Self-Balancing Search Trees

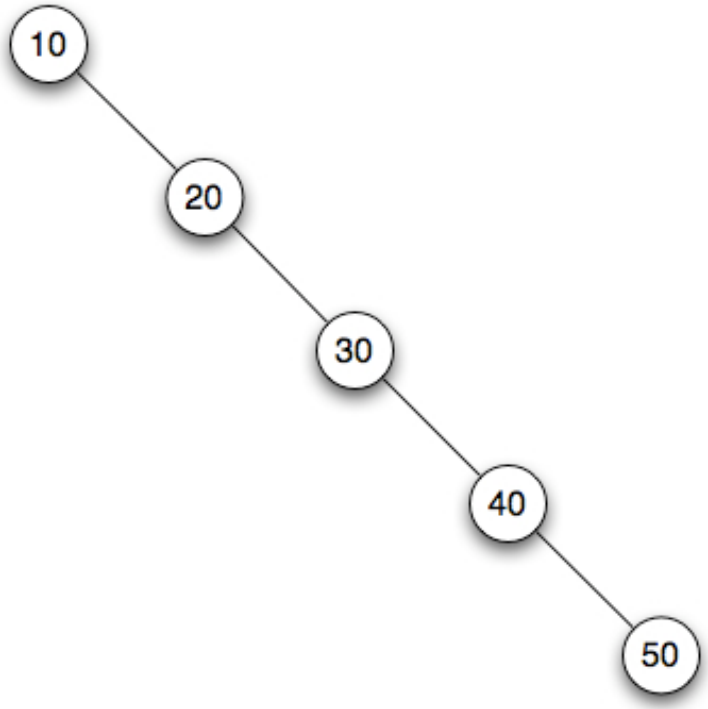
Chapter 9

Chapter Objectives

- To understand the impact of “height balance” on the performance of binary search trees
- To learn about the AVL tree for storing and maintaining a binary search tree in balance
- To learn about 2-3 trees, 2-3-4 trees, and B-trees and how they achieve balance (will not cover Red-Black trees)
- To understand the process of search and insertion in each of these trees and to be introduced to removal

What Is “Balance”

- Searching in an unwieldy search tree could cost O(n) steps



Balanced Search Tree Scheme

- Define the height of a tree to be the number of nodes in the longest path from the root to a leaf node
- In general we say that a tree node is a “balanced” node if there isn’t much different between the height of its left tree and the height of its right tree
  - Many binary search tree schemes force “balance” at every node so that the resulting tree has height at most c log(n) for any n node tree, for a constant c

