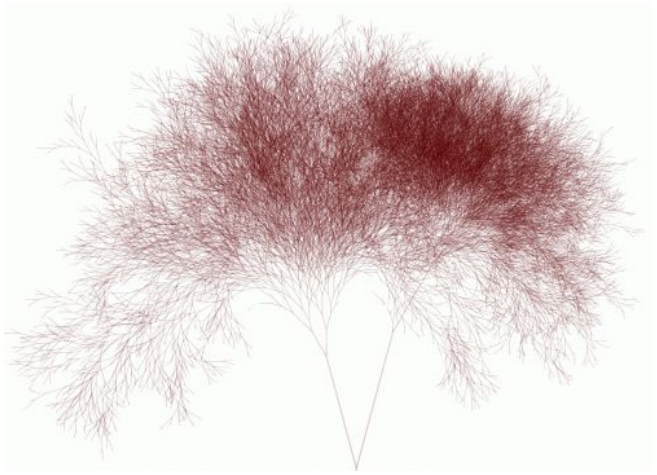
ICS-211 Lab

Bifurcating Arborescences

(a.k.a. Binary Trees)



Announcements

- Friday is a holiday (no lab)
- Assignment 8 due on Nov. 18
- Last opportunity for extra credit attached to A08 (15 points)
- Section 1 - you're welcome (and encouraged) to attend the next lab
    - I'll be covering Assignment 8 in more detail
    - You can leave whenever you want
    - I'll also cover this next Wednesday

Algorithm Analysis Write-up

- 25% of assignment
- Simply stating a tree sort is O(n log n) is not sufficient!
  - Must explain why
  - Must be supported by empirical data
    - I.e., you'll need to implement sorting using a b-tree
    - This is done with an in-order traversal (see book or Wikipedia)
- Should include:
  - A theoretical analysis of your code - why your code is O(n log n)
  - Best/worst-case performance and when/why those occur
  - Comparison of performance
    - Compare a slow sort you've already implemented to a tree sort
    - E.g., How long does it take to sort 1000, 10,000, 100,000, etc. elements for a bubble sort vs. tree sort
- Write-up should be in your README file(PDF or text files only)

Overview

- There are never duplicate entries in the b-tree (as per definition of interface)
- Implement a Contact and `ContactComparator` class
  - You should already have most of this
  - Just need to update comparator to compare last name, first name
- Test code (https://github.com/psoulier/ics211-fall16)
  - Basic JUnit test cases for assignment and extra credit
  - These tests are just a starting point (i.e., not complete)
    - Lots of corner cases
    - No tests for contact list
  - Useful method to generate data to populate balanced trees
- The add method needs to replace existing item
  - Think about updating the address of a contact.
  - The contact name (i.e., key) is the same, but the address changes
- The remove method is probably more challenging
  - Three cases: leaf, partial, and full nodes
  - Generally need to “remember” parent of node being removed for leaf and partial nodes
  - A full node can just have its content replaced (don’t need to remove then re-link)

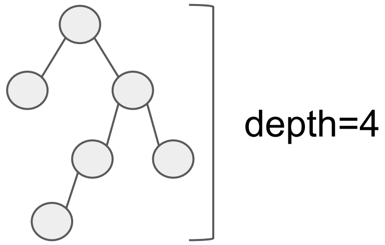
Extra Credit - A08 (15 points)

Overview

- See "SearchTree.java" in my GitHub repo for interface
- Both methods lend themselves to recursion

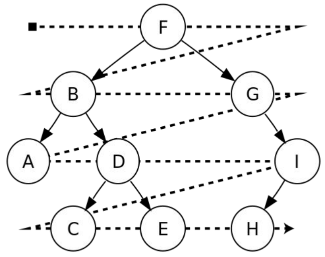
Tree Depth (7 pts)

