Binary Search Trees

CS16: Introduction to Data Structures & Algorithms

Summer 2021

New Homework 6

- **Optional**
- Out end of week, due June 28
- Mostly questions *reviewing* previous material
  - Good practice for midterm! (June 30-July 2)
- If you hand it in, will **replace** your lowest grade from HW1-5

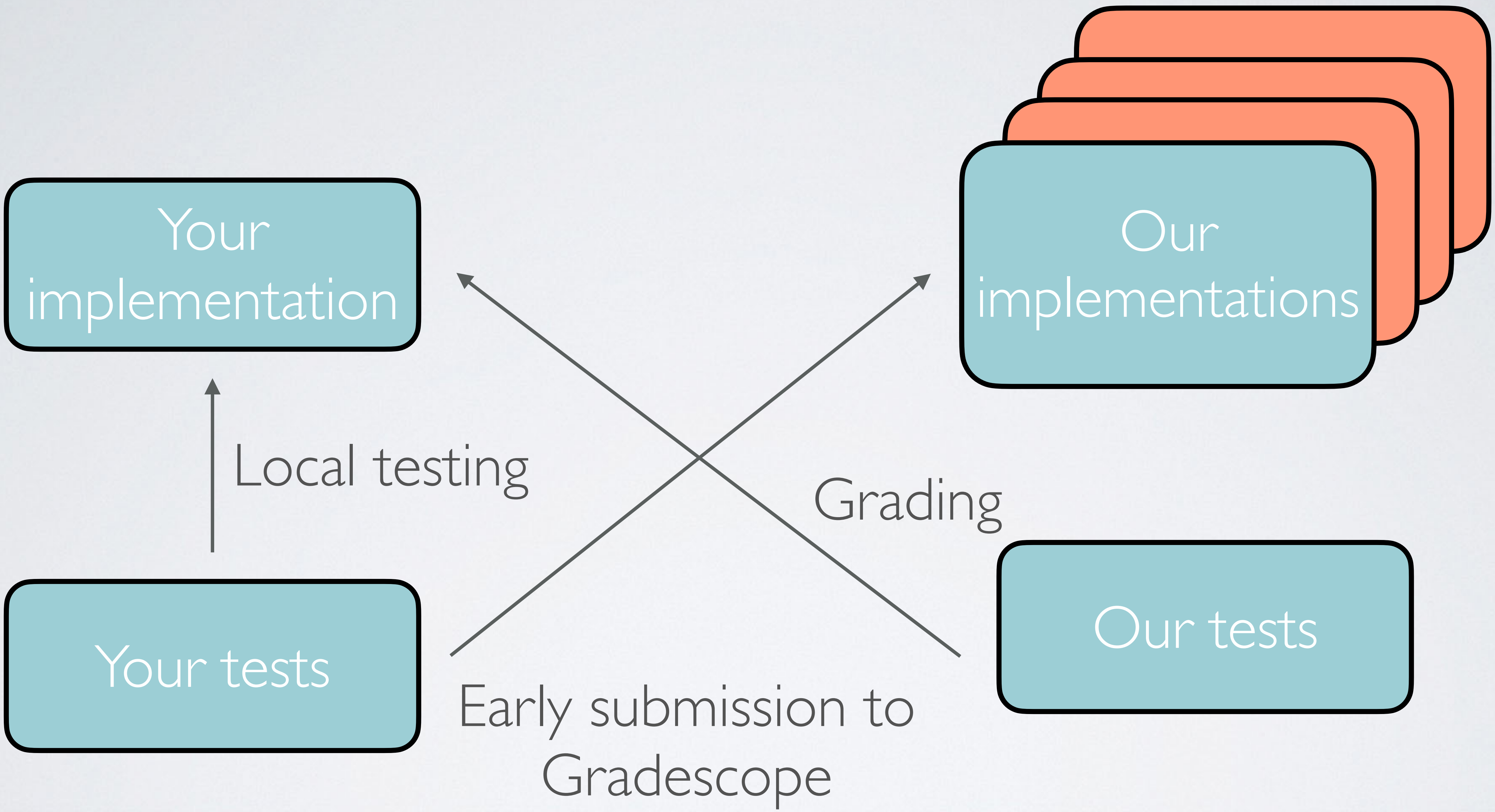
Other feedback

- Will post lecture slides before lecture
  - But—I encourage paper/pen (or possibly tablet equivalents) note-taking
- Concerns about cut material
  - **Not** cutting anything crucial
  - Will post old slides, etc.
- Leaving recording on after lecture
  - Not going to do this—"after-class" environment

Midterm

- Covers everything up through **today**, HW1-5
- Mostly open-ended problems (think written homeworks but somewhat simplified)
- Closed-book, closed-note
- Designed to take 1.5-2 hours, you'll have 3
- Available between June 30-July 2, online
- HW6 (optional) due June 28, next assignment out July 8
- Previous midterms available soon

