Binary Search Trees

CS16: Introduction to Data Structures & Algorithms

Spring 2018

Outline

- Notion or ordered retrieval
- Binary Search Trees
  - Definitions
  - CRUD operations - insert, search (for specific key), delete
  - Problems and a brief overview of the solutions to fixing them (Red-Black Trees and AVL Trees)

Ordered Retrieval

- Suppose that our data type of our keys have an ordering defined on them
- Want to retrieve keys and/or values sorted by keys
- Want to retrieve smallest or largest key (and maybe associated value if any)
- Can't do this using Hashtables
  - Why?

Naive Implementation

- Store in array!
- Use binary search to find data and to find place to insert order to retain ordering
- Deletion: search for key, remove data, move rest of array up
- Problems?
  - Search: `O(logn)` :-)
  - Insert: `O(n)` :-(
  - Delete: `O(n)` :-(

Ordered Retrieval

- Important operation!
- Backbone of most RDMS!
- Useful when we temporal data
  - Can find out information such as "What even occurred on or before 25th September 2019?"
- Need efficient data structure and algorithm
- Solution: Binary Search Trees! (BSTs)

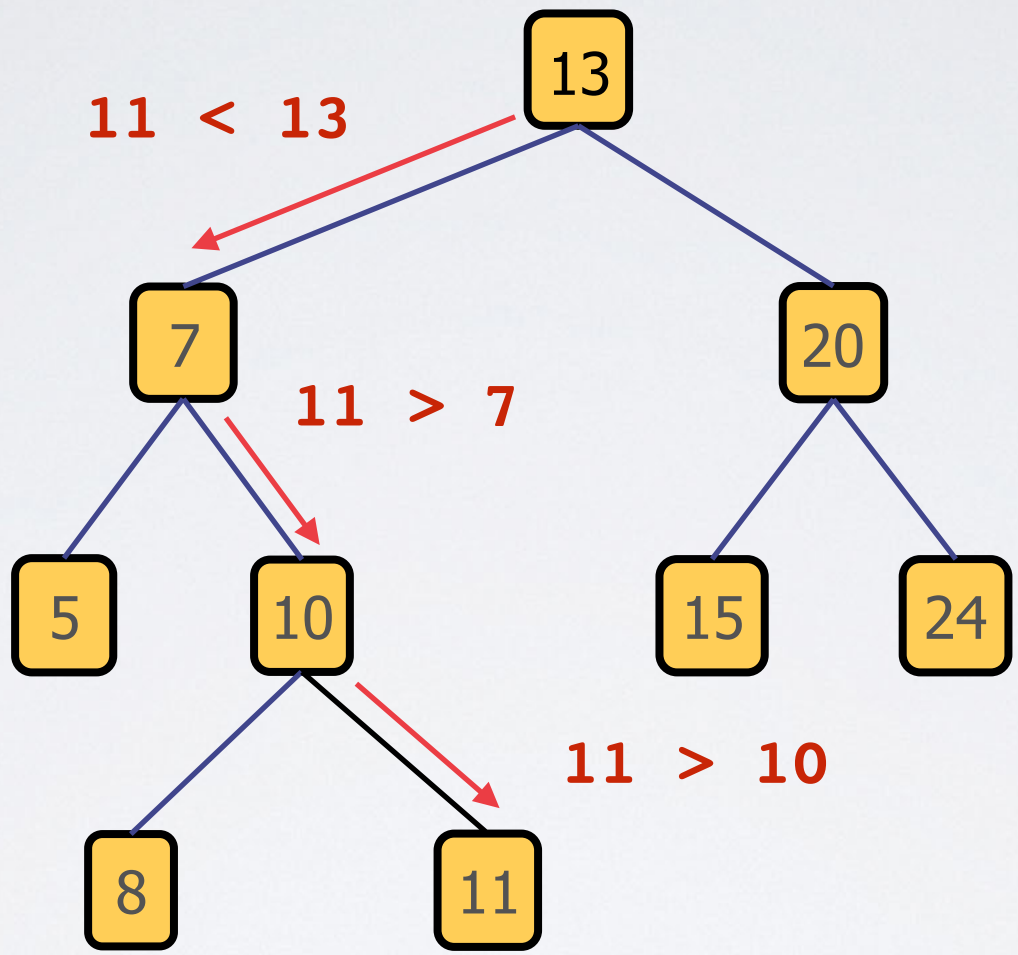
Binary Search Trees

- Special type of Binary Tree
- Each node has left field, right field, parent field and data field as before, but:
  - data field has two subfields: key, and (optionally) value
  - invariant on left field and right field:
    - All keys in left subtree are less then key in data
    - All keys in right subtree are greater than key in data
    - Corollary: in-order traversal of BST can be used to print keys in ascending order!
      - Can prove this formally!