- Implement `int maxDepth()` method
- An empty tree has a depth of 0

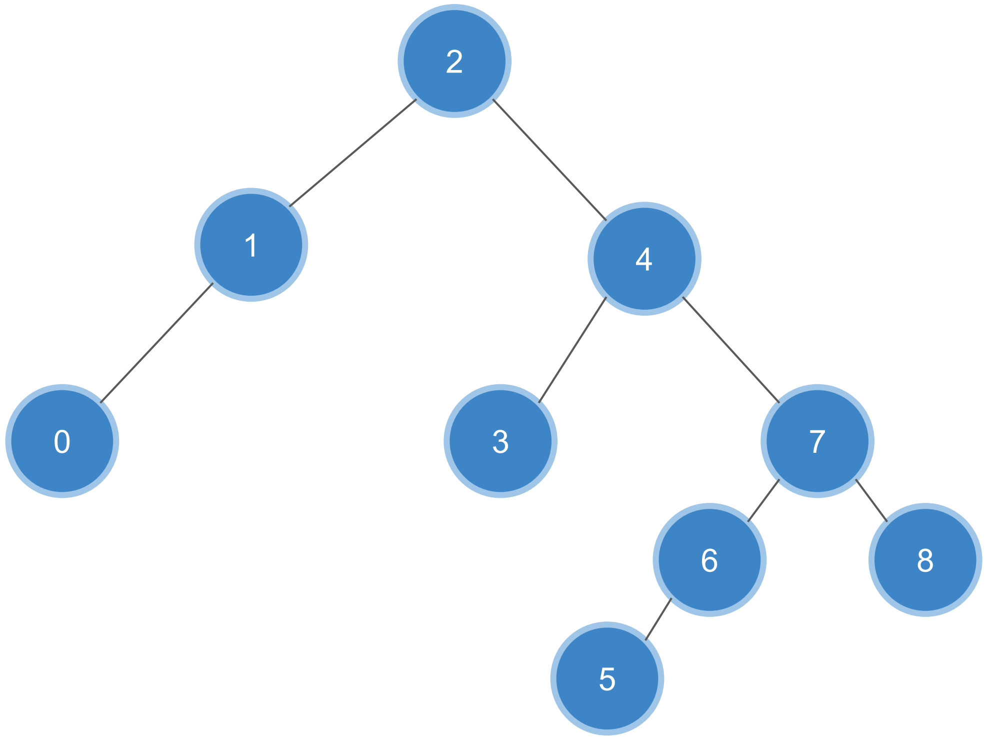


Level-Order Traversal (8 pts)

- Implement `List<E> levelOrderTraversal()` method
- Returns a list containing the elements of the tree in "level order"
- Below image would return a list containing [F, B, G, A, D, I, C, E, H] where "F" is at position 0 and "H" at position 8 (image taken from Wikipedia)



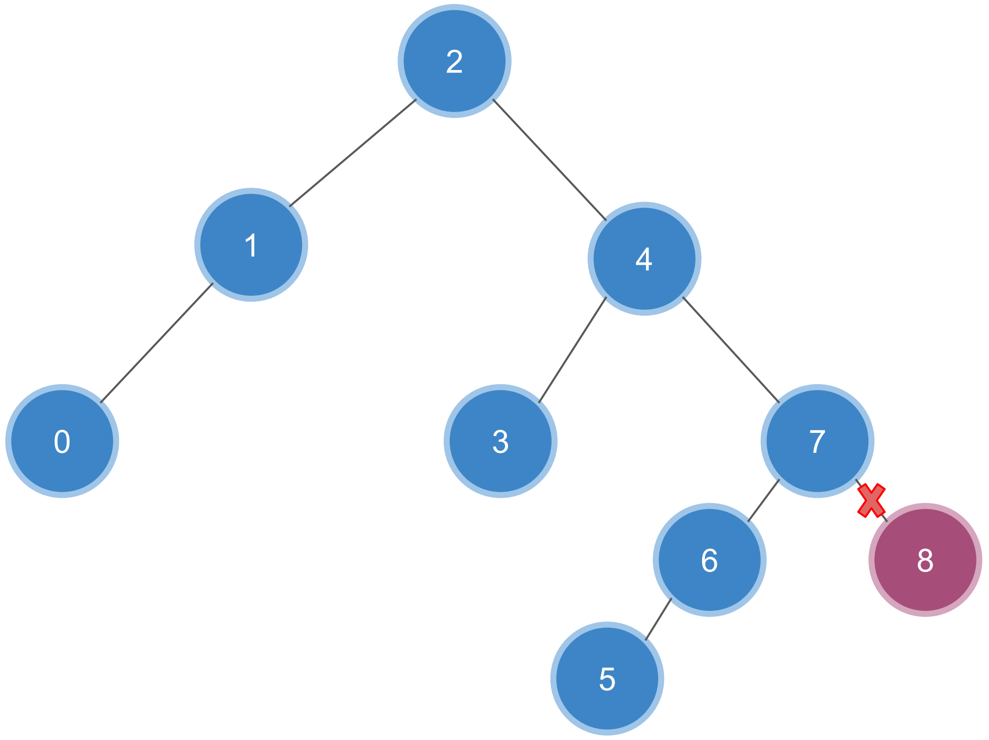
B-Tree `remove` Example (Leaf Node)