A note on testing



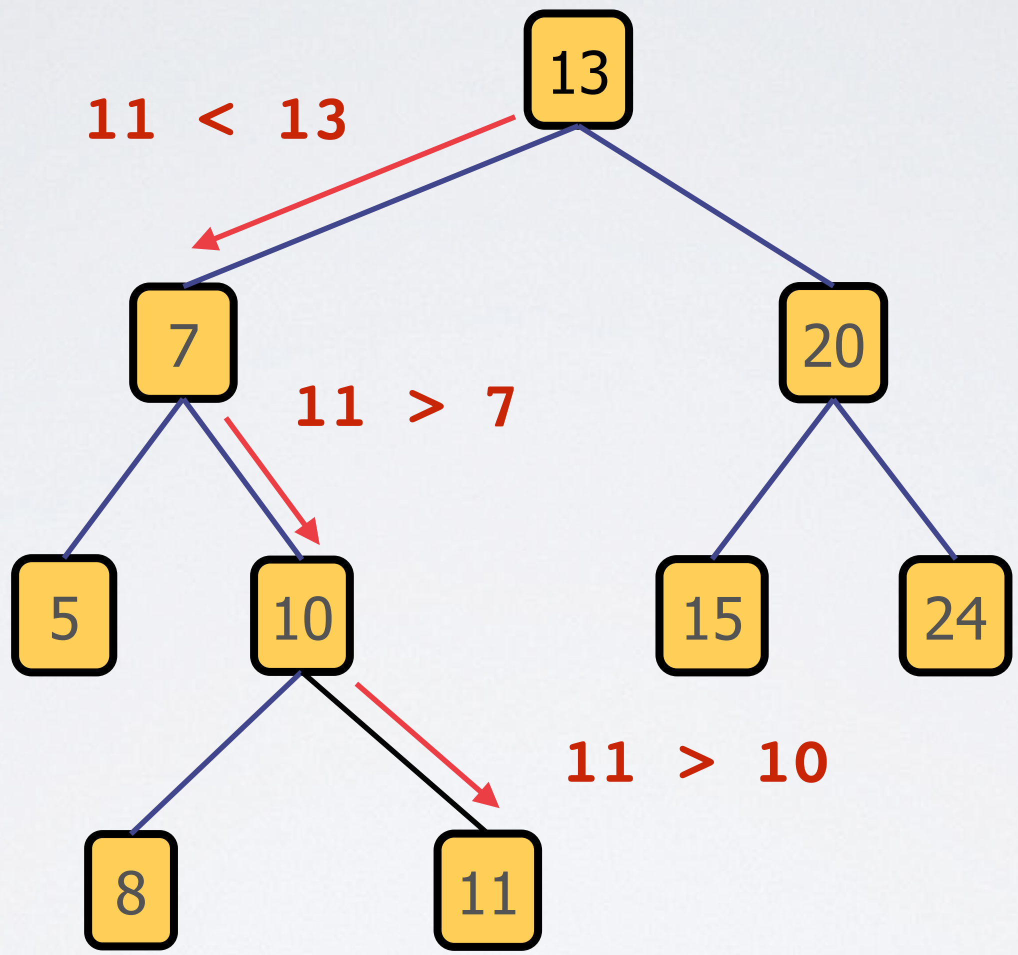
Problem solving session

- **Today** in my hours (will start around 2:45)
- We will solve a homework-style programming problem together
- Will demo good testing, problem-solving, debugging techniques
- Will be recorded but will be more fun/useful if there's an audience—can ask your debugging questions!

Binary Search Trees

- Binary trees with special property
  - For each node
    - left descendants have lower value than node
    - right descendants have higher value than node
- In-order traversal gives nodes in order

Searching a BST



- Find **11**
- Each comparison tells us whether to go left or right

Binary Search Tree — Find( )

```
function find(node, toFind):
   if node.data == toFind:
      return node

   else if toFind < node.data and node.left != null:
      return find(node.left, toFind)

   else if toFind > node.data and node.right != null:
      return find(node.right, toFind)

   return null
```

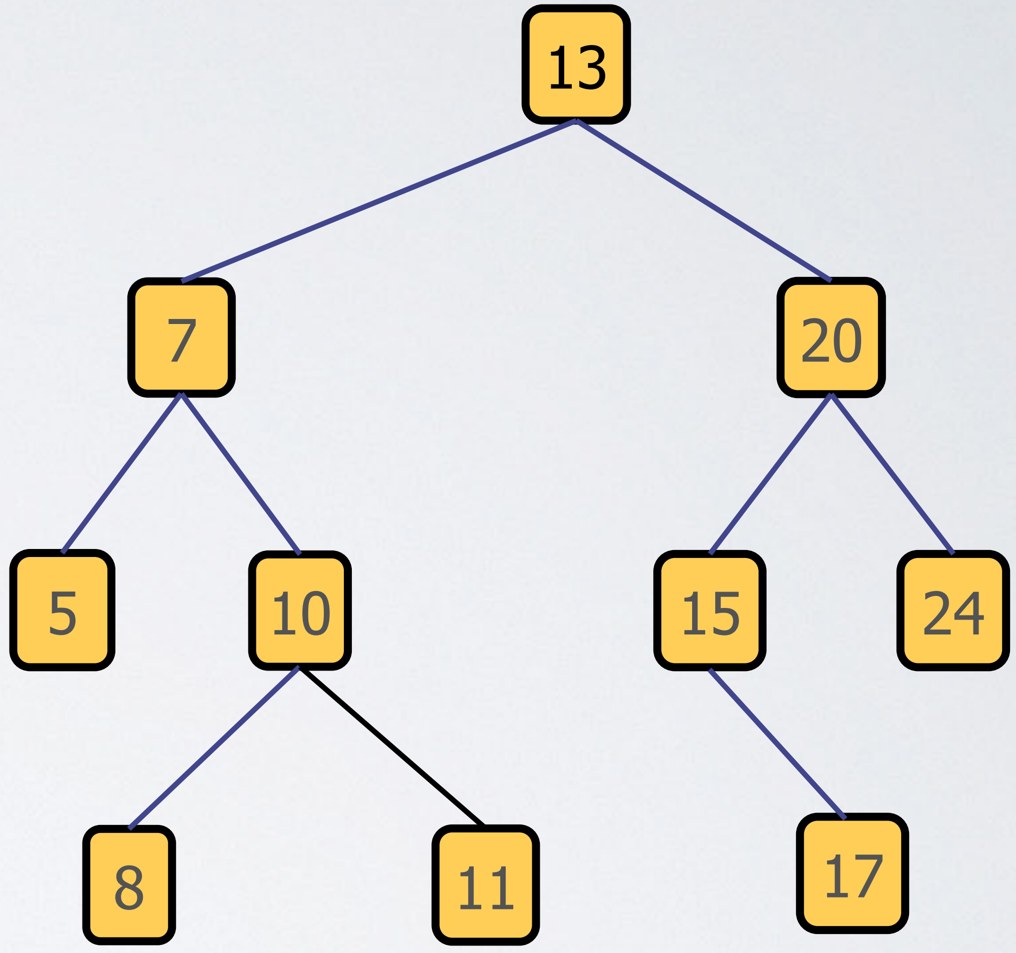
Binary Search Tree — Insert( )

```
function insert(node, toInsert):
  if node.data == toInsert:  # data already in tree
    return

  if toInsert < node.data:
    if node.left == null:    # add as left child
      node.addLeft(toInsert)
    else:
      insert(node.left, toInsert)
  else:
    if node.right == null:   # add as right child
      node.addRight(toInsert)
    else:
      insert(node.right, toInsert)
```