Binary Search Trees

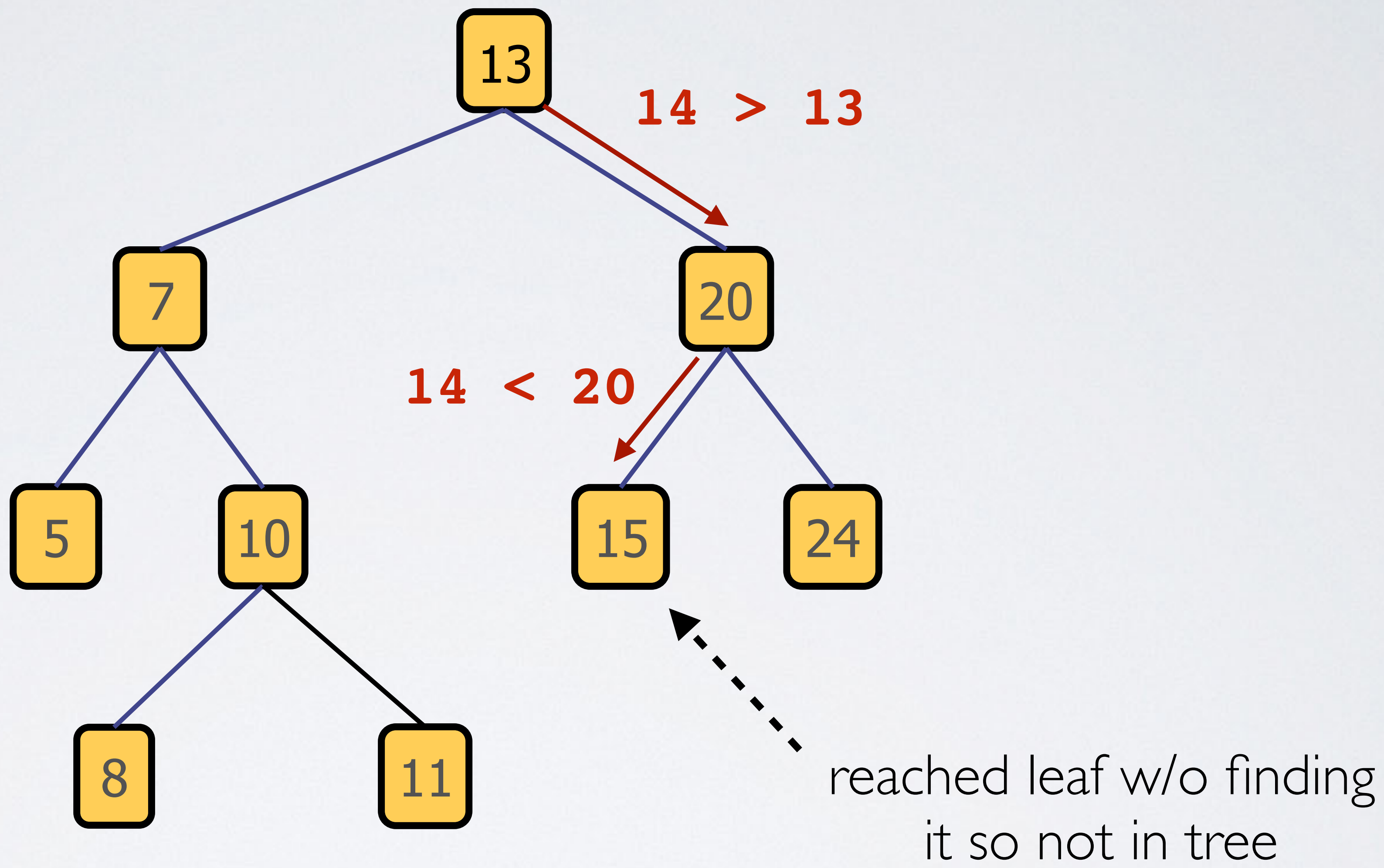
- We need to think recursively!
- Useful tip: typically need to think about 5 cases:
  - When a tree is empty
  - When a node has no children
  - When a node has only a left child
  - When a node has only a right child
  - When a node has both a left and right child
  - Often the action we take is common throughout many cases

