8 - B-Trees

Joseph Afework
CS 241

Dept. of Computer Science
California Polytechnic State University, Pomona, CA

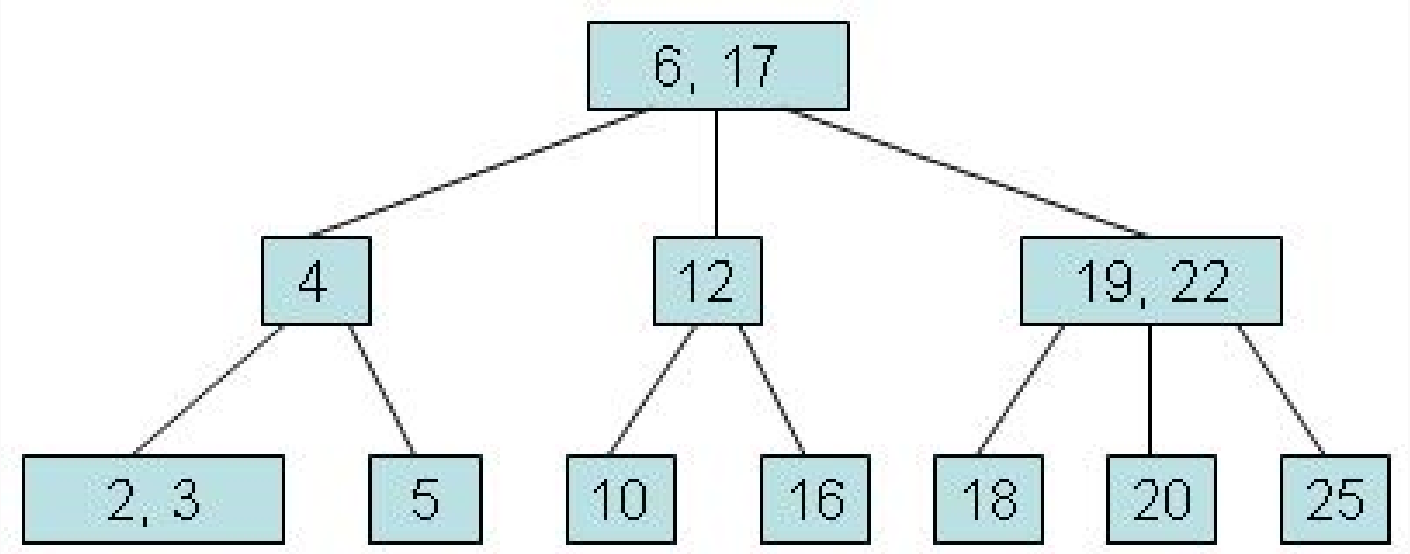
Agenda

- Intro
- Properties
- Search Operation
- Insertion Operation
- Deletion Operations
- Time Complexities
- Applications

Reading Assignment

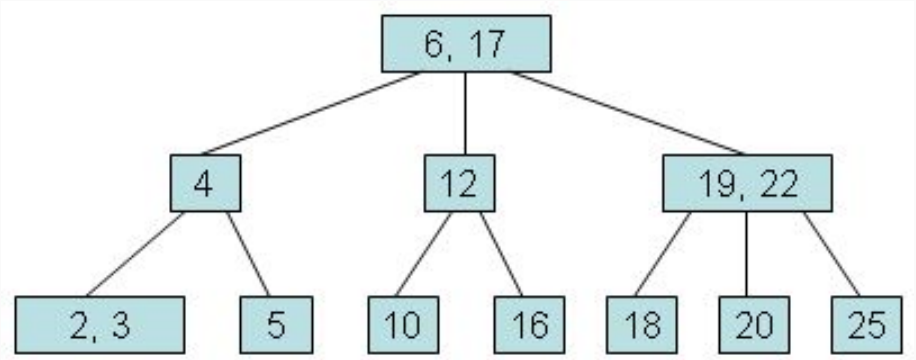
- Read Chapter 27 - Balanced Search Trees
  - Chapter 27 (Read about: AVL, Red-Black Trees, **B-Trees**)