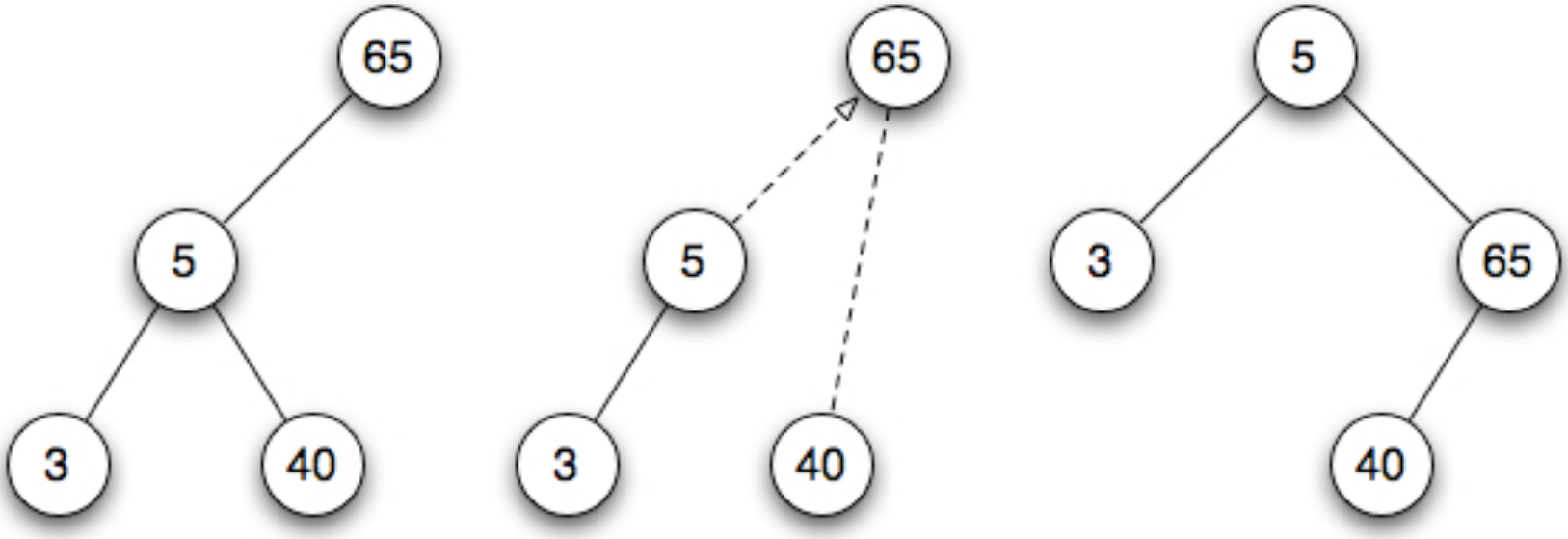
Balanced Search Tree Scheme

- When insertion or deletion is executed:
  - The “balance” may be broken
  - So the “balance” is fixed on the nodes that have been affected with a constant number of operations at each such node
  - The end result is an O(log(n)) method

Rotations of Node in a Binary-Search Tree

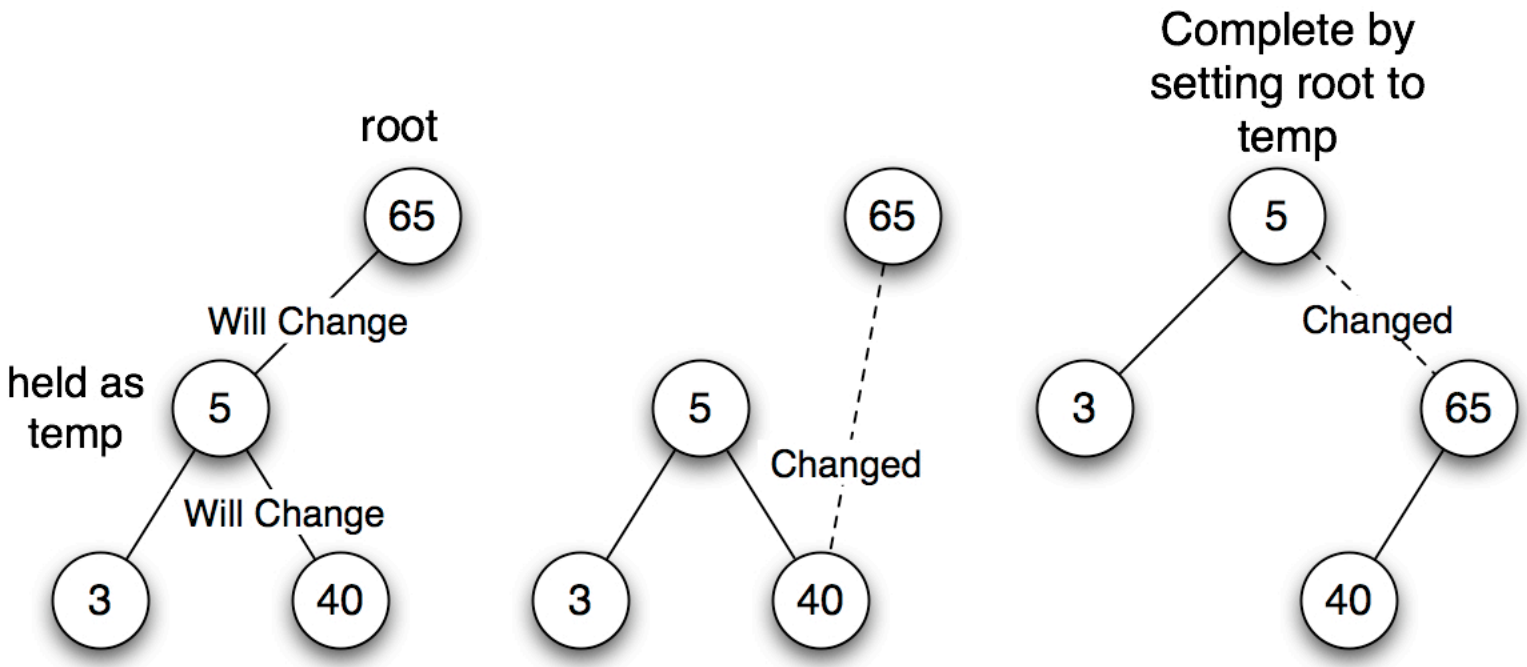
Basic operations for balancing in self-balanced trees.