Searching a BST



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

Searching a BST

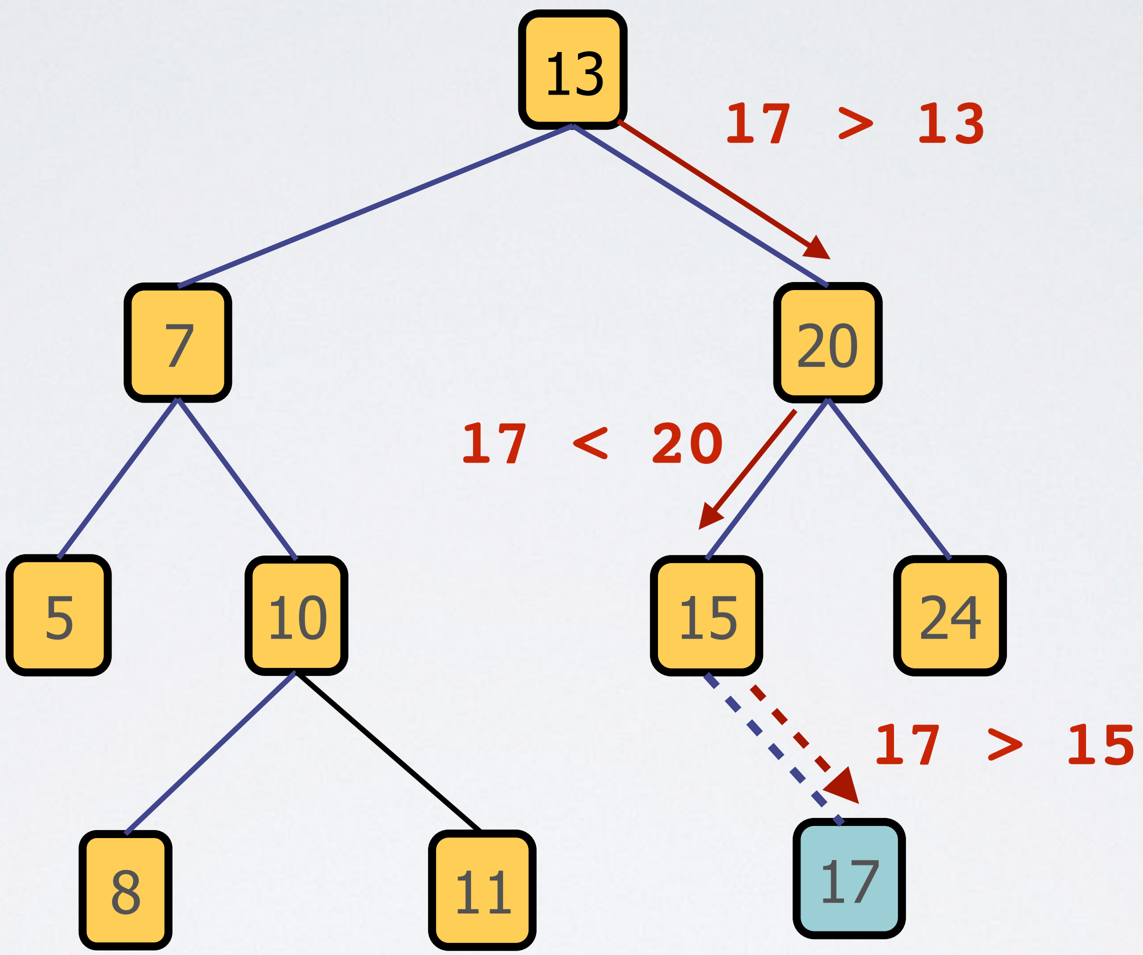


- What if item isn't in tree?
- Find `14`

Searching a BST

```
function search(root, key_to_find):
    if root is NIL:
        return NIL
    if root.data.key == key_to_find:
        return root
    if root.data.key < key_to_find:
        return search(root.left, key_to_find)
    return search(root.right, key_to_find)
```

Inserting in a BST



- To insert, perform a search and add as new leaf
- Insert `17`

Inserting into a BST

```
function insert(root, key, value=NIL):
   if root.data.key == key:
      return
   if root.data.key < key:
      if root.left is NIL:
        data = create_data(key, value)
        root.left = new_node(data, root)
      else:
        insert(root.left, key, value)
   if root.data.key < key:
      if root.right is NIL:
        data = create_data(key, value)
        root.right = new_node(data, root)
      else:
        insert(root.right, key, value)
```

Smallest Node

- Smallest node is the node with the smallest key in the BST
- How to find? Remember the cases we need to consider
  - Case #1: Tree is empty
  - Case #2: Root has no children (root is only node)
  - Case #3: Root has only a left child
  - Case #4: Root has only a right child
  - Case #5: Root has both left and right child

Smallest Node

- Smallest node is the node with the smallest key in the BST
- How to find? Remember the cases we need to consider
  - Case #1: Tree has no smallest node
  - Case #2: Root is smallest node
  - Case #3: Smallest node is in left child
  - Case #4: Root is smallest node
  - Case #5: Smallest node in left child

Smallest Node

- Lefts combine cases!
- Case#1 stands alone
- Case#2 and Case#4 are the same (what is common between them?)
- Case #3 and Case#5 are the same (what is common between them?)

Smallest Node

- Lefts combine cases!
- Case#1 stands alone
- Case#2 and Case#4 are the same (no left child)
- Case #3 and Case#5 are the same (has a left child)
- Can write algorithm from this!

Smallest Node in BST

```
function smallest(root):
    if root is None:
        return NIL
    if root.left is NIL:
        return root
    return smallest(root.left)
```

Should try solving for largest node!

Bounded Range Retrieval

- We have an upper bound x and a lower bound y
- Want to retrieve all nodes in a BST with keys greater than or equal to x and less than or equal to y
- Want to retrieve in order of key
  - First, we find root of subtree that is within range (how would we do this?)
  - Then use a modification of in-order traversal!

Bounded Range Retrieval

- Case #1: If the root is empty, nothing to retrieve
- Case #2:
  - 2a: if the root has no children and is in the range, add it to retrieved items
  - 2b: if the root has no children and is in not range, terminate
- Case #3:
  - 3a: if the root has a left children and is in range, add it to retrieved items and go down left child
  - 3b: terminate

Bounded Range Retrieval

- Case #4:
  - 4a: if the root has a right children and is in range, add it to retrieved items and go down right child
  - 4b: terminate Case #2:
- Case #5:
  - 5a: if the root has both children and is in range, add it to retrieved items and go down both children
  - 5b: terminate

Range Retrieval

- Core idea:
  - Once subtree is found
  - If root is in range, add to items
  - Go down children and apply recursively
- Find subtree using modification of search

Range Retrival

```
function range_ret_helper(root, hi, lo):
   if root is None or root.data.key < lo
      or root.data.key > hi:
      return []

   res = [root]
   res_from_left = range_ret_helper(root.left, hi, lo)
   res_from_right = range_ret_helper(root.right, hi, lo)
   res = concat(res, res_from_right)
   res = concat(res_from_left, res)
   return res
```

Range Retrieval

```
function range_ret(root, hi, lo):
   if root is None:
      return []
   if root.data.key >= lo and root.data.key <= hi:
      return range_ret_helper(root, hi, lo)
   if root.data.key < lo:
      return range_ret_helper(root.right, hi, lo)
   return range_ret_helper(root.left, hi, lo)
```

Successors and Predecessors

- In-order successor of a node **`x`** - the node that comes immediately after **`x`** in an in-order retrieval
- In-order predecessor of a node **`x`** - is the node that comes immediately after **`x`** in an in-order retrieval

Successor

- Core idea:
  - If we have a right child, then the successor is the smallest node in the right child
  - If we have no right child, then the successor is one of the nodes ancestors
    - Parent might be in successor if node is left of parent
    - If node not left of parent, then we need to find closest ancestor who is left of their parent. The parent of that ancestor is the in-order successor