B-Trees



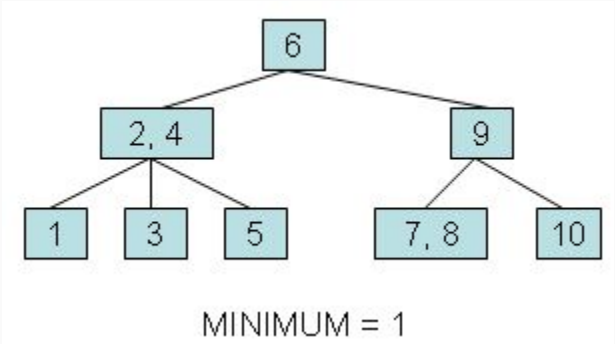
B-Trees

- B-Trees are a type of Tree data structure… **not Binary Search Tree**
  - Each node may have more than two children
- B-Trees are balanced search trees
- **A Node may have multiple keys**



Classification

- Every B-tree depends on a positive constant integer called the **minimum (m)**
  - used to determine the lowest number of keys a node can have.
  - # of keys (elements) of a node is **2m.**
  - **Max # of children for a node with k keys is k+1**



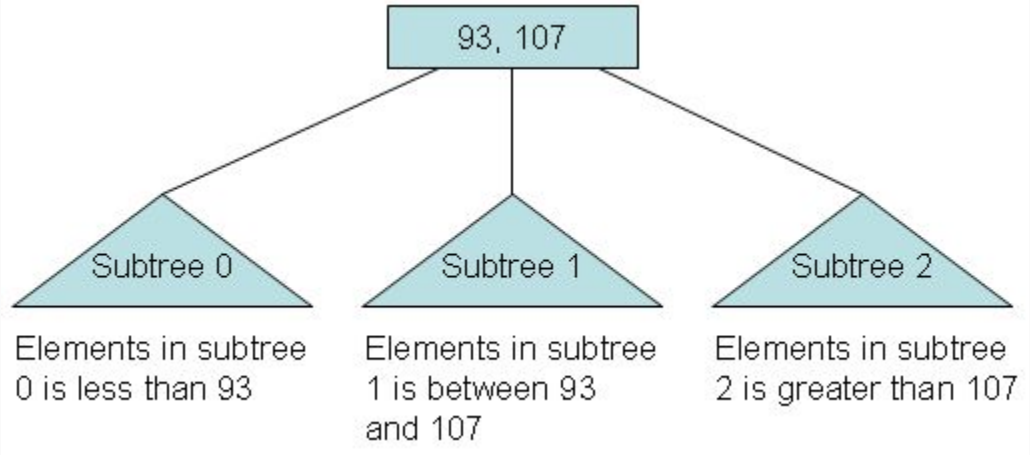
Properties

- **Rule 1**: The root can have as few as one element (or even no elements if it also has no children); every other node has **at least** MINIMUM elements.
- **Rule 2**: The maximum number of elements in a node is twice the value of MINIMUM.
- **Rule 3**: The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (at index 0) to the largest element (at the final used position of the array).

Properties Contd.

- **Rule 4**: The number of subtrees below a nonleaf node is always one more than the number of elements in the node.
- **Rule 5**: For any nonleaf node:
  1. An element at index *i* is greater than all the elements in subtree number *i* of the node, and
  2. An element at index *i* is less than all the elements in subtree number *i* + 1 of the node.
- **Rule 6**: Every leaf in a B-tree has the **same depth**. Thus it ensures that a B-tree avoids the problem of an unbalanced tree.

Properties Contd.