Right Rotation: L-child -> root
root-> R- child
LR-grandchild -> RL-grandchild



Step-by-step Right Rotation

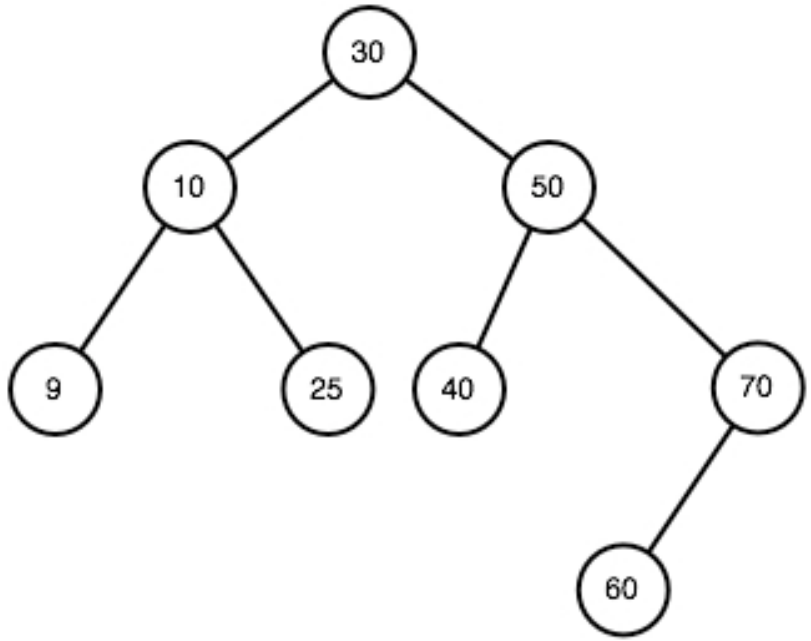
- Set temp to root.left so as not to lose it.
- Change root.left to temp.right
- Set temp.right to root
- Change root to temp



AVL Tree

AVL Tree

- A node in a tree is balanced if the difference between the height of its left subtree and the height of its right subtree is 0 or 1.
- **An AVL tree is a tree in which every node is balanced.**