Smallest Node in BST

```
function successor(root):
    if root is None:
        return NIL
    if root.right is not NIL:
        return smallest(root.right)
    curr = root
    p = root.parent
    while p != NIL and p.right == curr:
        curr = p
        p = curr.parent
    return p
```

Predecesor

- Predecessor is similar, but with some conditions changed
- Try writing on your own
- Will cover in labs

LTE queries

- We have some key **`k`**, want to find node with key **`x`** such that **`x`** is the largest key in the tree such that $\mathbf{x} \leq \mathbf{k}$

LTE Queries

- Break down by cases!
  - In all cases, if root exists, then root is answer if root's key is the key we are concerned with
  - Empty tree: no answer
  - No children: root is either answer or not
  - Left child: if root's key is greater than k, then answer must be in left child; else not in this subtree
  - Right child: if root's key is less than k, then answer might be in right child; else answer is root

LTE Query

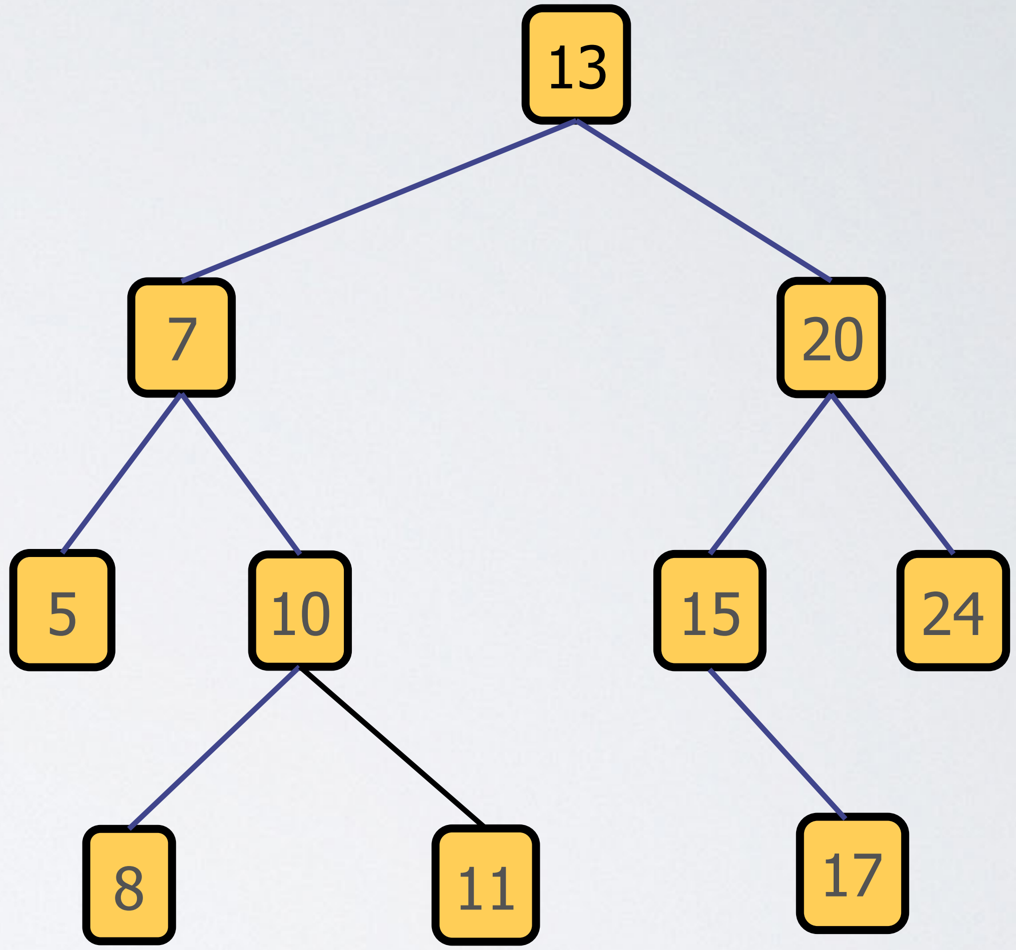
```
function lte_search(root, key):
    if root is None:
        return NIL
    if root.data.key == key:
        return root
    if root.data.key < key:
        res = lte_seaerch(root.right, key)
        return (res is NIL) ? root:res
    return lte_root(root.left, key)
```

GTE Query

- GTE query is similar
- Try writing on own
- Part of the assignment!

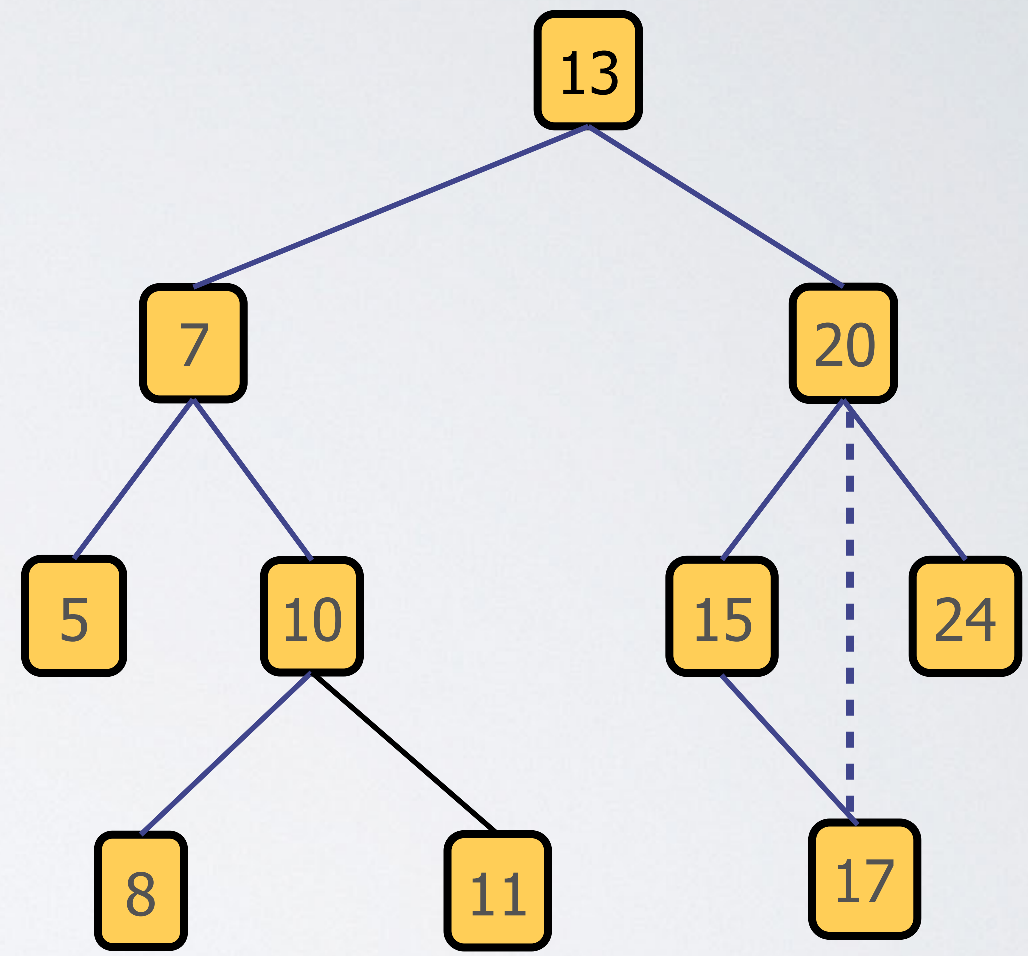
Removing from a BST

- 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`)



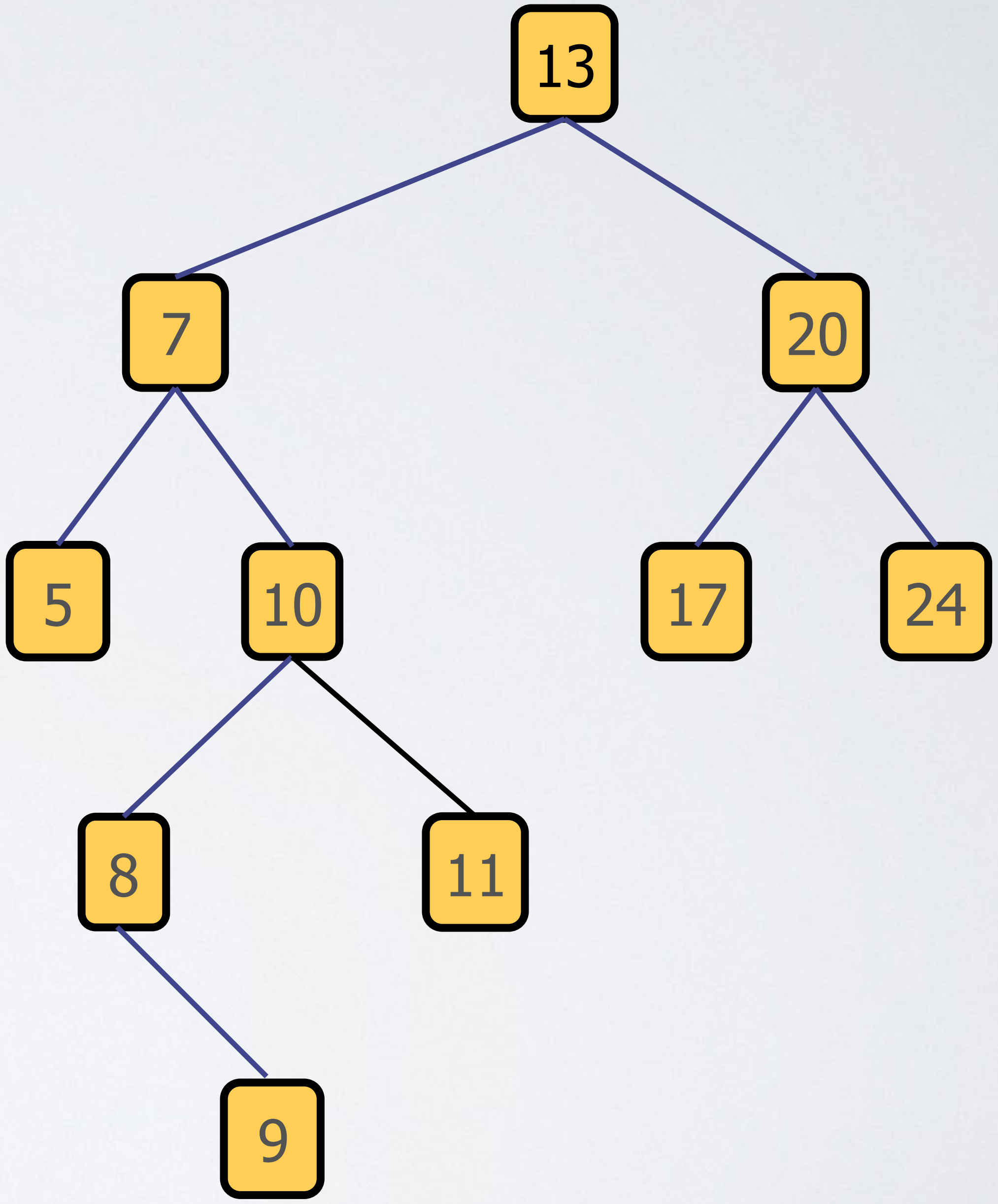
Removing from a BST - Case #2

- Removing internal node w/ `1` child
- Strategy
  - "Splice out" node by connecting its parent to its child
- Example: remove `15`
  - set parent's left pointer to 17
  - remove 15's pointer
  - no more references to 15 so erased (garbage collected)
  - BST order is maintained



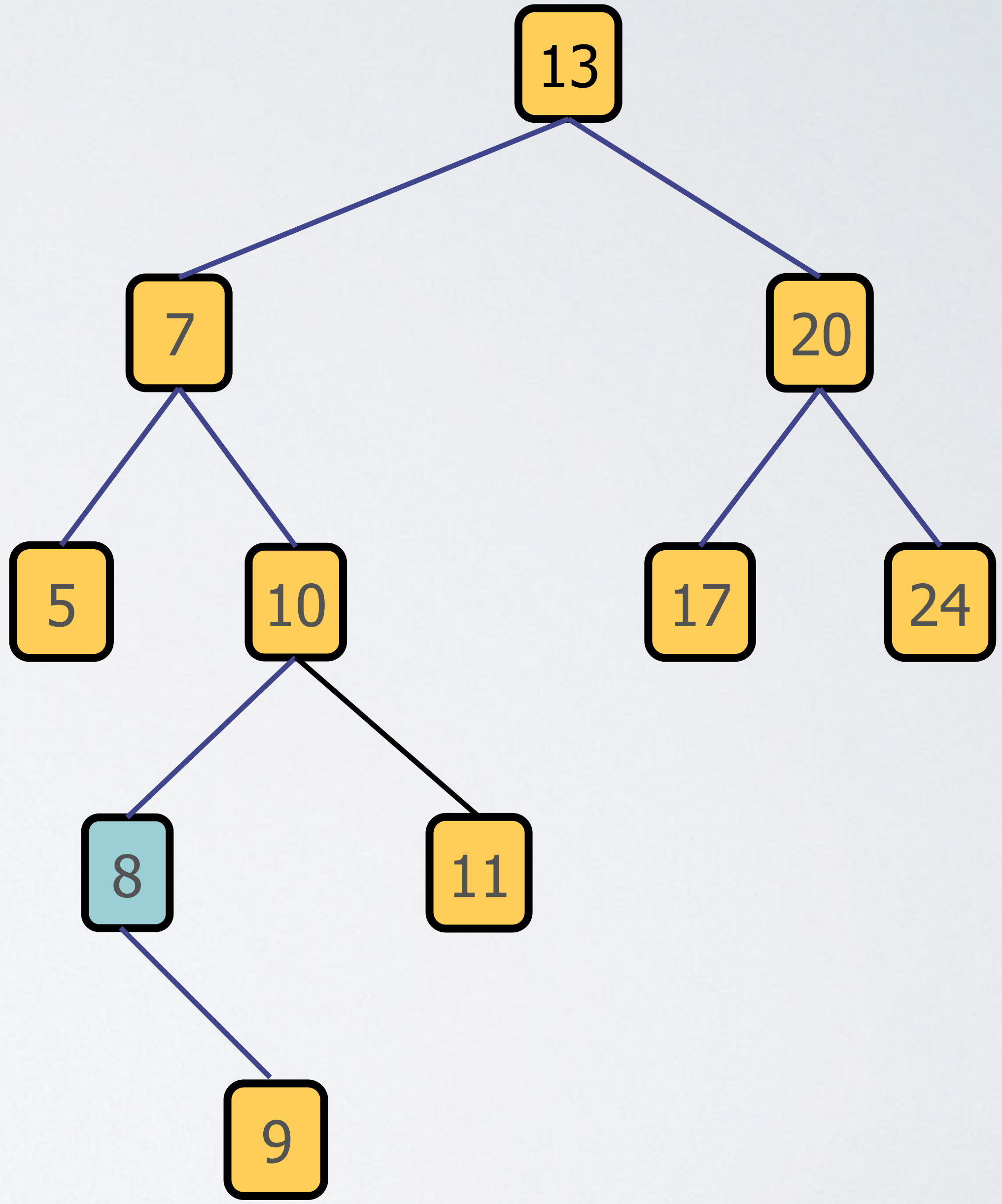
Removing from a BST - Case #3

- 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**?



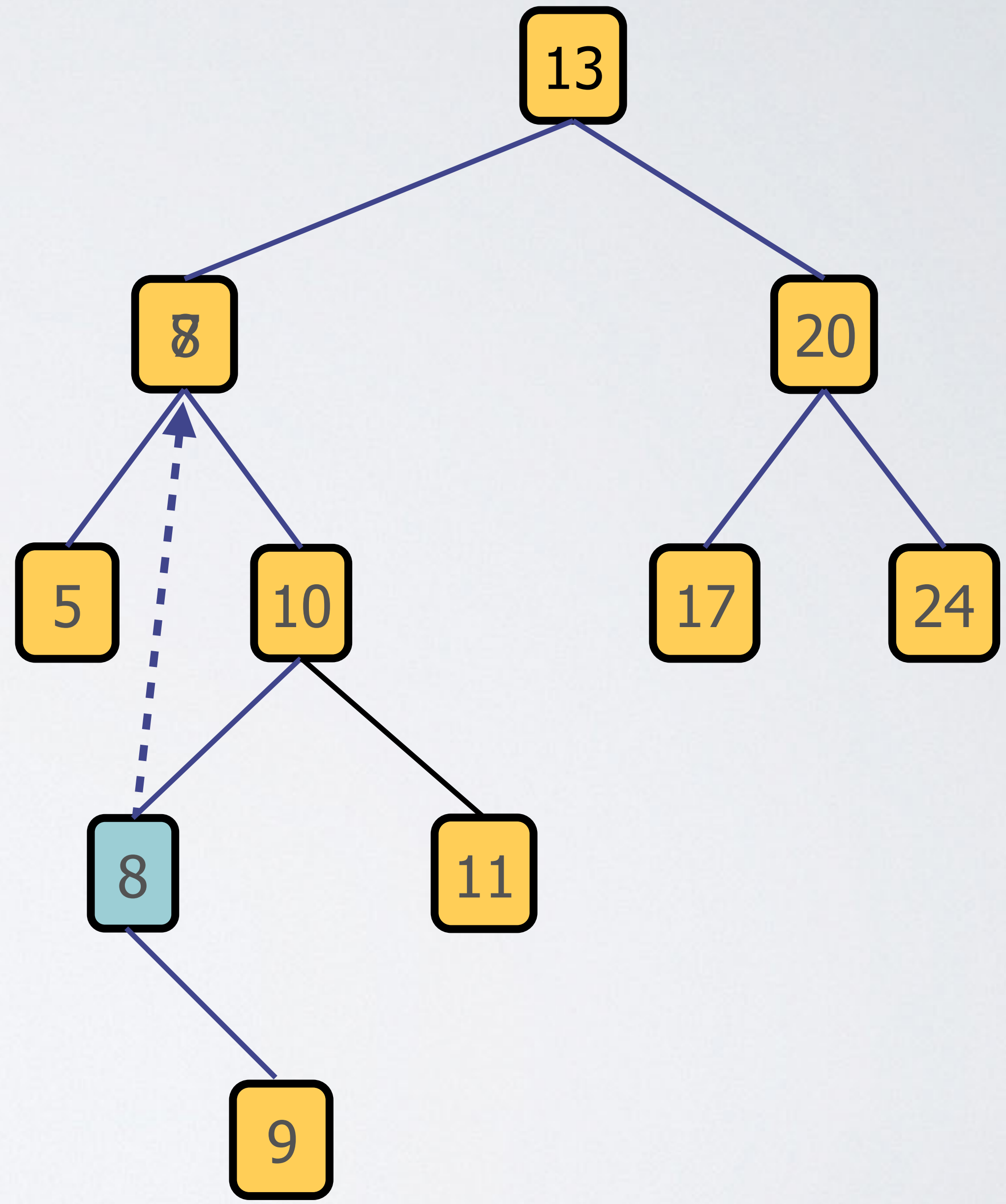
Removing from a BST - Case #3

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



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

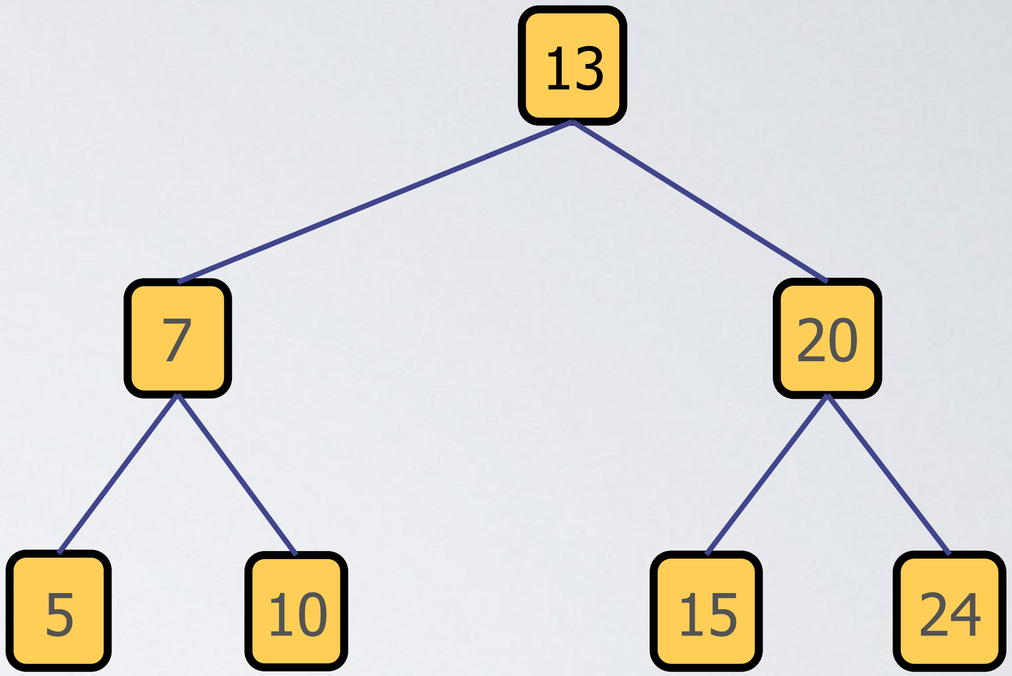


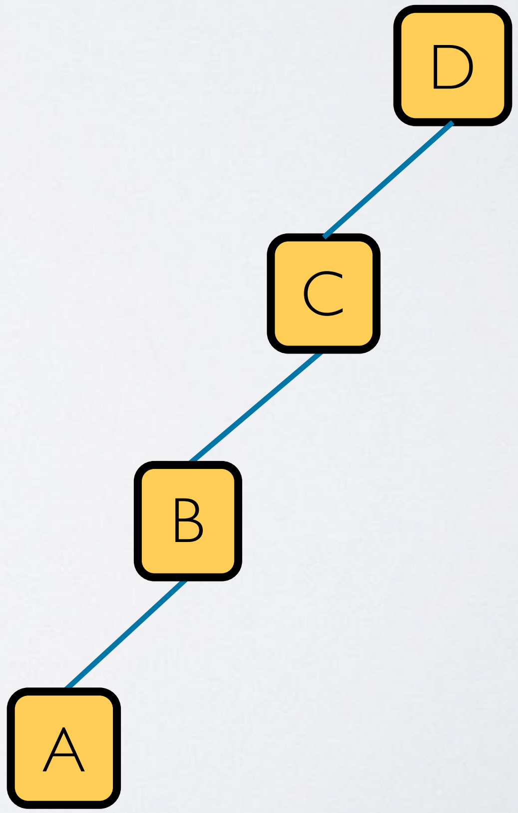
Deletion

- Try writing pseudocode as an exercise
- Will cover solution in lab

Binary Search Tree Analysis

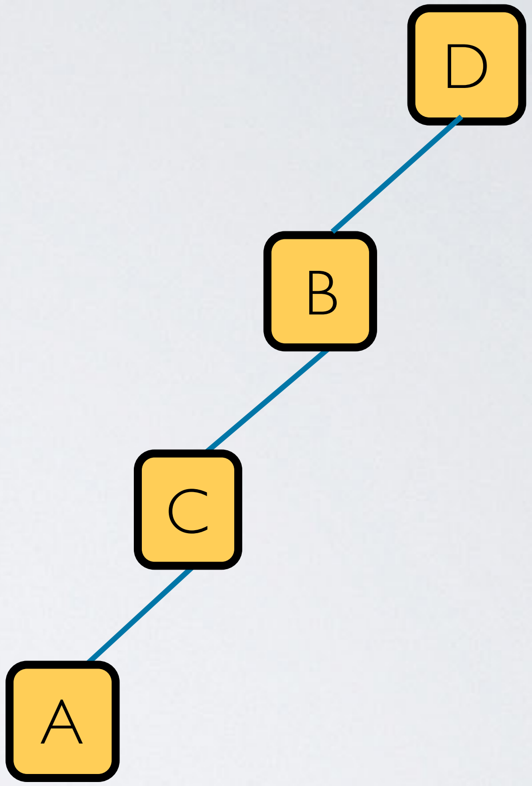
- How fast are BST operations?
  - Given a tree, what is the worst-case node to find/remove?
- What is the best-case tree?