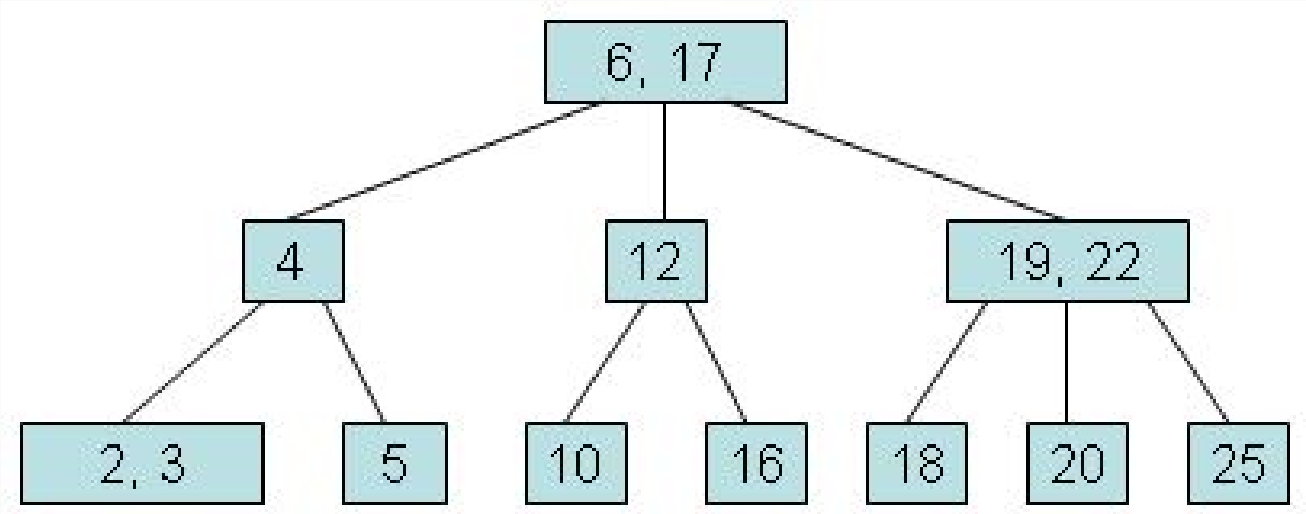


Search Operation

1) Compare the target value to all the keys at a node
2) If only 1 key found, follow BST rules (<= go to left child), (> go to right child)
3) If the target falls in between a pair of keys, traverse down to the corresponding child node. (go to child[index of right key])
4) Repeat Step 1 until node is found, or child is null

ICE 8.1 Search

Find: 19, 1, 25, 16



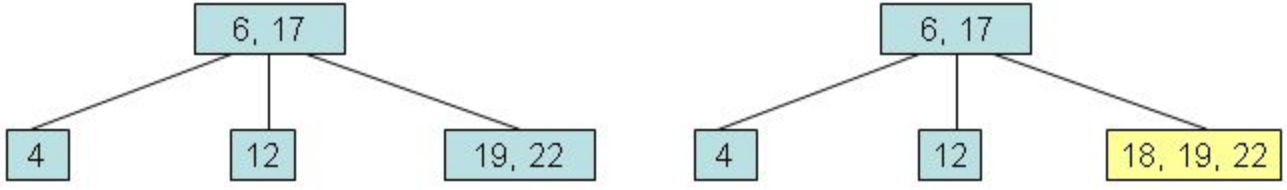
Insertion

1) Traverse the tree using the search algorithm
2) Stop when you find the location of insertion
3) add the value to the key of the insertion node
4) If inserting the key doesn't break the B-Tree rules, stop
5) Else - rules are violated (most likely too many keys),
    a) Find the middle key, and move the key up a level
    b) Go to Step 4 if node has parent, else, create new root, and insert value into root

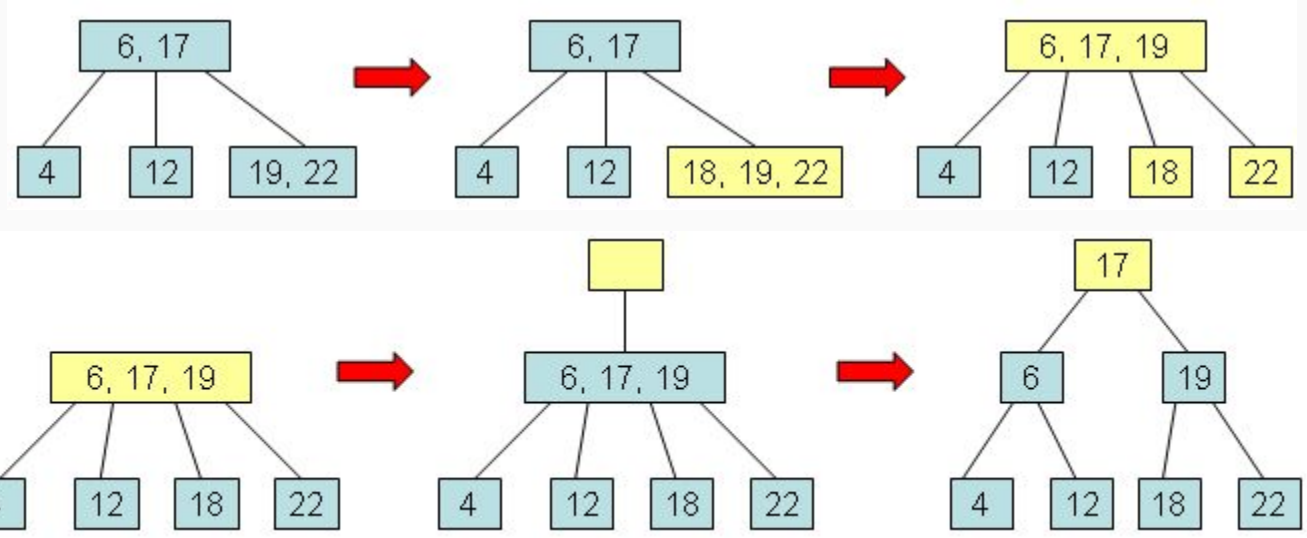
Insertion Contd.