- Remove “8”

In-order list: `0, 1, 2, 3, 4, 5, 6, 7, 8`

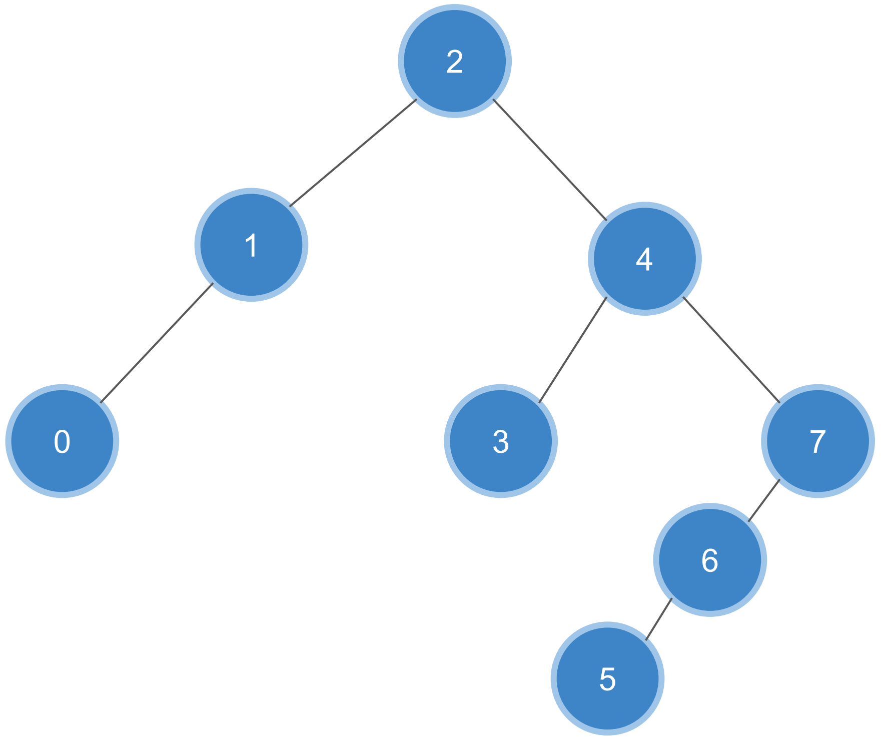
B-Tree `remove` Example (Leaf Node)



- Remove “8”
- It’s a leaf node
- Just need to set the “right” link in parent to `null`

In-order list: `0, 1, 2, 3, 4, 5, 6, 7, 8`

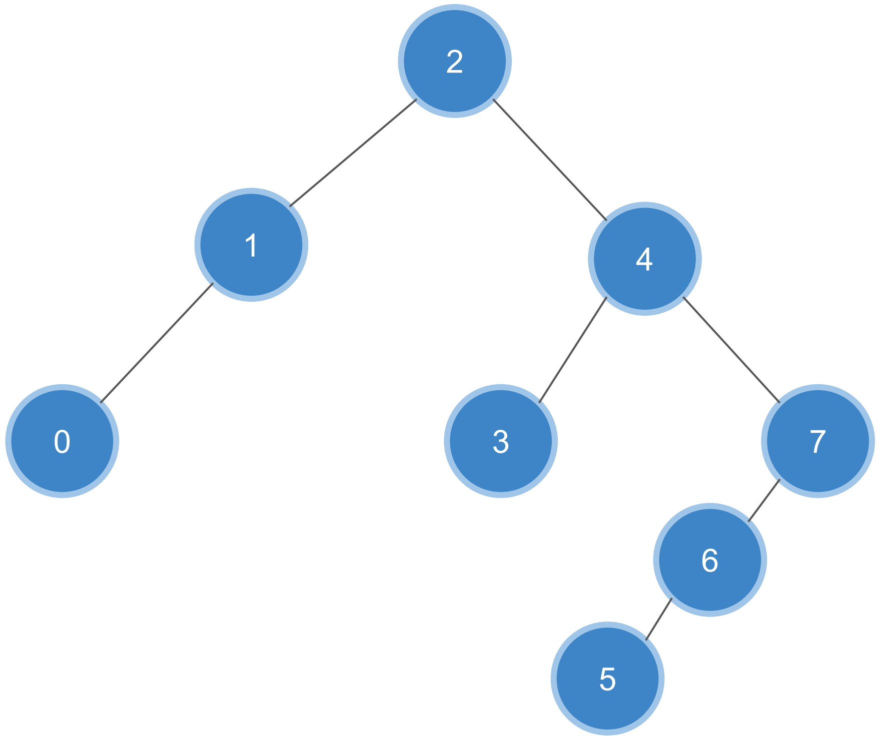
B-Tree `remove` Example (Leaf Node)