  - a balanced tree
- What is the worst-case tree?
  - a completely unbalanced tree





Degenerate Cases

- What if we insert sorted data (either ascending or descending)into BST?
- Tree looks like linked list
- Performance expectations breakdown!

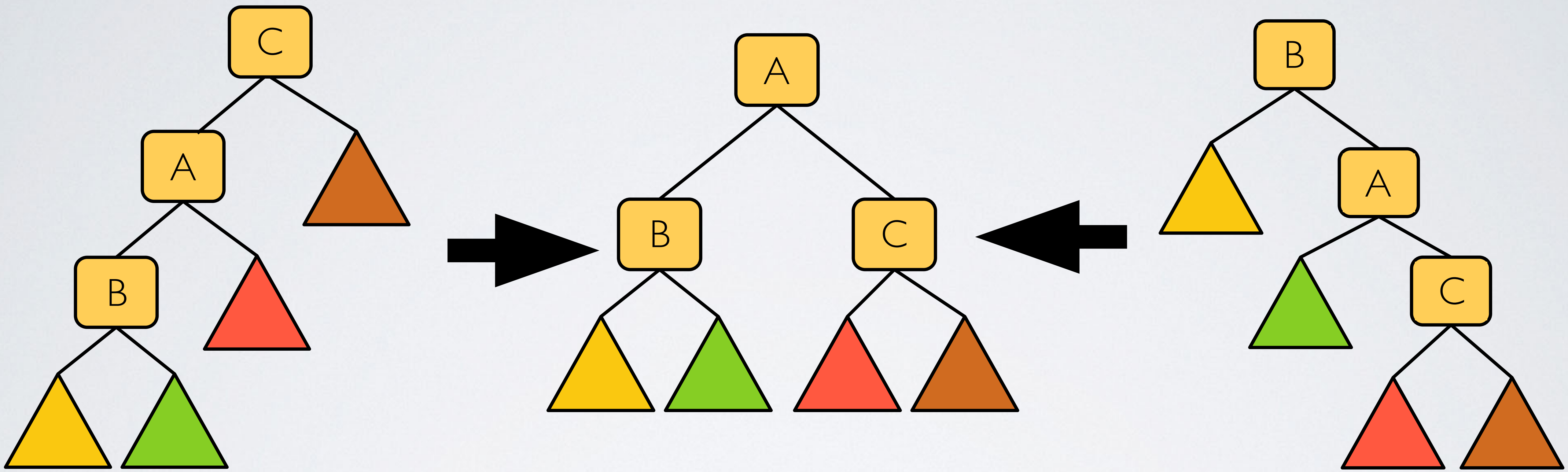


How to solve?

- We modify our method of inserting data
- Use rotations to keep tree balanced or mostly balanced
- Can help guarantee `O(logn)` performance
- Techniques: AVL trees, Red-black trees

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