Min = 1, Max = 2

Insert 18



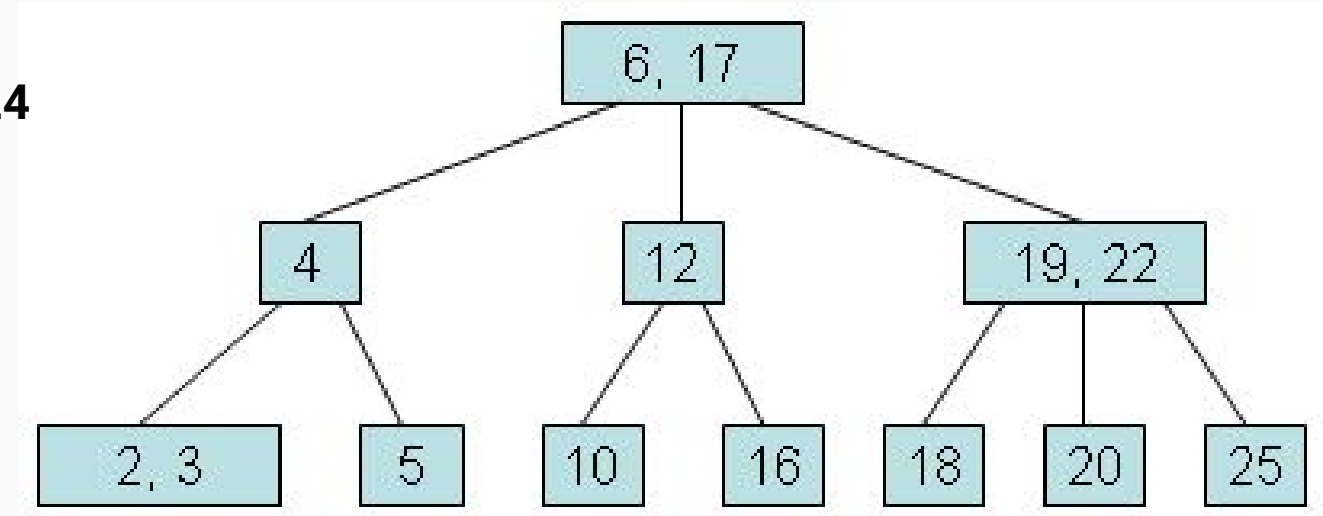
Insertion Contd.



