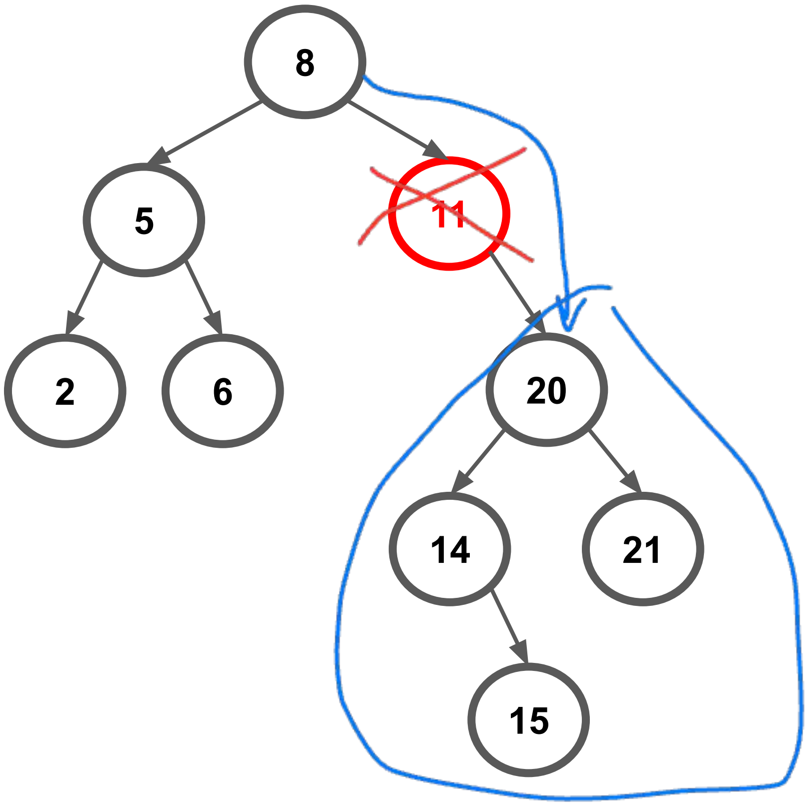
- Removing an item disrupts the tree structure
- Basic idea:
  - **find** the node to be removed
  - Remove it
  - “fix” the tree so that it is still a binary search tree
- Three cases:
  - node has no children (leaf)
  - node has one child
  - node has two children

Deletion - The Leaf Case



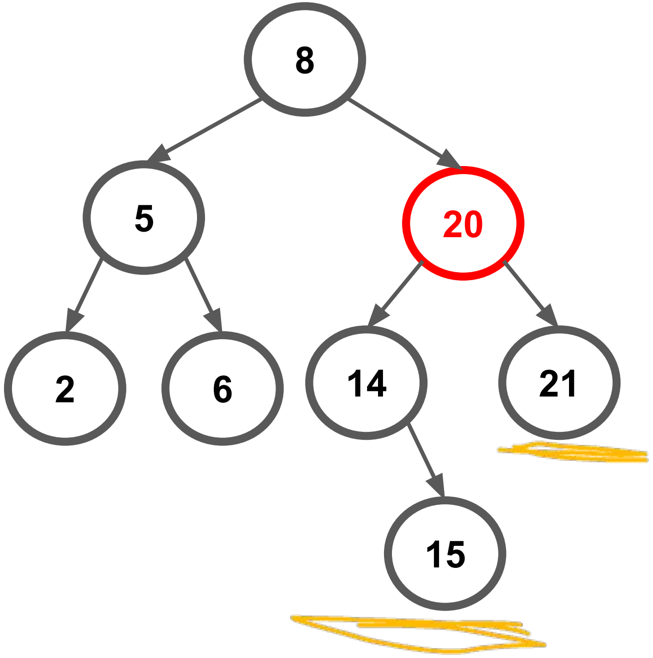

```
delete(13)
```

Deletion - The One Child Case



`delete(11)`

Deletion - The Two Child Case



`delete(20)`

What can we replace **20** with?