- Remove “8”
- It’s a leaf node
- Just need to set the “right” link in parent to `null`

In-order list: `0, 1, 2, 3, 4, 5, 6, 7`

B-Tree `remove` Example (Partial Node)

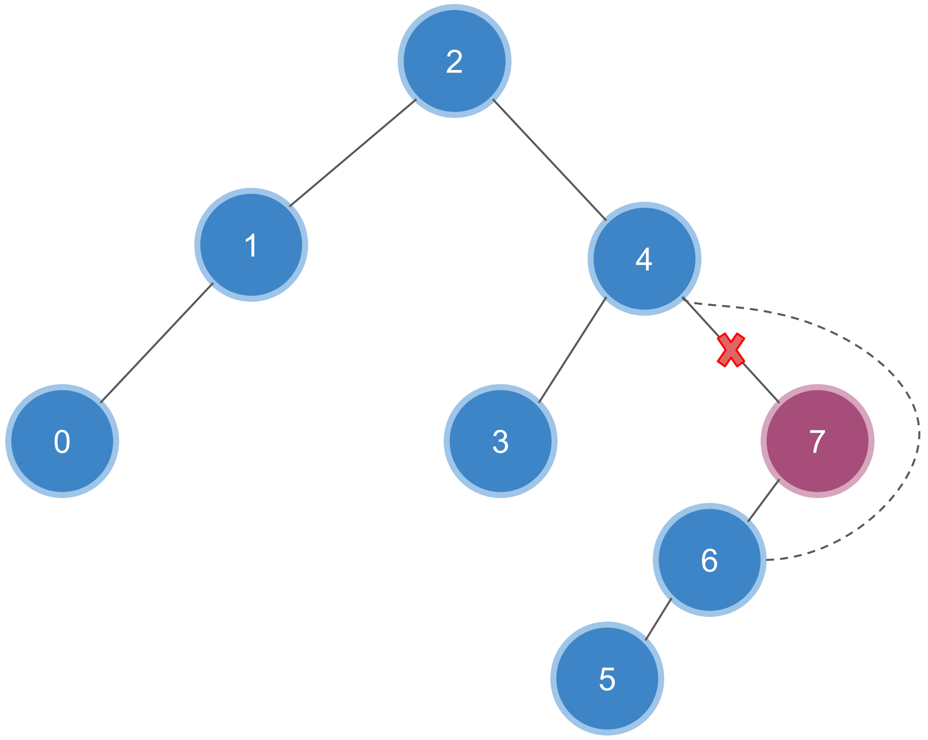


- Remove “8”
- It’s a leaf node
- Just need to set the “right” link in parent to `null`

- Remove “7”

In-order list: `0, 1, 2, 3, 4, 5, 6, 7`

B-Tree `remove` Example (Partial Node)