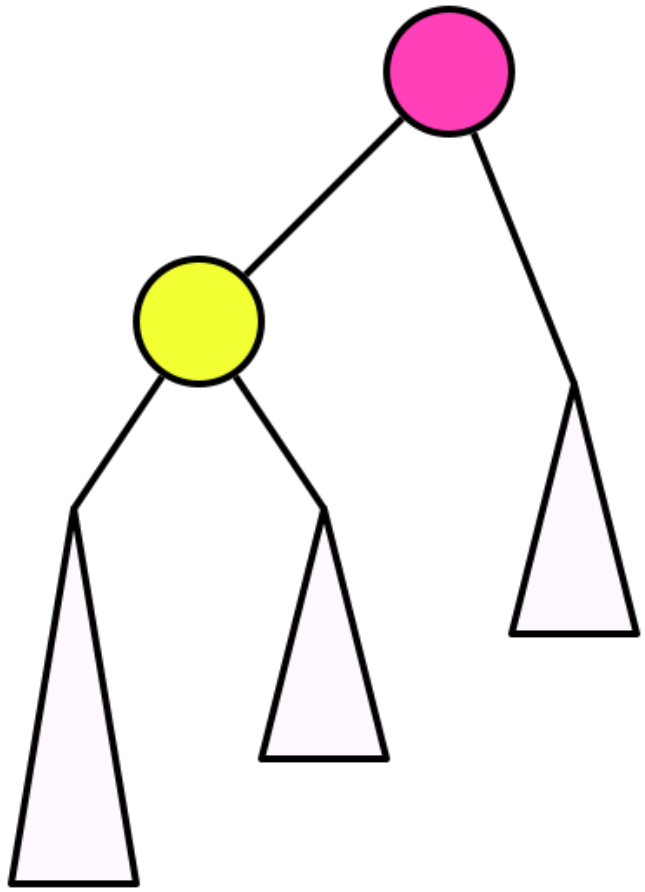


Insertion/Deletion in an AVL Tree

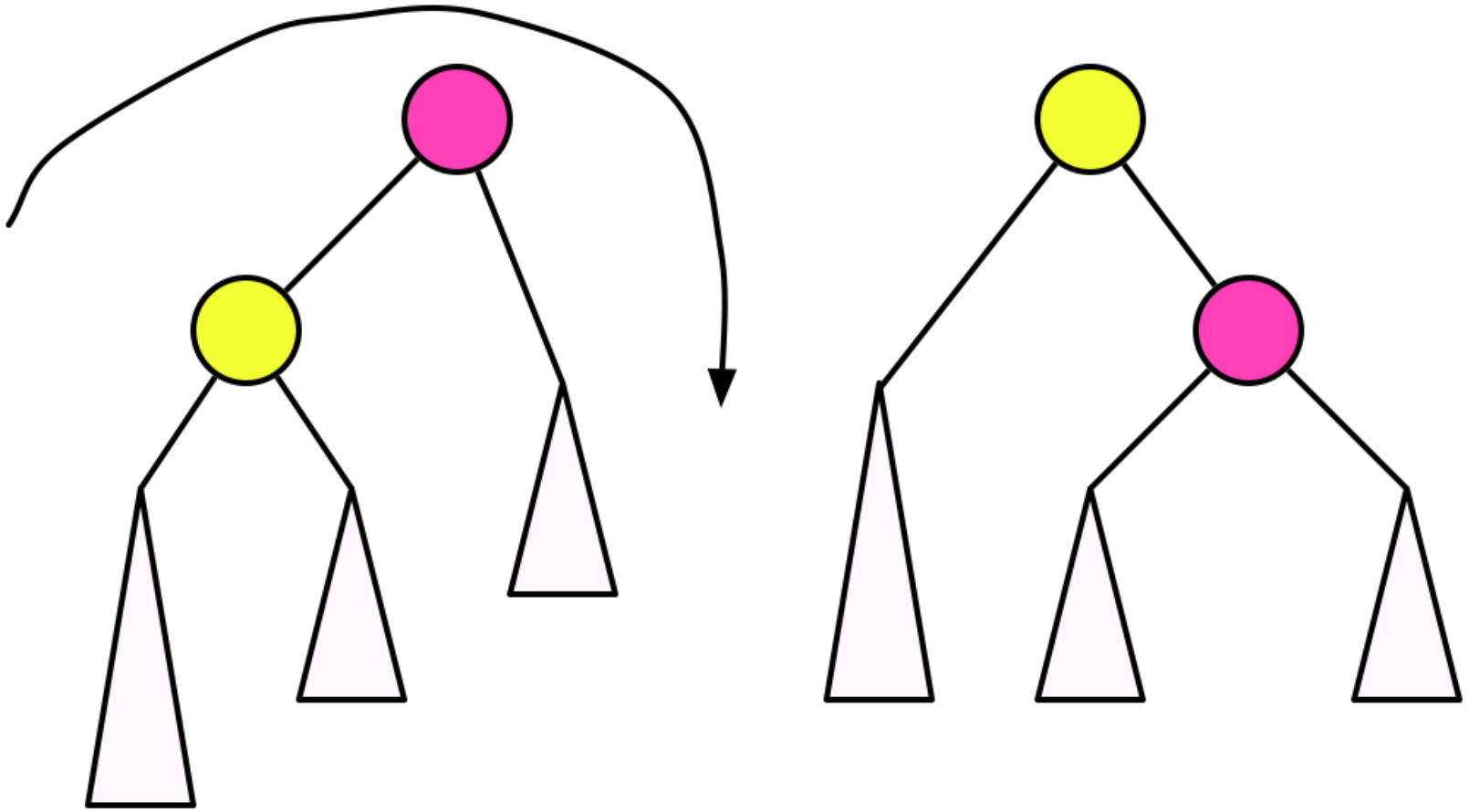
- Execute insertion/deletion as in the usual binary search tree.
- The operation may break the balance of some subtrees, but the roots of such trees are on the straight path from the location of insertion/deletion to the root.
- Climb from the location to the root and fix all problems.

Left-Left Tree

- A tree is **left-heavy** if the height of the left subtree is more than the height of the right subtree
- A **left-left tree** is a tree in which both the root and its left subtree are left-heavy



Re-balancing Left-left Trees



Right-right Tree

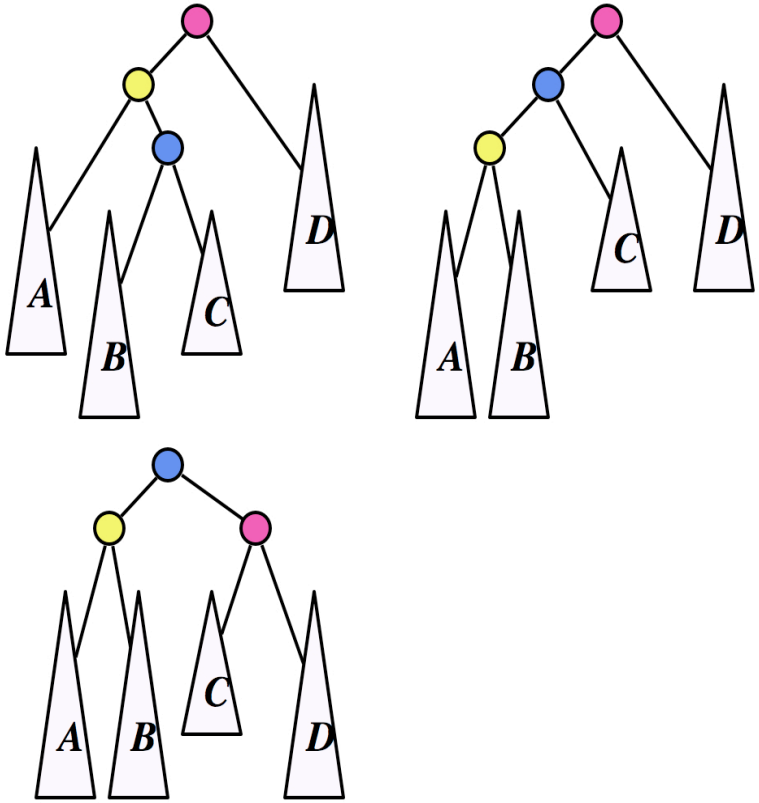
- A tree is **right-heavy** if the height of the right subtree is more than the height of the left subtree.
- A **right-right-tree** is a tree in which both the root and its right subtree is right-heavy
- By symmetry, use one left rotation to re-balance