Deletion – The Two Child Case

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

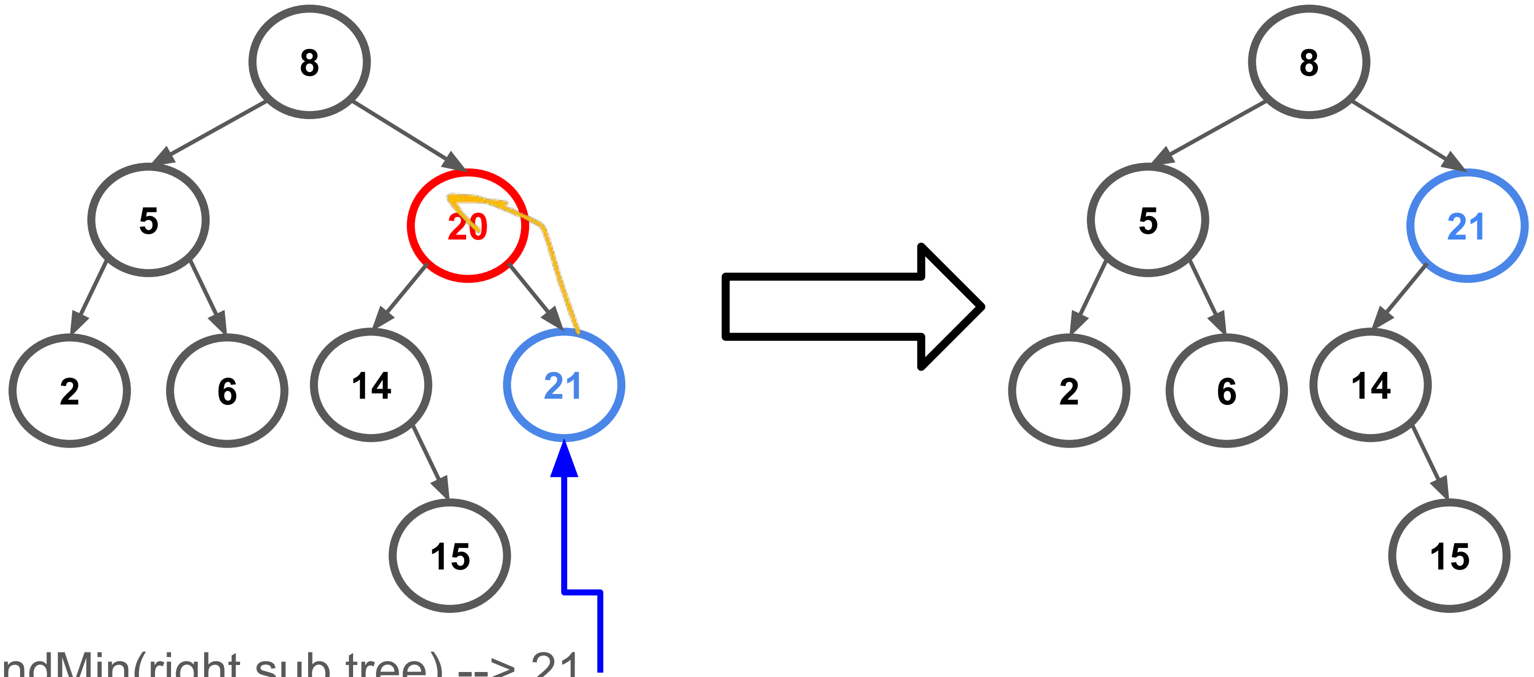
Options:

- *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!

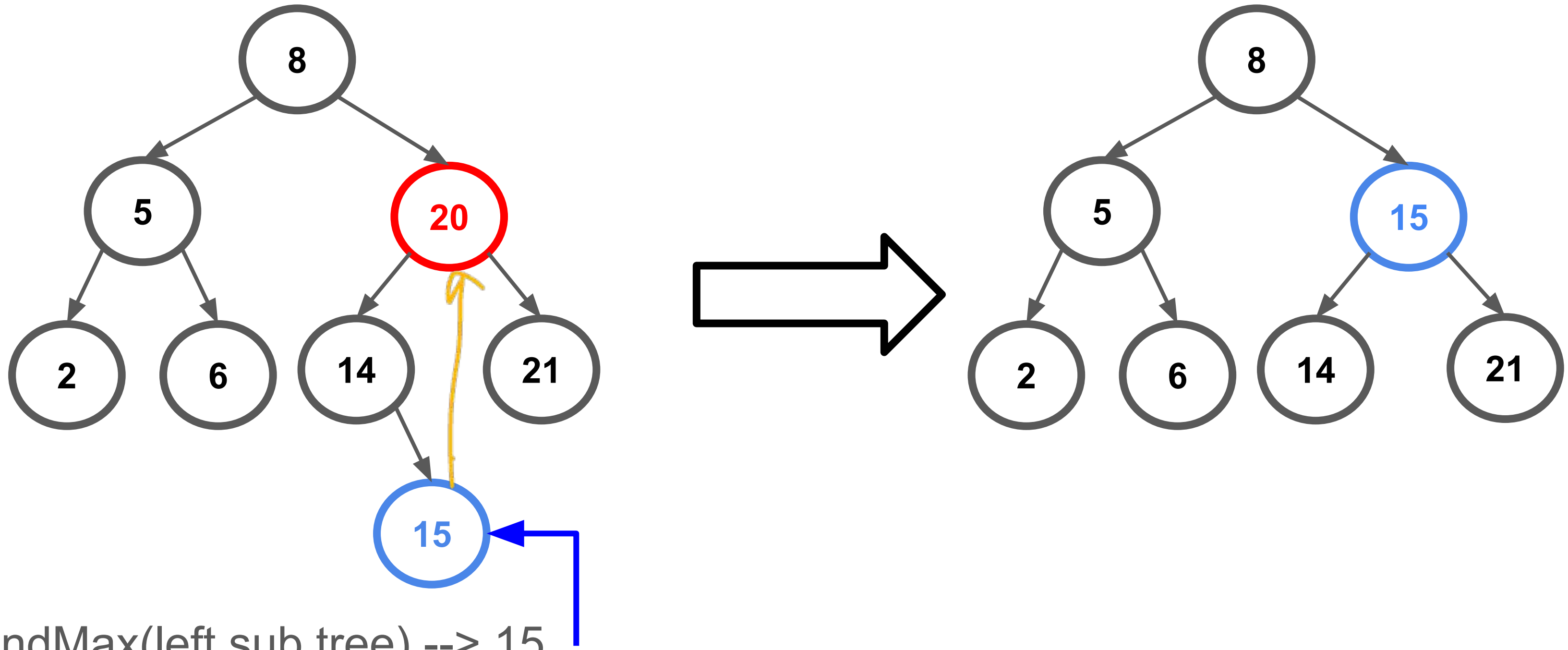
Deletion Using Successor



findMin(right sub tree) --> 21

`delete(20)`

Deletion Using Predecessor

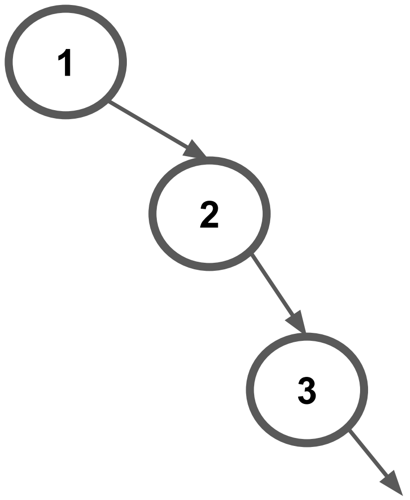


findMax(left sub tree) --> 15

`delete(20)`

`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?



Balanced BST

<u>Observation:</u>

- 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!