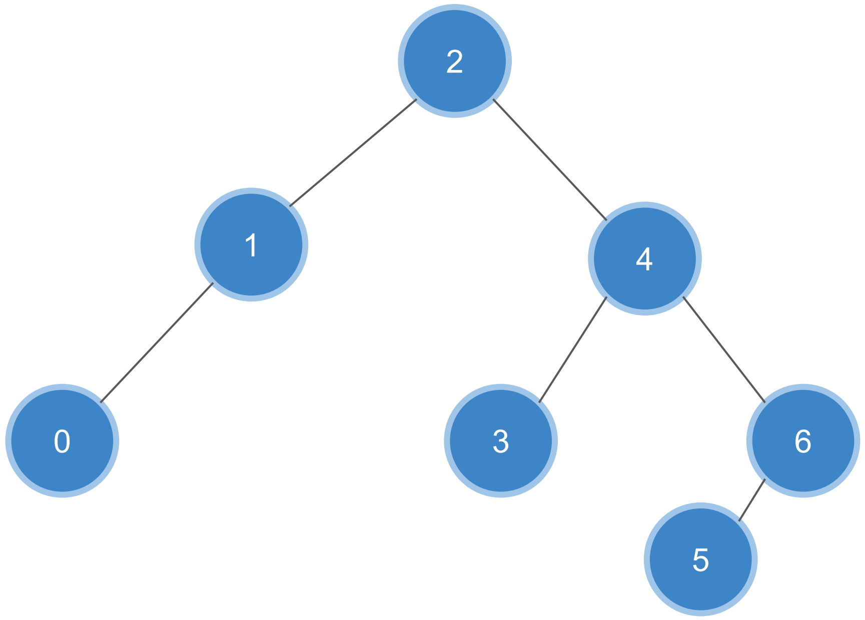


- Remove “8”
- It’s a leaf node
- Just need to set the “right” link in parent to `null`

- Remove “7”
- It’s a partial node
- Reassign parent “right” node to child of node being removed

In-order list: `0, 1, 2, 3, 4, 5, 6, 7`

B-Tree `remove` Example (Partial Node)

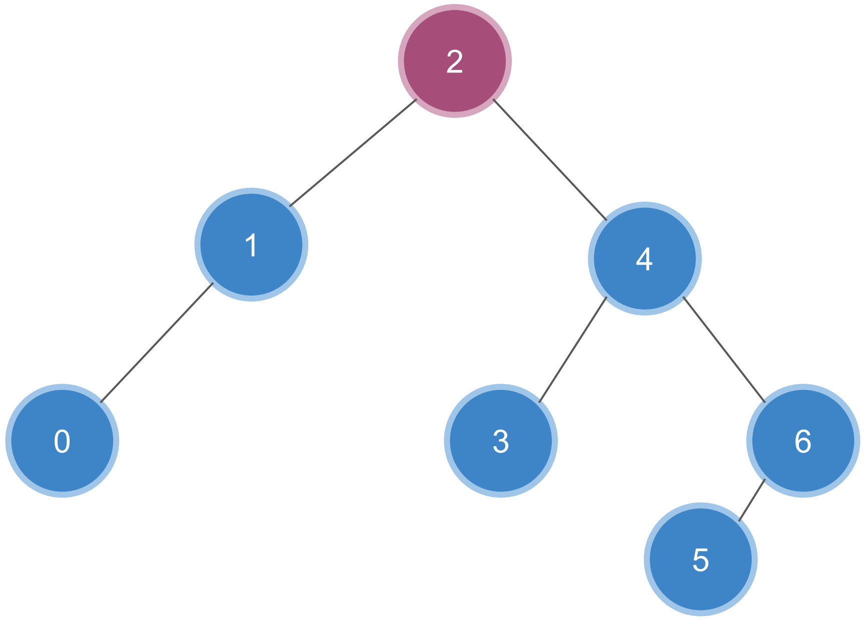


- Remove “8”
- It’s a leaf node
- Just need to set the “right” link in parent to `null`

- Remove “7”
- It’s a partial node
- Reassign parent “right” node to child of node being removed

In-order list: `0, 1, 2, 3, 4, 5, 6`

B-Tree `remove` Example (Full Node)

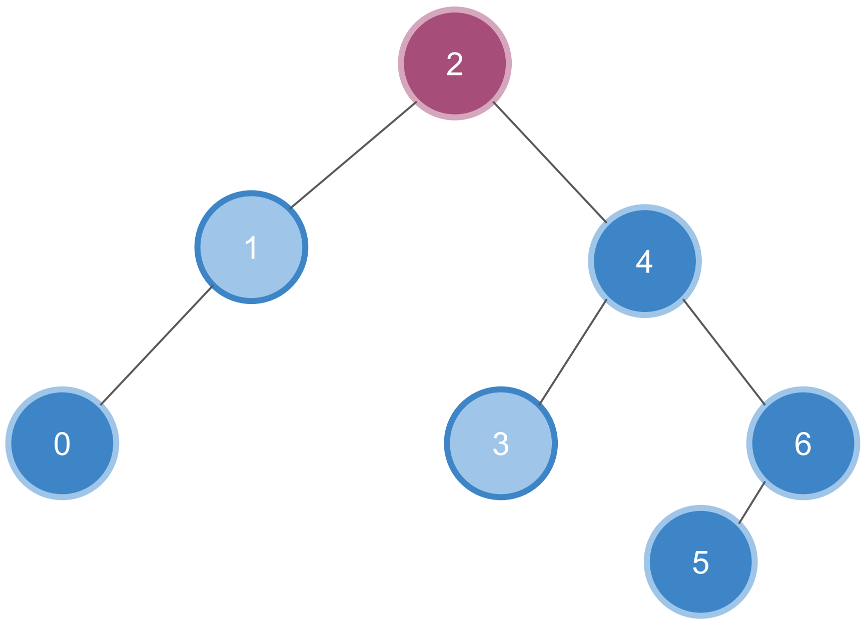


- Remove “7”
- It’s a partial node
- Reassign parent “right” node to child of node being removed