ICE 8.2 Insertion

Min = 1, Max = 2

Insert 21,11, 23, 24

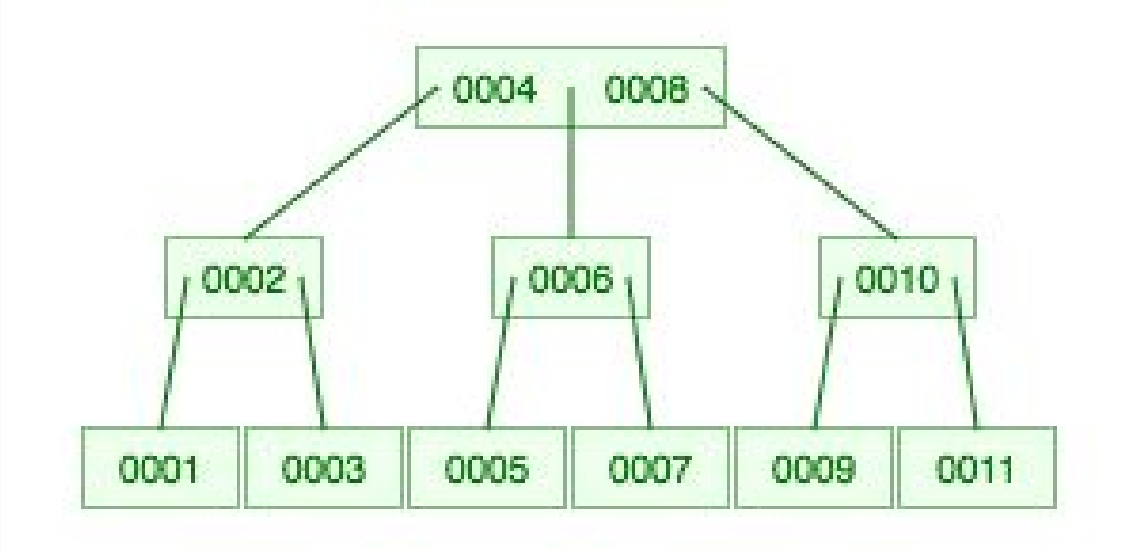


ICE 8.3 Insertion

Insert the values 1 to 11, how does the tree look?

Hint: https://www.cs.usfca.edu/~galles/visualization/BTree.html

Solution



Deletion

- Deletion is more complicated:
  - **Insertion**:
    - The height of the tree can **expand** to preserve the properties of the b-tree
  - **Deletion**:
    - The height of the tree can **shrink** to preserve the properties of the b-tree
      - Some rebalancing operations may be required
      - Some rotation operations may need to occur

Deletion

3 possible cases…. With subcases…..

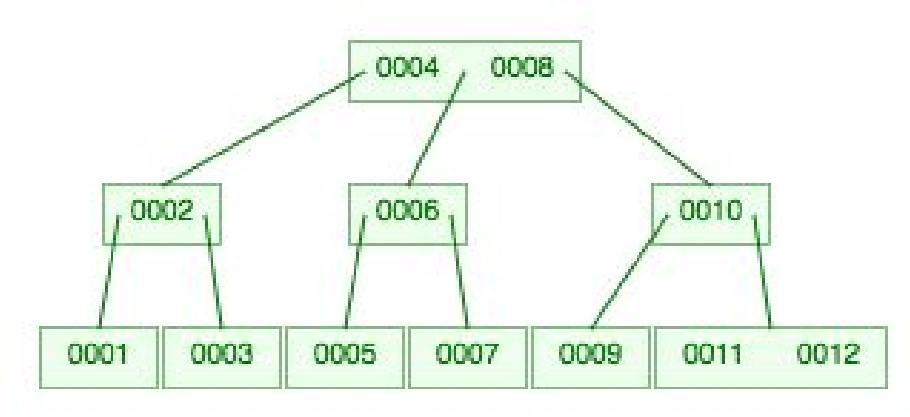
1. Deletion rule in leaf node, (where after deletion number of keys in node is still valid)
2. Deletion rules if target exists in an internal node (non-leaf) - rotate a key in (from left or right child… or merge children)
3. Deletion rules if child nodes can’t borrow a replacement node

Case 1

Case 1: If the target is in a leaf node **AND** the leaf node has more than the minimum number of keys

Fix: Delete the key

Ex. **Delete 12**

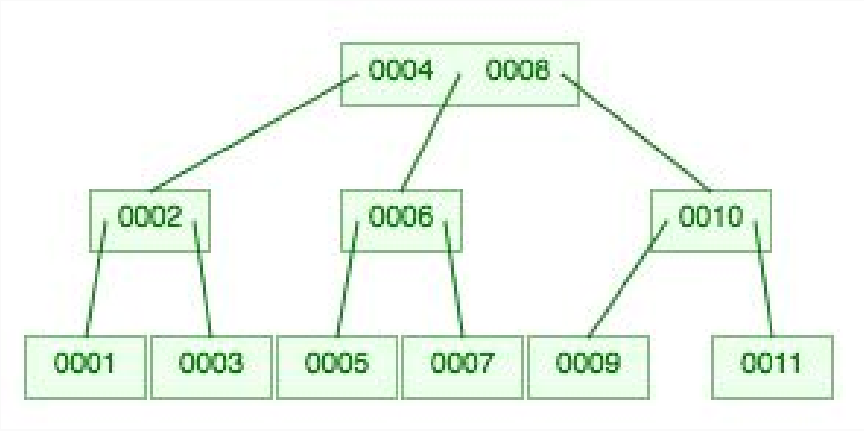


Case 1

Case 1: If the target is in a leaf node **AND** the leaf node has more than the minimum number of keys

Fix: Delete the key

Ex. **Delete 12**



Case 2

Case 2: If the target is in a leaf node **AND** the leaf node has exactly the minimum number of keys before deletion…. Consider the following subcases...