Left-Right Tree

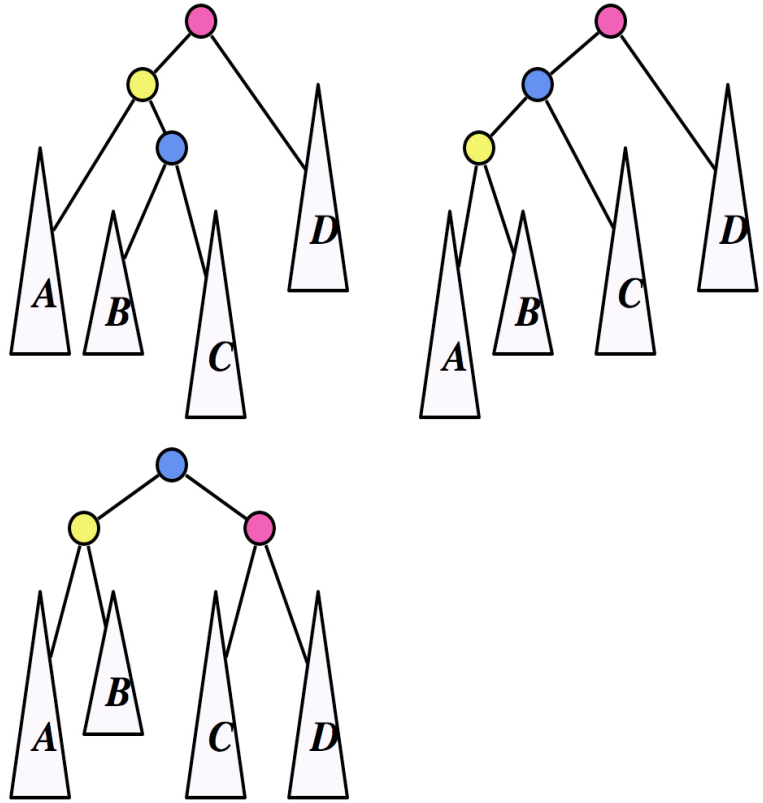
- In a left-right tree, the root is left-heavy and its left subtree is right-heavy
- Needs one left rotation and one right rotation

Re-balancing a Left-right Heavy Tree
"L-rotate L-child; R-rotate Parent"

Case 1: Left-right-left is the heaviest



Case 2: Left-right-right is the heaviest



Right-left Tree

- In a right-left tree, the root is right-heavy and its right subtree is left-heavy
- By symmetry, needs one rigt rotation and one left rotation