- Remove “2”

In-order list: `0, 1, 2, 3, 4, 5, 6`

B-Tree `remove` Example (Full Node)

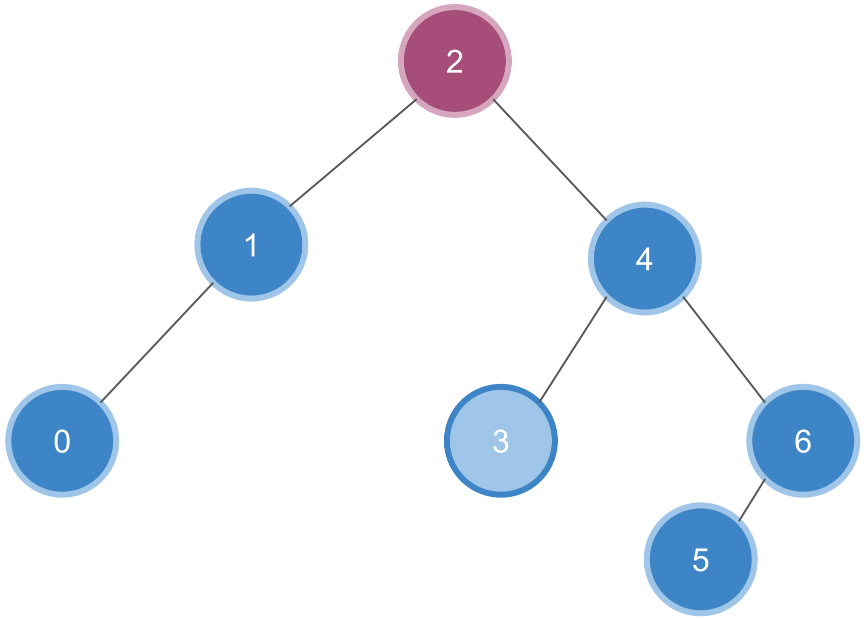


- Remove “7”
- It’s a partial node
- Reassign parent “right” node to child of node being removed

- Remove “2”
- It’s a full node
- Replace with in-order predecessor (“1”) ***<u>or</u>*** successor (“3”)
- Predecessor is left, then (while ***not*** null); right, right, right...
- Successor or right, then (while ***not*** null); left, left, left…
- Note that the predecessor/successor will be a leaf or partial node (never a full node)

In-order list: `0, 1, 2, 3, 4, 5, 6`

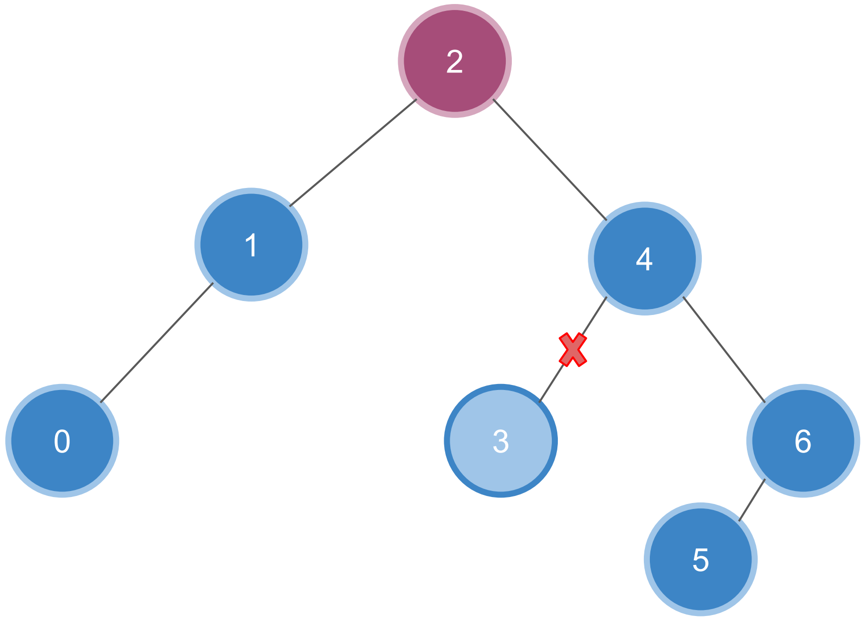
B-Tree `remove` Example (Full Node)