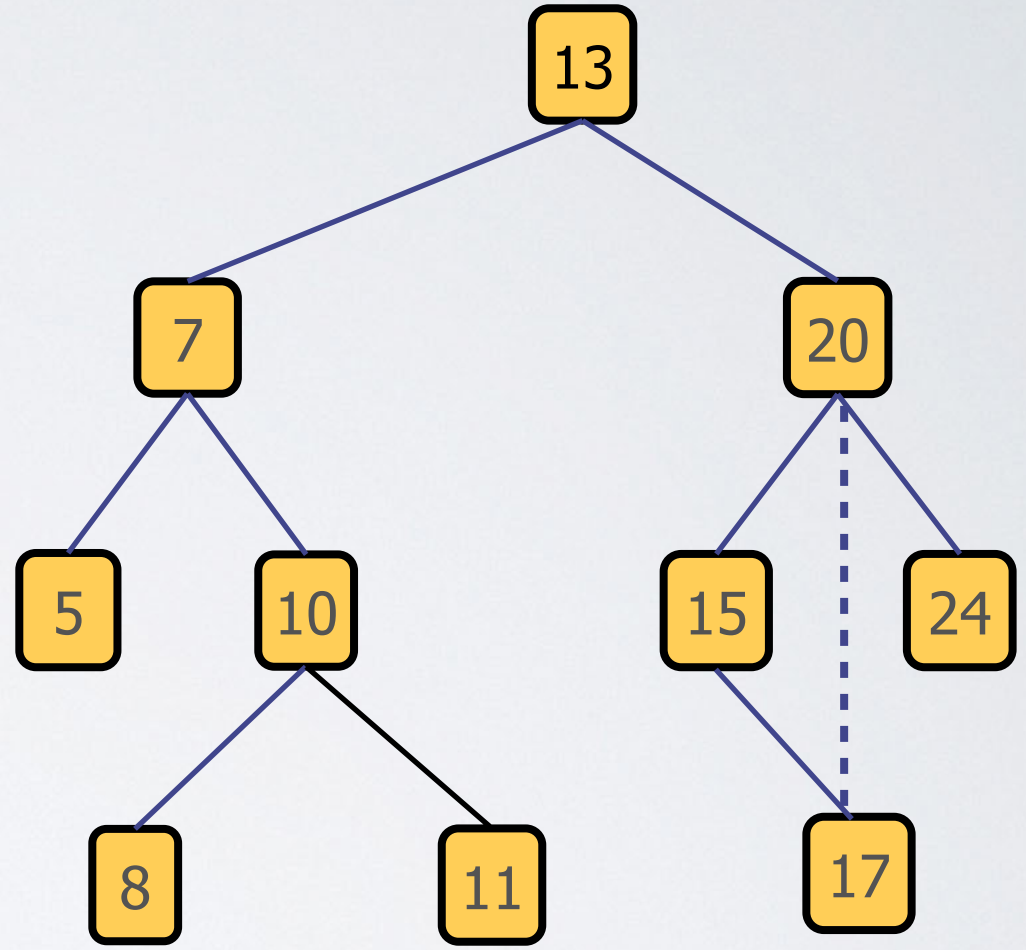
Removing from a BST

- Can be tricky
- Three cases to consider
  - Removing a leaf: easy, just do it
  - Removing internal node w/ `1` child (e.g., `15`)
  - Removing internal node w/ `2` children (e.g., `7`)



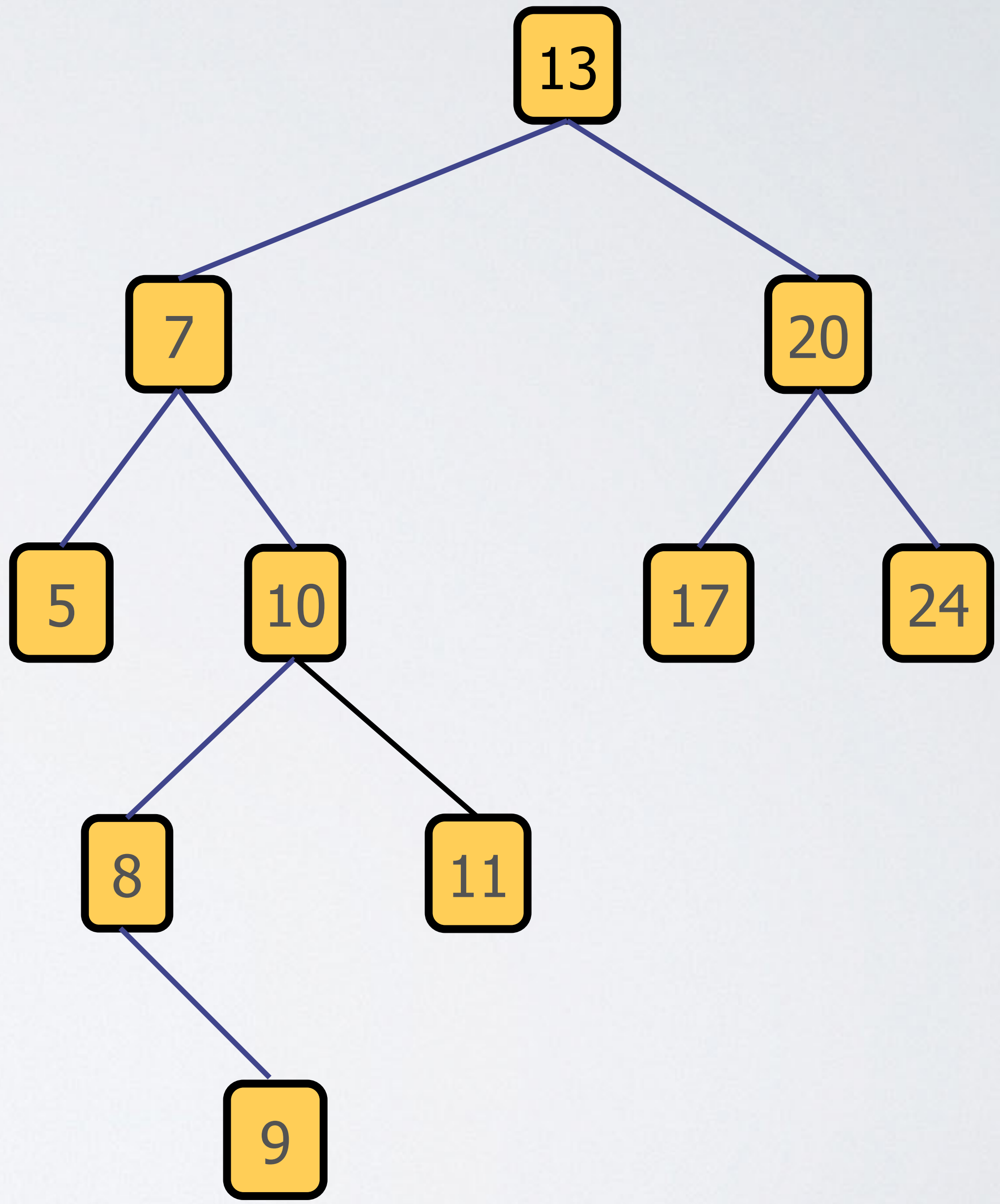
Removing from a BST - Case #2

- Removing internal node w/ **1** child
- Strategy
  - "Splice out" node by connecting its parent to its child
- Example: remove **15**
  - set parent's left child to 17
  - set 17's parent to 20
  - BST order is maintained