2a: If the target node has a sibling node with **at least min + 1** keys...

2b: If the target node siblings also have the minimum number of keys...

Case 2a

Case 2: If the target is in a leaf node **AND** the leaf node has exactly the minimum number of keys before deletion…. Consider the following subcases...

2a: If the target node has a sibling node with **at least min + 1** keys

> **Fix:** Move down the parent key to the target node, and move the appropriate key from the sibling node, into the empty parent slot, Then Delete the target.

Case 2a Example

Delete: 9

Move 10 Down… Move 11 Up.. Delete 9

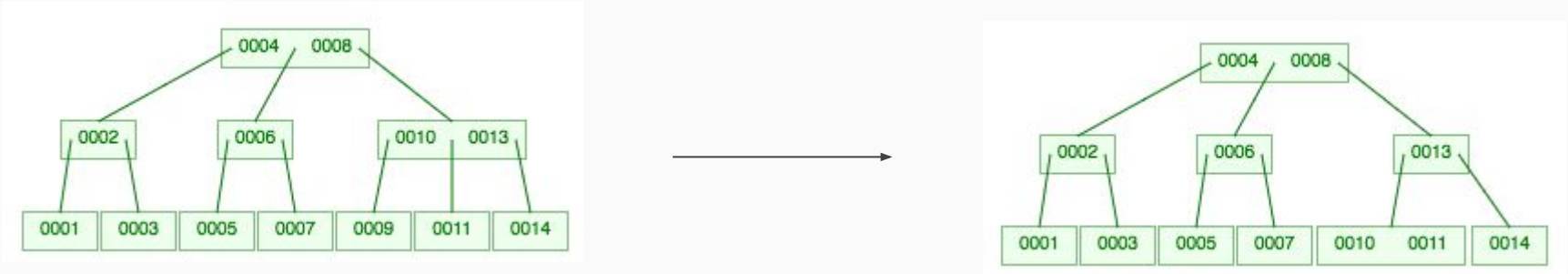
Case 2b

Case 2: If the target is in a leaf node **AND** the leaf node has exactly the minimum number of keys before deletion…. Consider the following subcases...

2b: If the target node siblings also have the minimum number of keys...

> **Fix:** Merge the target node with one of it’s siblings, bring down the parent key as a median. Delete the target

Case 2b Example



Delete: 9

Move 10 down, Merge 9, 10 and 11… Delete 9

Case 3

Case 3: If the target node is an internal node:

Fix:

1) **Select a replacement key:** Find the largest key in the left subtree of the target value, or the smallest key in the right subtree of the target value. This key will be called the replacement key.
2) Delete the target value, and replace it with the replacement key.
3) **Leaf node should now be empty (violation):** Rebalance the tree.

Case 3: Rebalance

- Rebalance the tree (reduce the height if needed).

- Starts from a leaf (empty) and moves upwards, toward the root, until the tree is balanced.

Case 3: Rebalance

- If the missing node's right sibling exists and has **more than** the min number of keys... **Rotate Left… Think Case 2**
  1. Move parent key down into missing node
  2. Move first key in the right sibling up to the parent

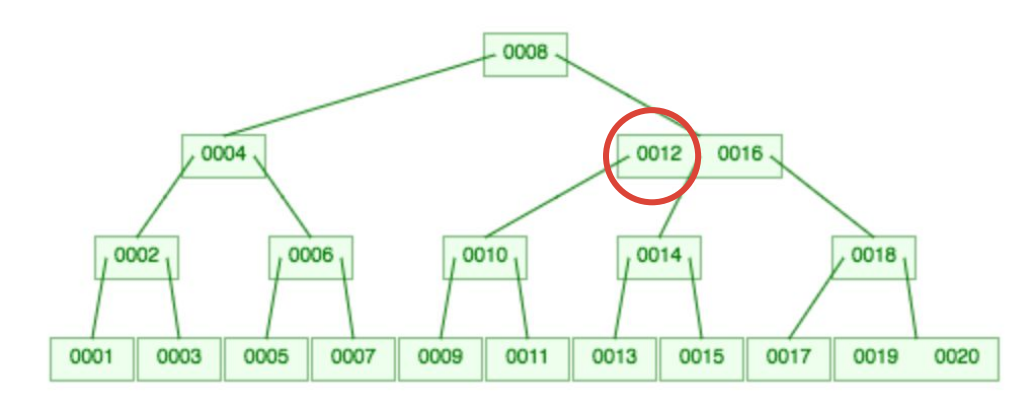
- If the missing node's left sibling exists and has **more than** the min number of keys.. **Rotate Right… Think Case 2**
  1. Move parent key down into missing node
  2. Move the last key in the left sibling up to the parent

Case 3: Rebalance Contd.

- If both immediate siblings have only the min number of keys.. **Merge...**
  1. Move a parent key down into the left node
  2. Move all keys from the right node to the left node
  3. Remove the right node
     - If the parent is the root and is now empty, make the merged node the new root (tree becomes shorter)
     - if the parent has fewer than the min number of keys, then **rebalance the parent (restart)**

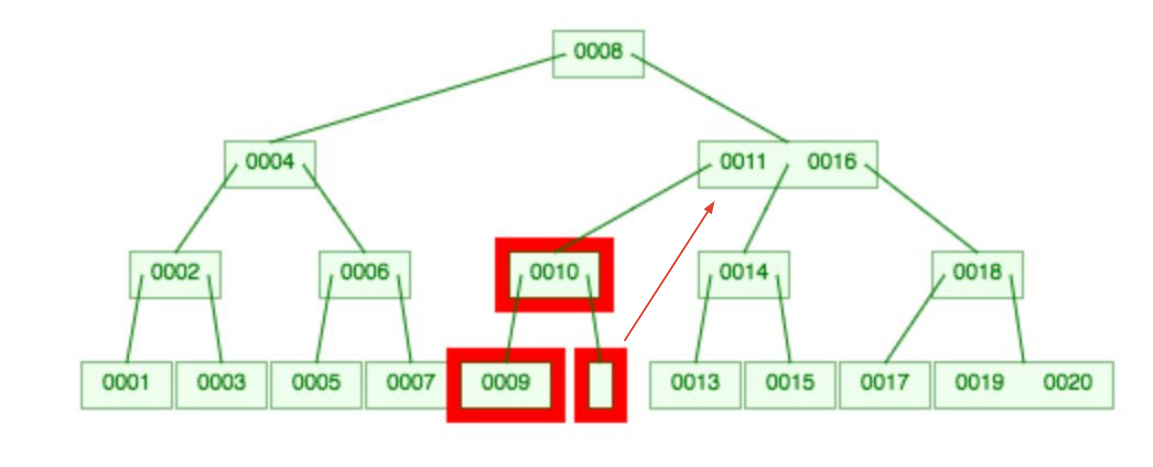
Case 3 Example

Delete: 12



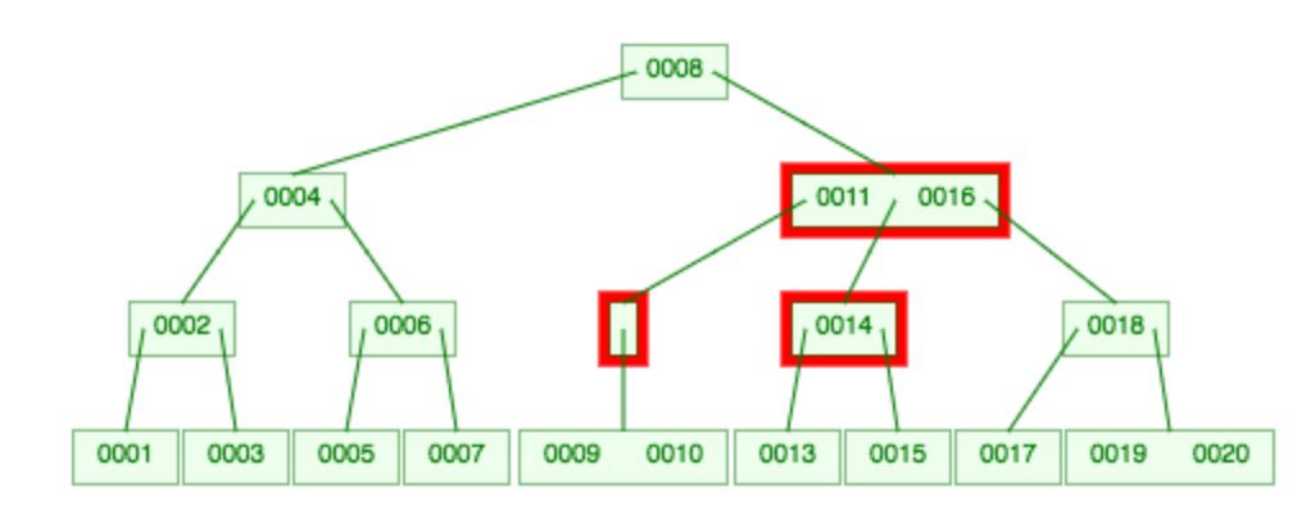
Case 3 Example Contd.

Find largest value in Left Subtree, move to empty position



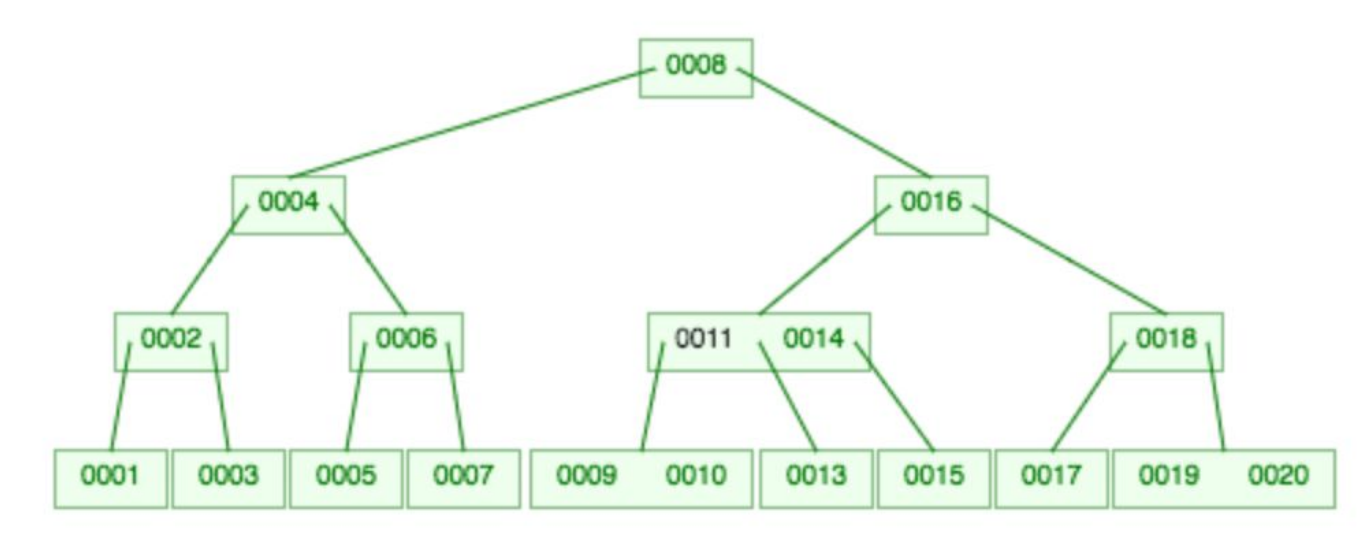
Case 3 Example Contd.

Continue Rebalancing From parent of deleted node



Case 3 Example Contd.

Pull parent down into the left node and merge right node



Time Complexity

Suppose a B-tree has ***n*** elements and ***M*** is the maximum number of children a node can have.

What is the **maximum depth** the tree could have?

$\log_{M/2} n$ - Every node has the minimum children = (M/2)

What is the **minimum depth** the tree could have?

$\log_{M} n$ - Every node has the maximum children = (M)

Time Complexity Contd.

Search, Insert and Delete:

B-tree with n elements is $O(\log n)$

Applications

- Databases
  - Ex. used by Sqlite internal storage engine
- Data-Set Indexing
  - Mapping to a larger data set