2-3 Trees

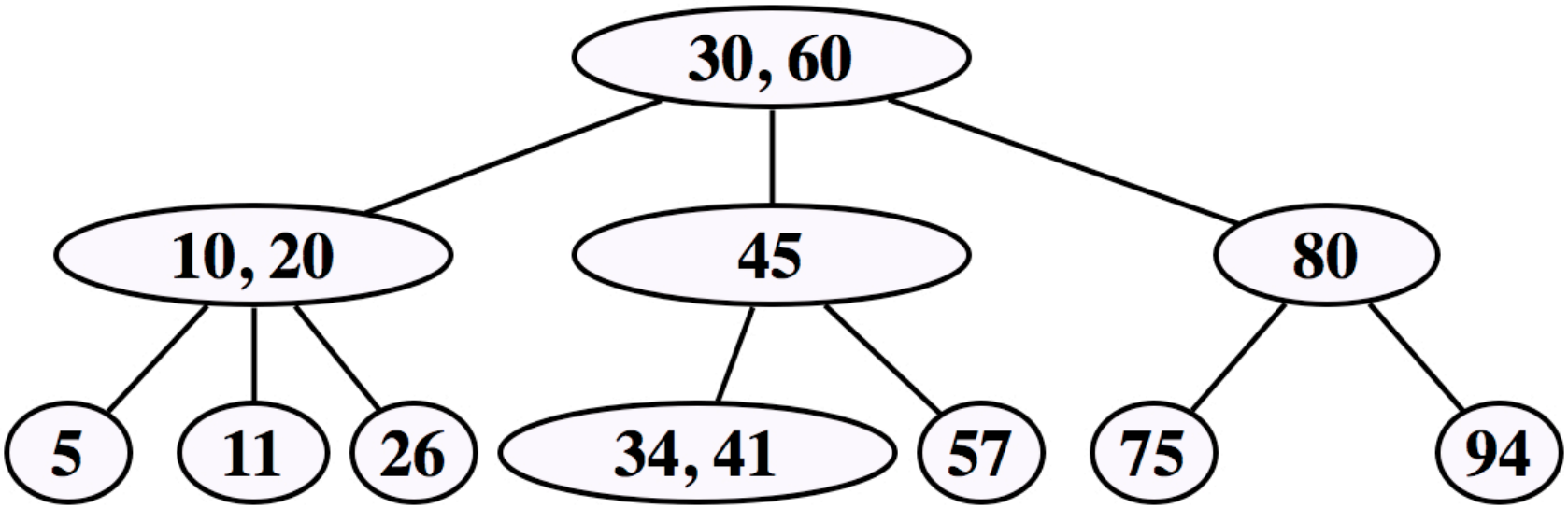
2-3 Trees

- The number of possible children for a non-leaf is 2 (“2-nodes”) or 3 (“3-nodes”)
- A 2-node is the same as a binary search tree node
- A 3-node contains two data fields, ordered so that first is less than the second, and references to three children
    - One child contains values less than the first data field
    - One child contains values between the two data fields
    - Once child contains values greater than the second data field
- 2-3 tree has property that all of the leaves are at the same distance from the root

Searching in a 2-3 Tree

- Choose a child by comparing the target against the key (s) stored at the node.

A 2-3 Tree



Insertion of a Key into a 2-3 Tree

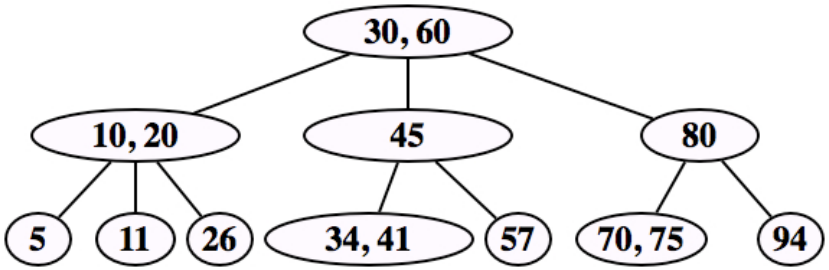
- Execute search; start from the leaf node where the search stopped.
- If the number of keys in the leaf is one, add the new key
- Otherwise, do the following:
  - Set a, b, c (a<b<c) to the two existing keys and the new key
  - Set x to the parent of the leaf
  - Create two nodes, u and v, who have a and c as the sole key, respectively
  - Call an upward insertion procedure on the quadruple (x, b, u, v)

Insertion of (x,b,u,v)

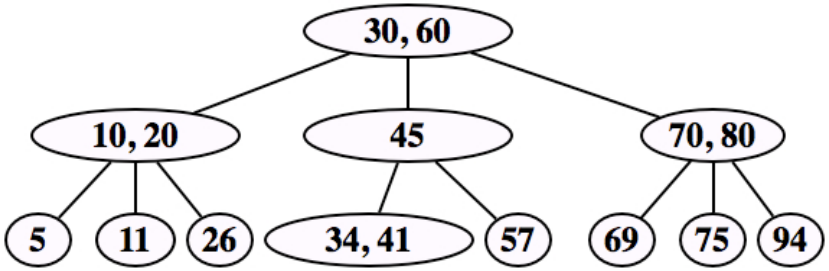
- Let p be the parent of x.
- (Case 1) p has only one key
  - Insert b into p and insert references u and v into
- (Case 2) p has two keys
  - Let b’ be the median of the existing two keys and b
  - Create a new node x’ whose unique key is b’ and whose children u’ and v’, where u’ takes care of the smaller of the three and v’ the larger
  - Make a recursive call to the insertion procedure with (x’, b’, u’, v’)