Removing from a BST - Case #3

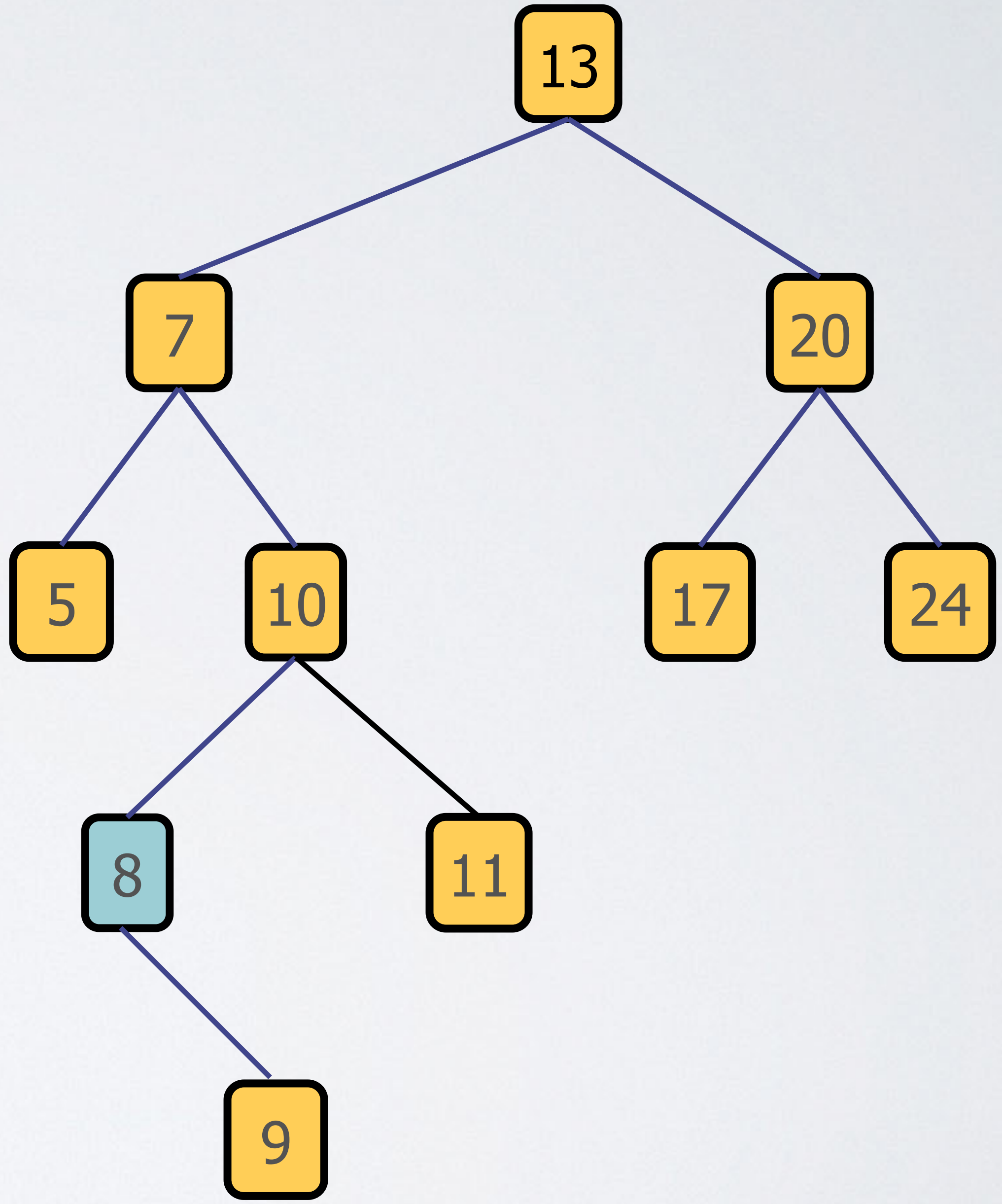
- Removing internal node w/ **2** children
- Replace node w/ successor
  - successor: next largest node
- Delete successor
  - Successor a.k.a. the in-order successor
- Example: remove **7**
  - What is successor of **7**?



Removing from a BST - Case #3

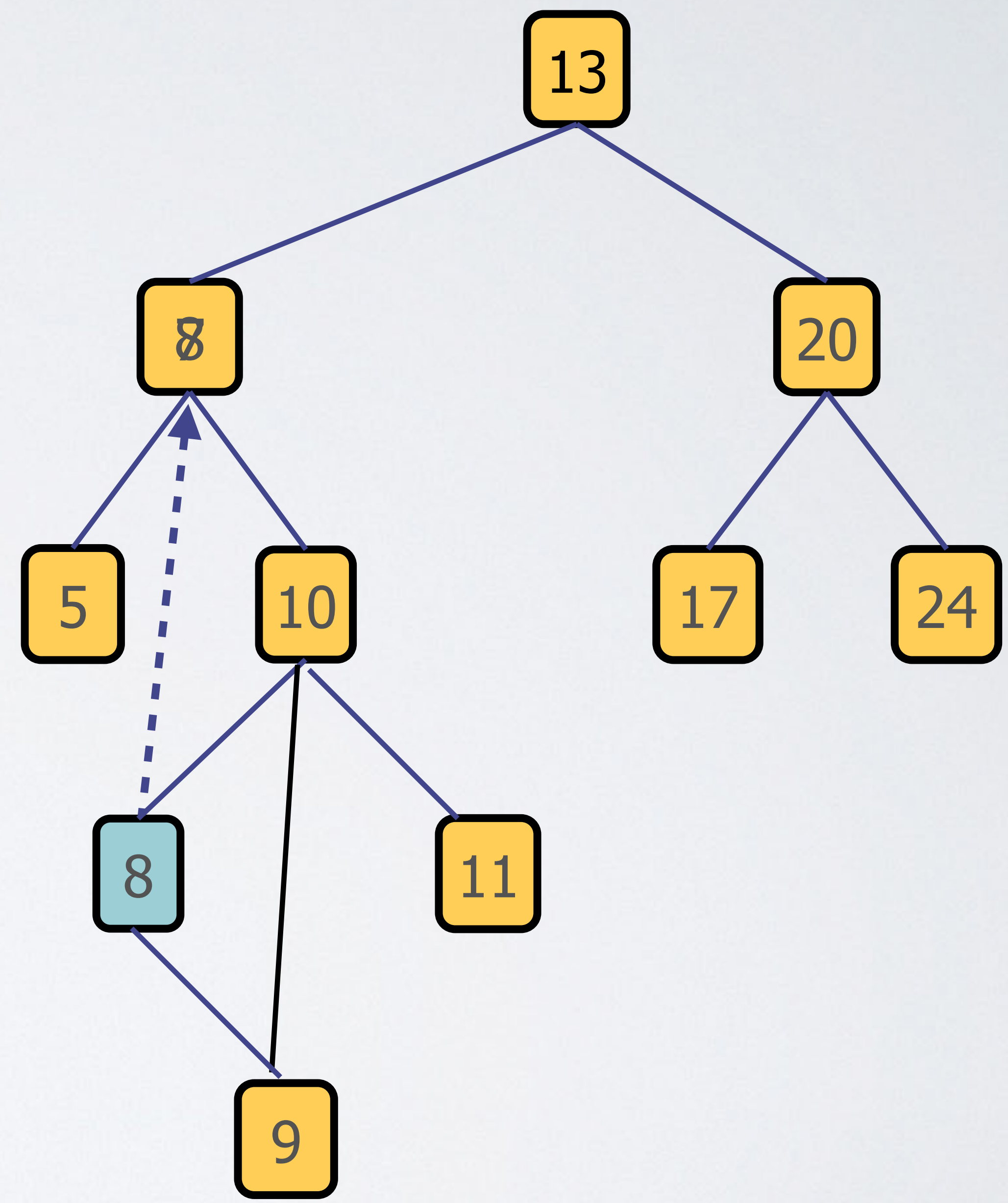
- Since node has **2** children…
  - …it has a right subtree
- Successor is leftmost node in right subtree
- **7**'s successor is **8**

```
successor(node):
   curr = node.right
   while (curr.left != null):
      curr = curr.left
   return curr
```



Removing from a BST - Case #3

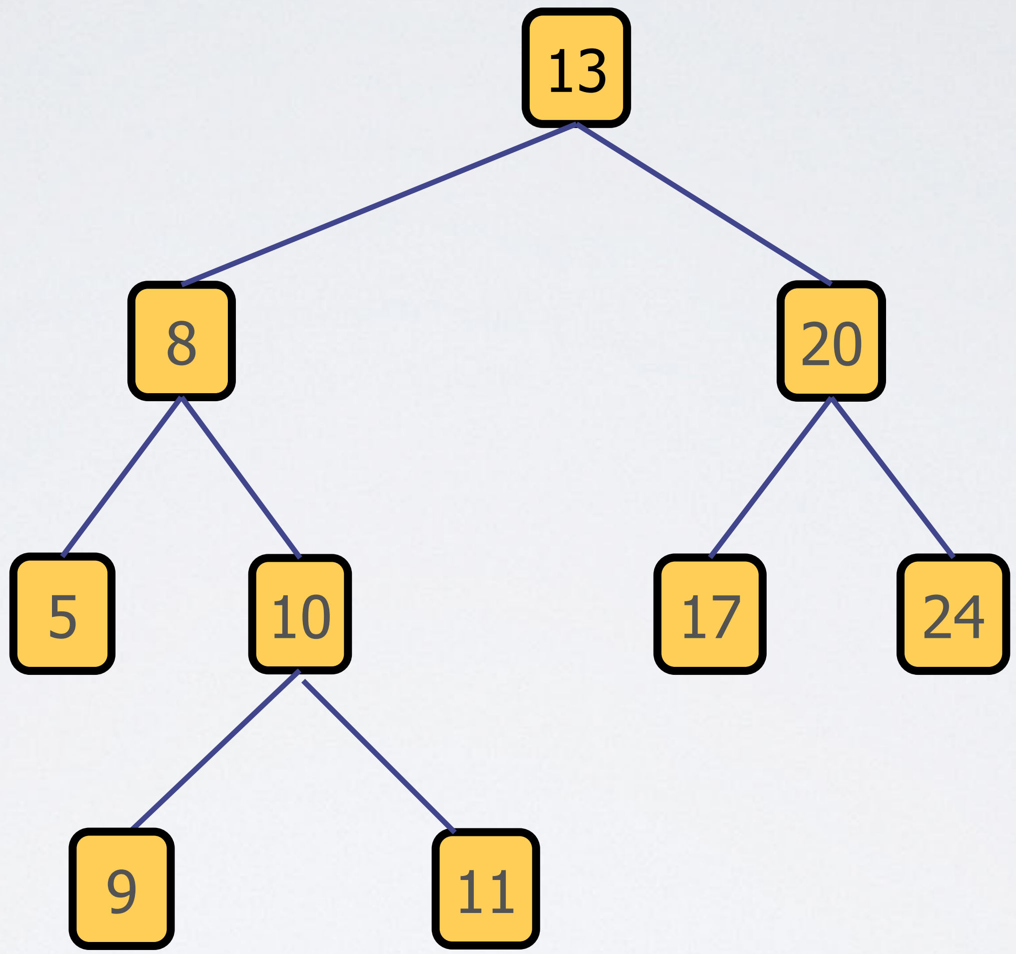
- Now, replace node with successor
- Observation
  - Successor can't have left sub-tree
  - …otherwise its left child would be successor
  - so successor only has right child
- Remove successor using Case #1 or #2
  - Here, use case #2 (internal w/ 1 child)
- Successor removed and BST order restored



Binary Search Tree — Remove( )

```
function remove(node):
  if node has no children:  # case 1
    node.parent.removeChild(node)
  else if node only has left child:   # case 2a
    if node.parent.left == node:  # node is a left child
      node.parent.left = node.left
    else:
      node.parent.right = node.left
  else if node only has right child:  # case 2b
    if node.parent.left == node:
      node.parent.left = node.right
    else:
      node.parent.right = node.right
  else:  # case 3 (node has two children)
    nextNode = successor(node)
    node.data = nextNode.data  #replace w/ nextNode
    remove(nextNode) # nextNode has at most one child
```

Binary Search Tree — Remove( )



Remove 13

Successor vs. Predecessor

- In Remove( )
  - OK to remove **in-order predecessor** instead of in-order successor
- Randomly picking between the two keeps tree balanced
- In Case #3
  - Predecessor is rightmost node of left subtree

Implementing Set

- Store set elements in BST, one per node
- **`add`**(`object`):
  - insert object into BST at the right place
- **`remove`**(`object`):
  - remove object from BST
- `boolean` **`contains`**(`object`):
  - search BST for object

Which objects?

- Say we have a kind of object we want to store in our Set
  - e.g. integers, strings, or a class we've build
- What do we need in order to use a hash-based set?
- A hash function!
- What about a BST?

Which objects?

- Say we have a kind of object we want to store in our Set
  - e.g. integers, strings, or a class we've build
- What do we need in order to use a hash-based set?
- A hash function!
- What about a BST?
  - Need an ordering on elements

Range queries

- Additional operation on sets
- **`between`**(`object1, object2`):
  - returns all items **`o`** where

    `object1 <= o < object2`
- How to implement with a hash-based set?

Range queries

- Additional operation on sets
- **`between`**(`object1, object2`):
  - returns all items **`o`** where

    `object1 <= o < object2`
- How to implement with a hash-based set?
  - Have to look at all items, O(n) (where n is the size of the set)

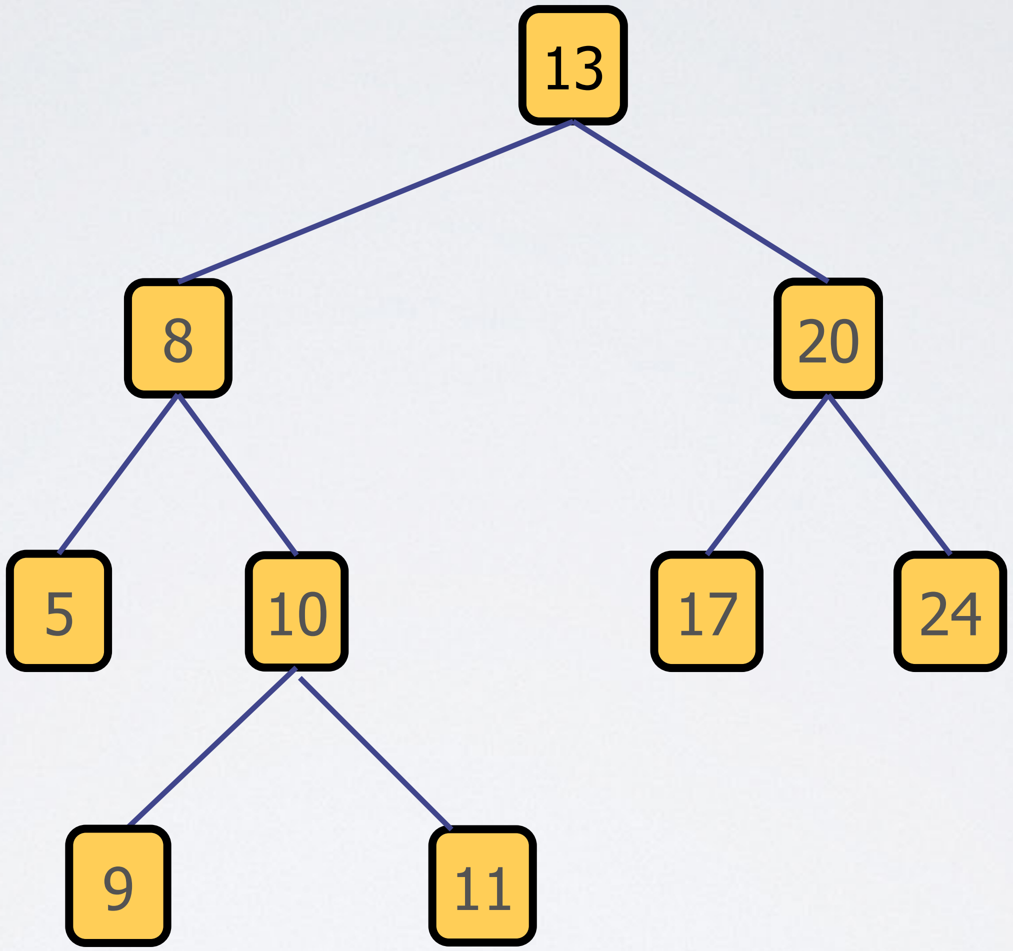
Range queries

- Additional operation on sets
- **`between`**(`object1, object2`):
  - returns all items **`o`** where

    `object1 <= o < object2`
- How to implement with a hash-based set?
  - Have to look at all items, O(n)
    (where n is the size of the set)

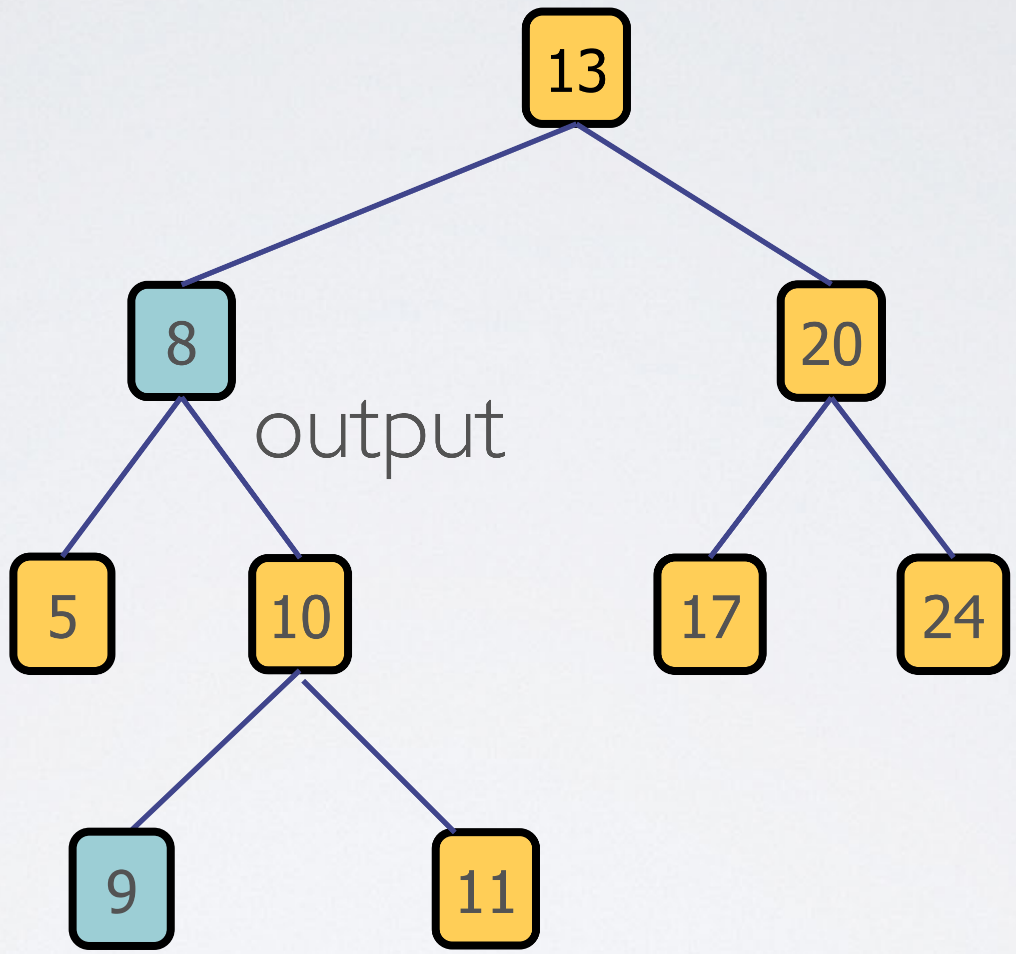
Range queries

between(6, 10)

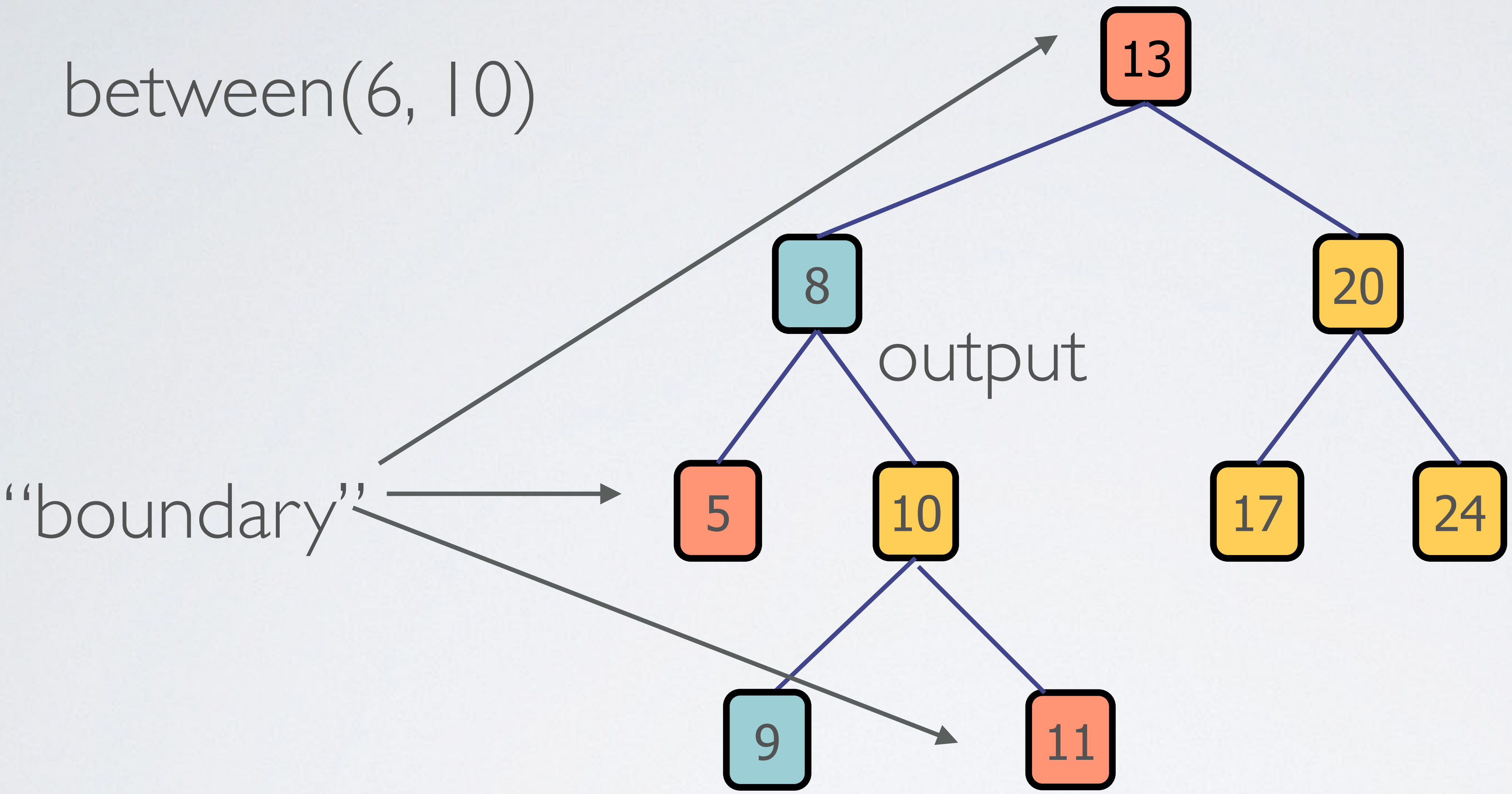


Range queries

between(6, 10)



Range queries



Binary Search Tree — between( )

```
function between(node, object1, object2):
  if object1 <= node.data < object2:
    output node.data
    if node has left child:
      between(node.left, object1, object2)
    if node has right child:
      between(node.right, object1, object2)
  else if node.data >= object2 and node has left child:
    between(node.left, object1, object2)
  else if node.data < object1 and node has right child:
    between(node.right, object1, object2)
```

Range queries

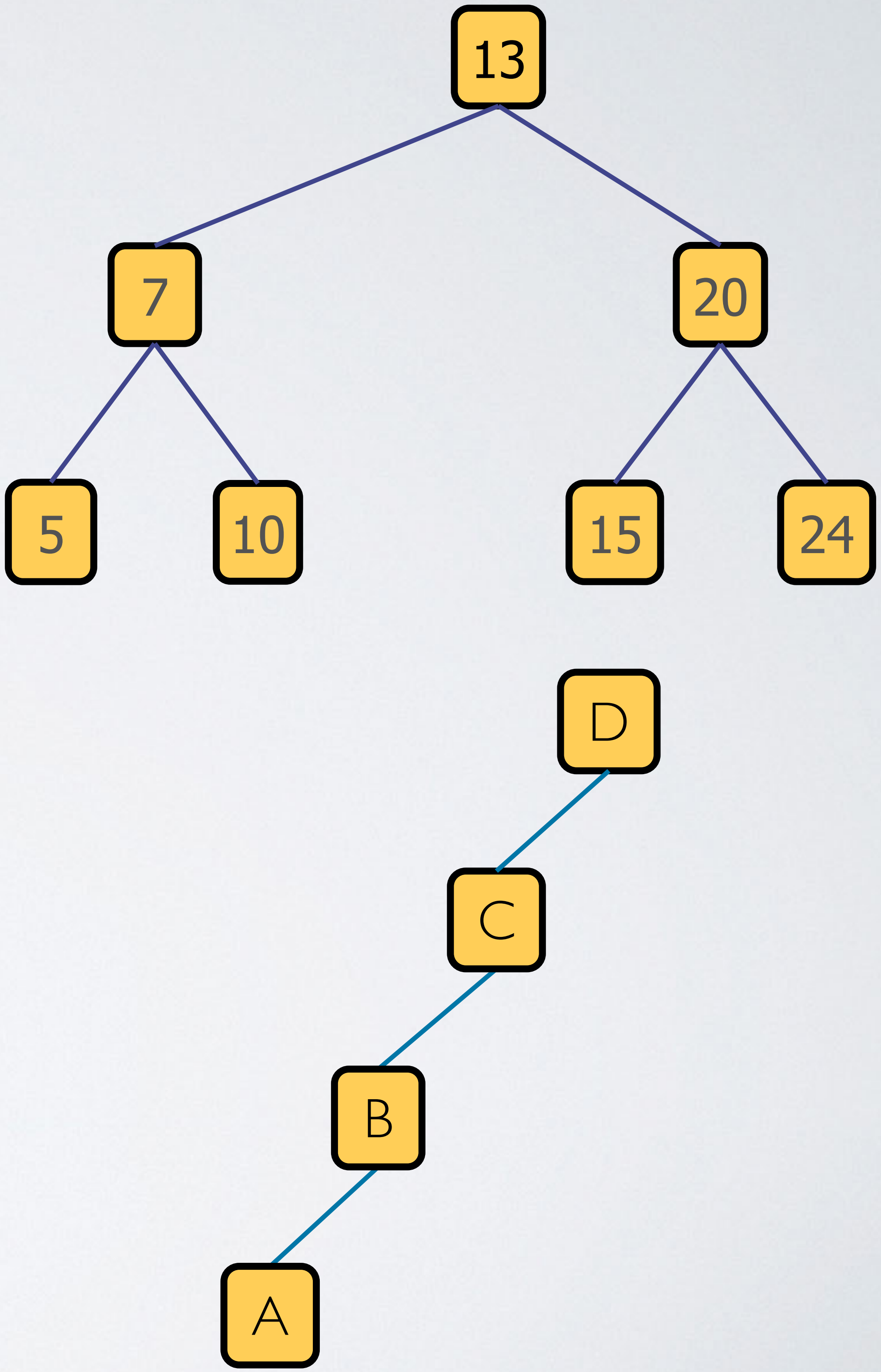
- What's the worst-case runtime of **`between`**(`object1, object2`) on a tree-based set with n elements?
- Depends on the **output** size
- Definitely at least O(m) if m elements between object1 and object2
- Turns out to be O(m + tree height)

Implementing Dictionary

- Just like with hashing, can implement Dictionary as well as Set
- Store keys and values at nodes, use keys as ordering

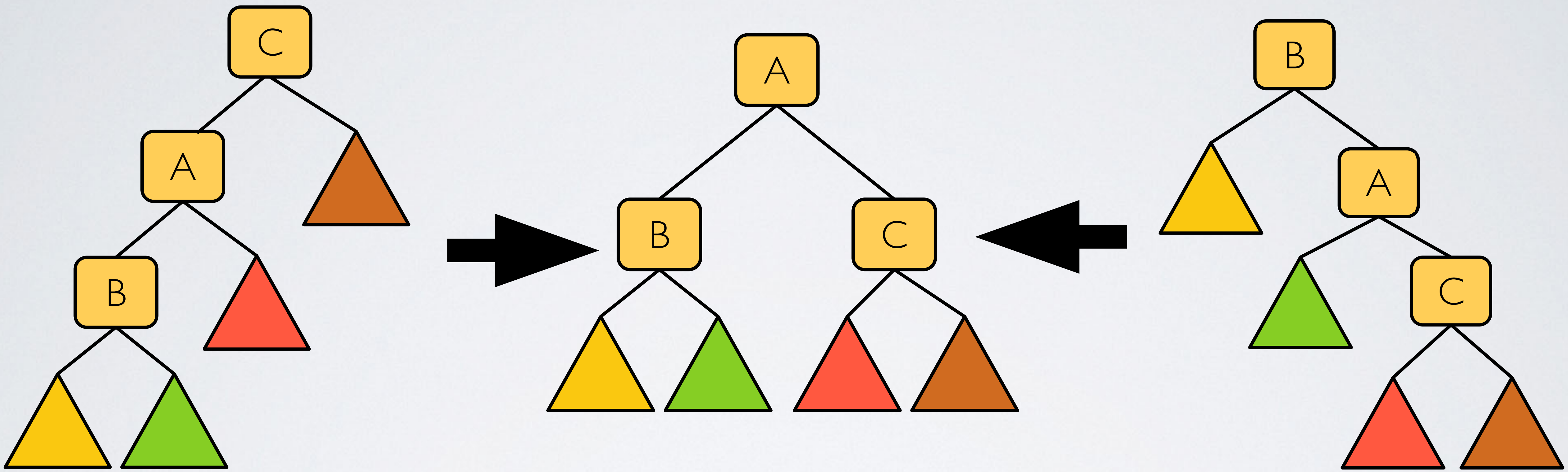
Binary Search Tree Analysis

- How fast are BST operations?
  - Given a tree, what is the worst-case node to find/remove?
- What is the best-case tree?
  - a balanced tree
- What is the worst-case tree?
  - a completely unbalanced tree



Binary Search Trees — Rotations

- We can re-balance unbalanced trees w/ tree rotations



- In-order traversal of all 3 trees is

- so BST order is preserved

Beyond CS16, But good to know