- Remove “2”
- It’s a full node
- Replace with in-order predecessor (“1”) ***or*** successor (“3”)
- Predecessor is left, then (while ***not*** null); right, right, right...
- Successor or right, then (while ***not*** null); left, left, left…
- Note that the predecessor/successor will be a leaf or partial node (never a full node)
- Assume implementation uses successor (node “3”)

In-order list: `0, 1, 2, 3, 4, 5, 6`

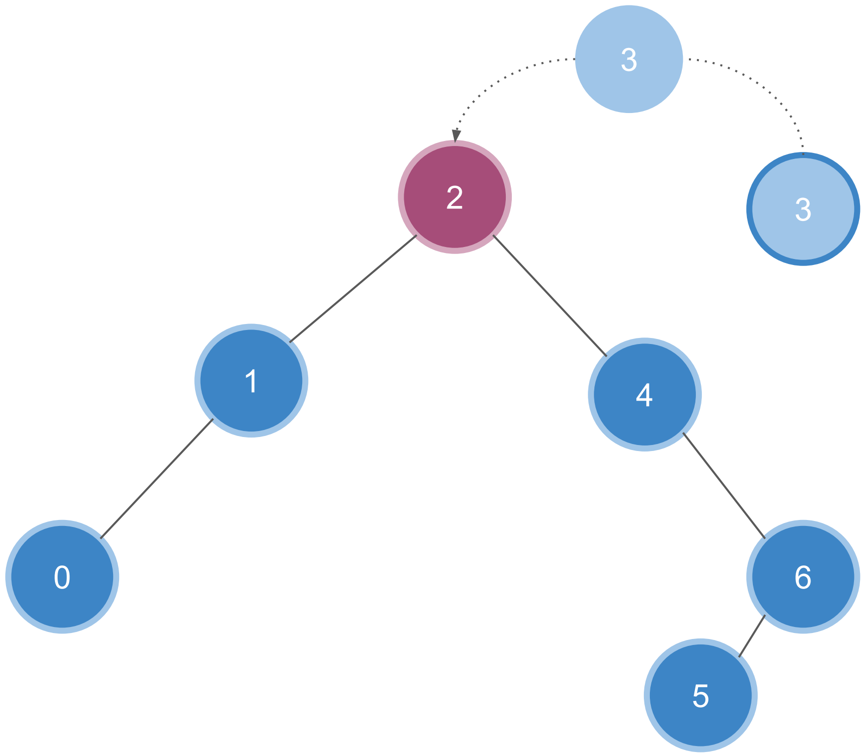
B-Tree `remove` Example (Full Node)



In-order list: `0, 1, 2, 3, 4, 5, 6`

- Remove “2”
- It’s a full node
- Replace with in-order predecessor (“1”) ***or*** successor (“3”)
- Predecessor is left, then (while ***not*** null); right, right, right...
- Successor or right, then (while ***not*** null); left, left, left…
- Note that the predecessor/successor will be a leaf or partial node (never a full node)
- Assume implementation uses successor (node “3”)
- Remove “3”

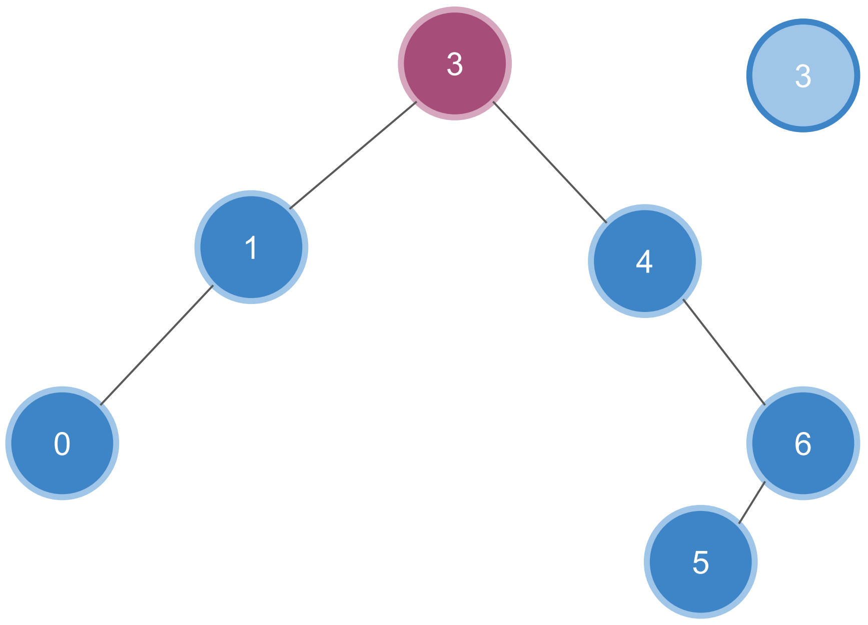
B-Tree `remove` Example (Full Node)



- Remove "2"
- It's a full node
- Replace with in-order predecessor ("1") ***or*** successor ("3")
- Predecessor is left, then (while ***not*** null); right, right, right...
- Successor or right, then (while ***not*** null); left, left, left…
- Note that the predecessor/successor will be a leaf or partial node (never a full node)
- Assume implementation uses successor (node "3")
- Remove "3"
- Replace content of node (don't need to replace node itself)

In-order list: `0, 1, 3, 4, 5, 6`

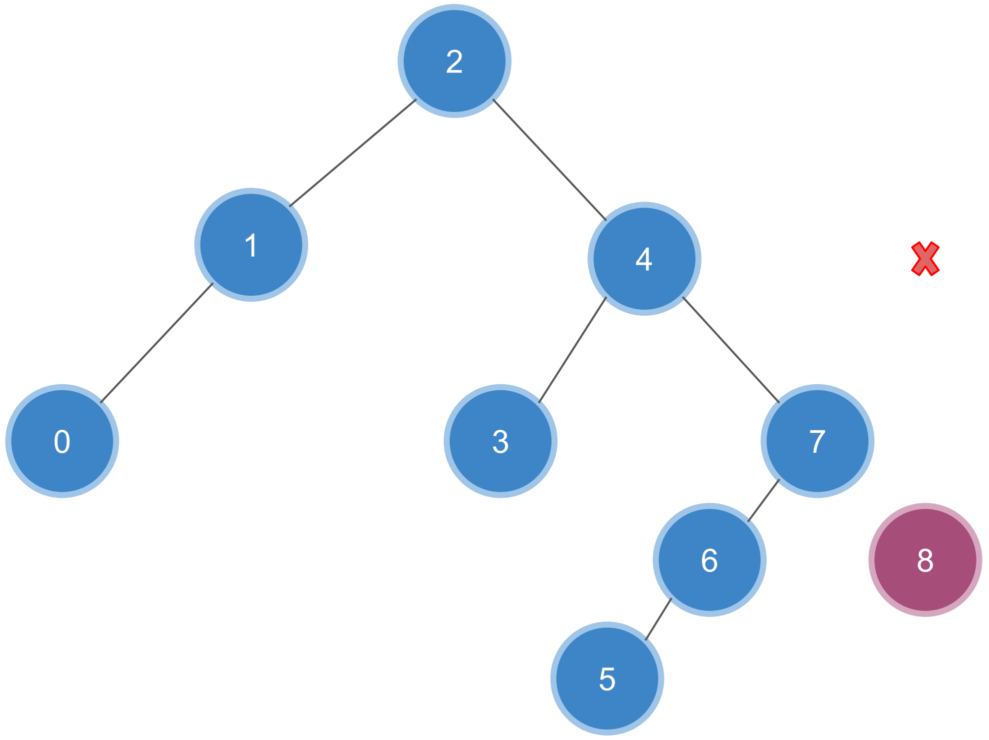
B-Tree `remove` Example (Full Node)



- Remove “2”
- It’s a full node
- Replace with in-order predecessor (“1”) ***or*** successor (“3”)
- Predecessor is left, then (while ***not*** null); right, right, right...
- Successor or right, then (while ***not*** null); left, left, left…
- Note that the predecessor/successor will be a leaf or partial node (never a full node)
- Assume implementation uses successor (node “3”)
- Remove “3”
- Replace content of node (don’t need to replace node itself)

In-order list: `0, 1, 3, 4, 5, 6`

B-Tree `remove` Example (Leaf)



- Remove “7”
- It’s a partial node
-

In-order list: `0, 1, 2, 3, 4, 5, 6, 7, 8`