Example of Insertion

After insertion of 70

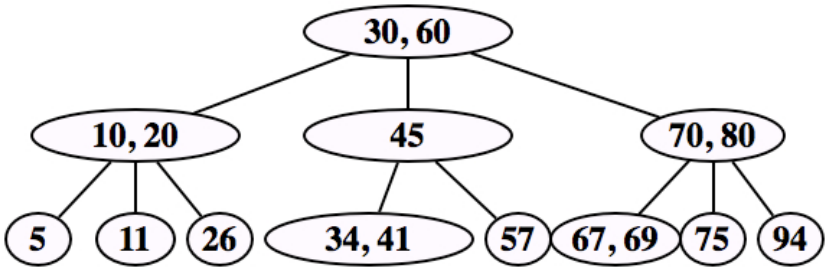


Insertion of 69: 70 migrated

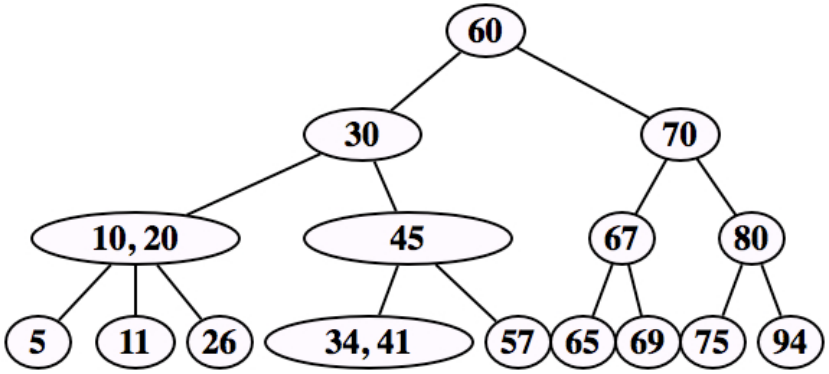


Example (cont'd)

Insertion of 67



Insertion of 65: 67 then 70 migrated



Removal from a 2-3 Tree

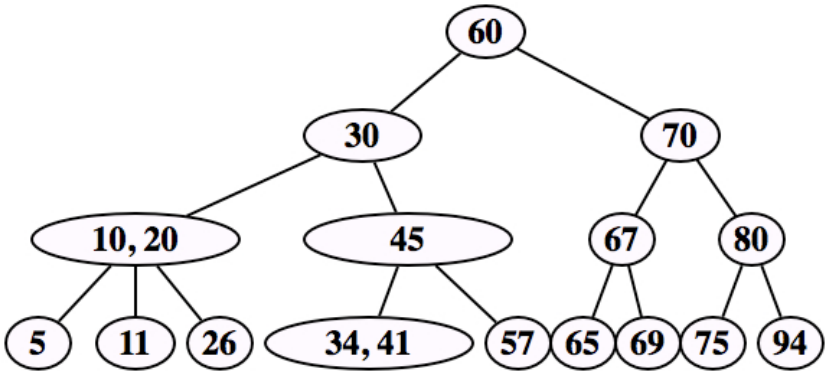
- Preparation:
  - If the key to be removed is not in a leaf node, exchange it with the inorder predecessor; that is, the largest key in the rightmost node in the subtree immediately to the left of the key.
  - If either the tree consists of only one node or the key is a leaf with one more key, remove the key.
- Now the key to be removed is the sole key in a leaf node.
- Let R be the key, let U be the leaf node, and let P be its parent

Removal from a 2-3 Tree

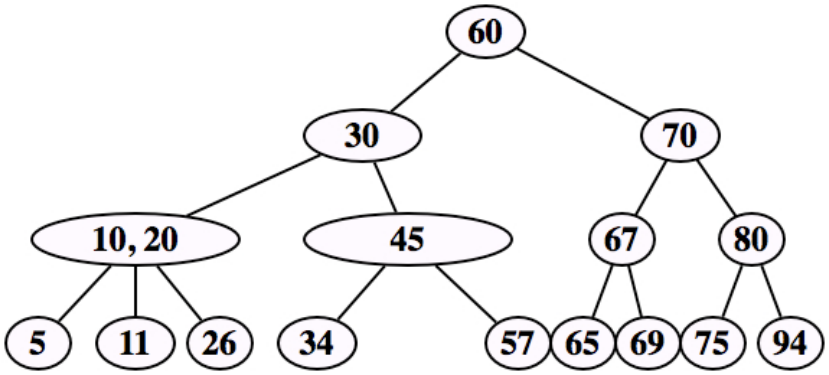
- If P has two keys, remove U, and reinsert one key of P
  - If U is the left child, the smaller key
  - Otherwise, the larger key
- Else if the unique sibling of U has two keys, move the key of P to U and put one key of the sibling into P
  - From the left sibling: the larger key
  - From the right sibling: the smaller key
- Else, remove U; move P's key to the unique sibling, and then enter resolution stage

Example

Original

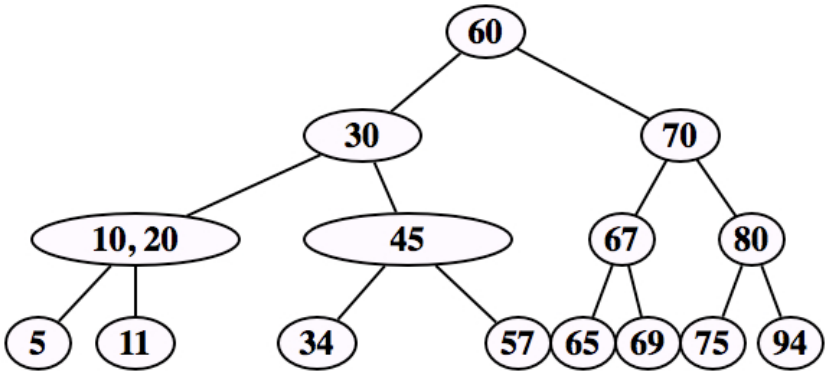


Removal of 41

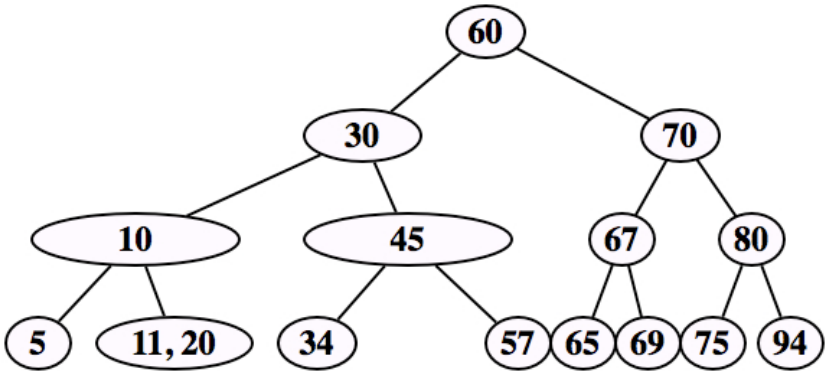


Example (cont'd)

26 Removed

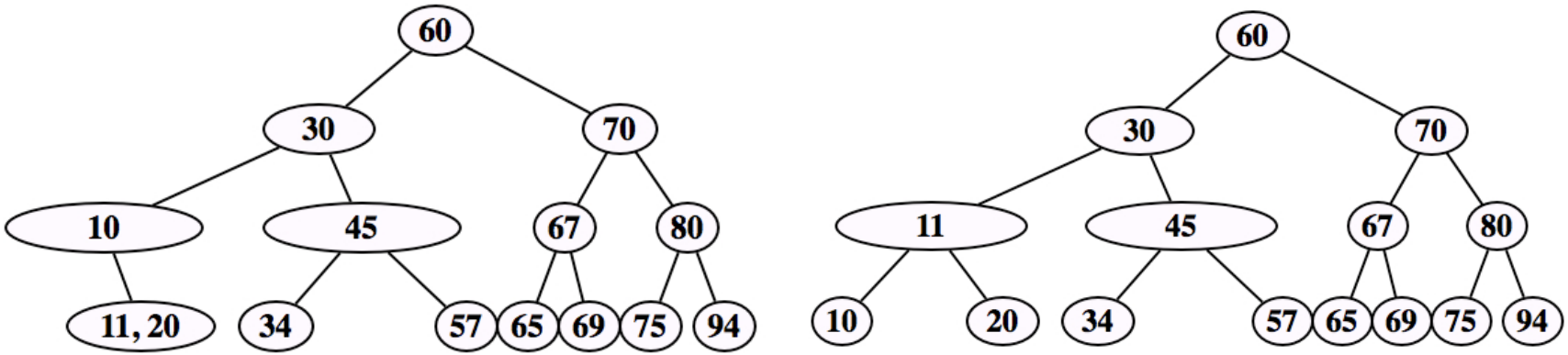


20 removed and inserted



Example (cont'd)

5 Removed



Resolution Stage

- The loop-invariant during this stage is:
  - ***There is a non-leaf node, P, that has no key and has only one child, V***

Resolution Strategy

- If P is the root of the tree, simply change the root to V.
- Else if P has only one sibling, which has only one key
  - Merge P and P's sibling, insert P's parent there -> the location of resolution moves to P's parent.

Resolution Strategy

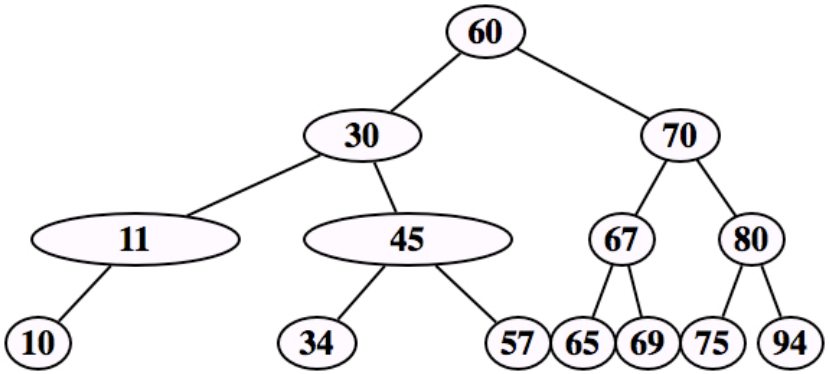
- Else if P has a sibling, S, with two keys.
  - 1key at the parent, 2 at S, and 0 at P == 3 total
  - 1 subtree at P and 3 at S == 4 total
  - Reorganize these into:
    - One key at the parent
    - One key at P
    - One key at S
    - Two subtrees at P
    - