

# CSC148 – Introduction to Computer Science

Lecture 7: Binary Search Trees Part 2

Sean Henderson

# Last time...

- Defined what a Binary Search Tree is
- Implemented searching and inserting into such an object
- Worst case height  $O(n)$
- Best case height  $O(\log n)$
- Theorem: “average” case height is  $O(\log n)$ 
  - When inserting  $n$  elements into the tree in random order

# Removal

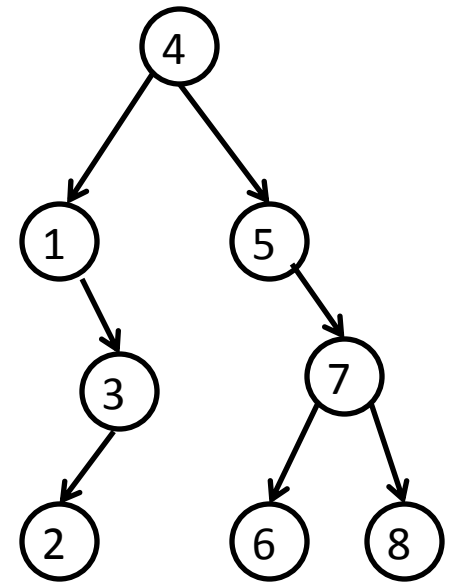
- Problem: given a value we want to remove from the tree, remove it
- Only want to remove it if it exists
- What are some easy cases?

# Lots of Cases

- Easy cases:
  - Leaf nodes
  - Nodes with 1 child
  - Values not in the tree
- Hard cases:
  - Nodes with 2 children
  - The root

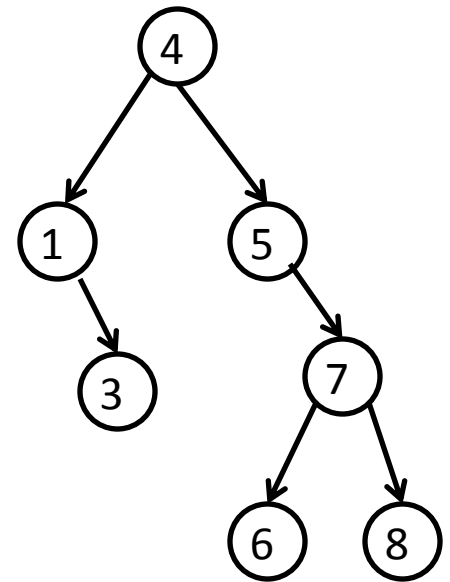
# Easy Case #1

- Removing a leaf node; let's say 2
- All we really need to do is set 3's left pointer to be None



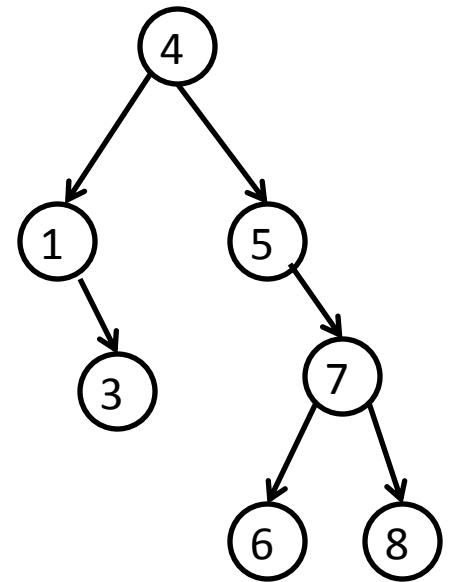
# Easy Case #1

- Removing a leaf node; let's say 2
- All we really need to do is set 3's left pointer to be None



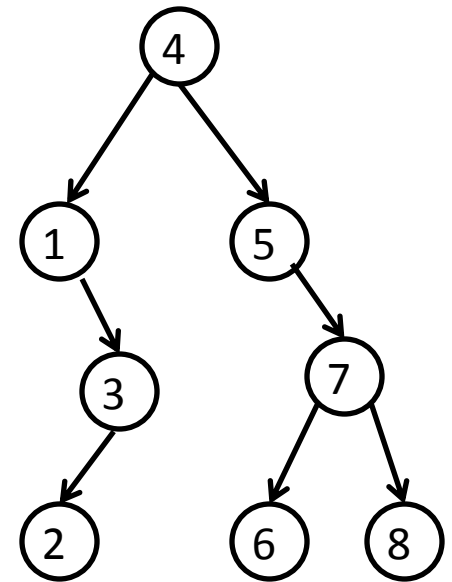
# Easy Case #1

- Removing a leaf node; let's say 2
- All we really need to do is set 3's left pointer to be None
- Steps Involved:
  - Find the parent of 2
  - Update child pointers



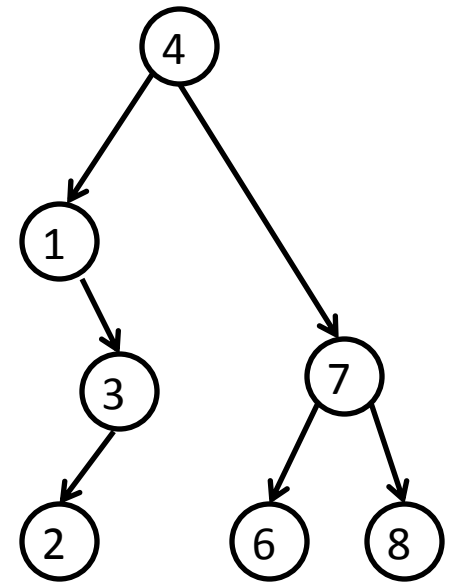
# Easy Case #2

- Removing a node with one child; let's say 5
- We can set 4s right child to be 7



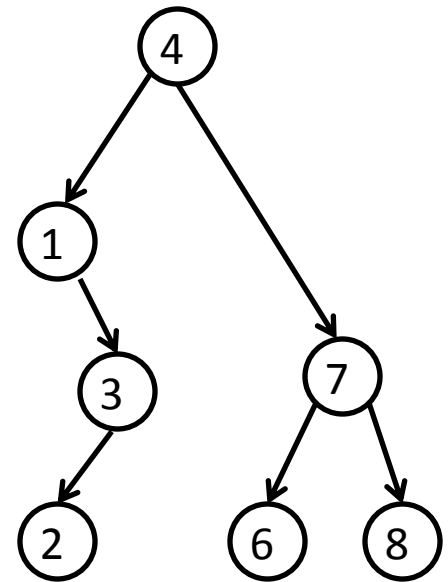
# Easy Case #2

- Removing a node with one child; let's say 5
- We can set 4s right child to be 7



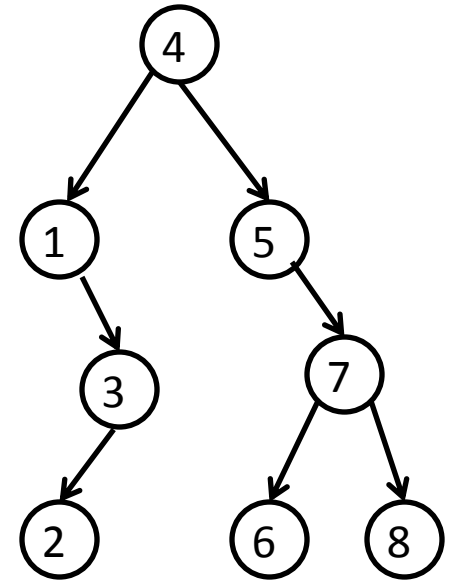
# Easy Case #2

- Removing a node with one child; let's say 5
- We can set 4s right child to be 7
- Steps involved:
  - Find the parent of 5
  - Update child pointer of parent to be the sole child



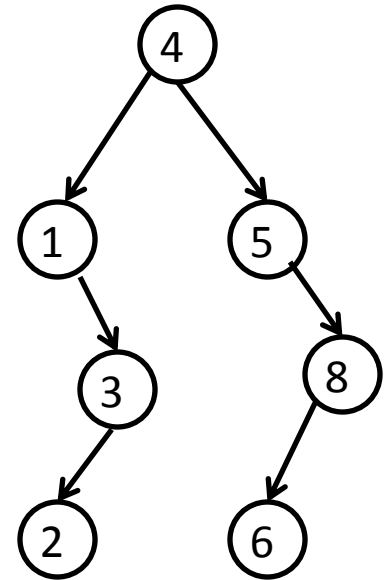
# Hard Case #1

- We want to remove a node with 2 children
- Let's say 7
- Idea: replace 7 with the successor of 7
  - Successor: next largest element
  - What about predecessor?
- Replace 7 with 8



# Hard Case #1

- We want to remove a node with 2 children
- Let's say 7
- Idea: replace 7 with the successor of 7
  - Successor: next largest element
- Replace 7 with 8



# Hard Case #1

- First, have to find the node to remove
- Then, need to find the successor and remove it from the tree
  - The successor will be the left-most node in the right subtree (why?)
  - Successor will have at most one child, making its removal “easy” (why?)
- Finally, replace the node to remove with the successor

# Hard Case #2

- Removing the root node
- Basically, the three cases already discussed, with the caveat that we don't need to update the parent of the node we removed; instead need to update self.root

# Implementation

- We'll write a `remove(self, key)` method
- Also, something to extract the successor of a node with two children
- Iterative, not recursive
  - Try thinking how to write it recursively

# Runtime

- $O(\text{height of tree})$ 
  - Only go from root to node we want to delete, and then potentially its successor
- Worst case runtime still linear if tree was poorly constructed
- If we were keeping track of a size attribute on each node, how would we update those after removal in each of the cases?

# Summary

- Insertion, removal, search all  $O(\text{height of tree})$
- Tree can be very unbalanced (height nowhere near  $\log n$ ) in the worst case
- When inserting items randomly, a tree with  $n$  nodes has height  $O(\log n)$  with high probability
- There exist (relatively) simple ways of augmenting a BST to guarantee  $O(\log n)$  height after any sequence of operations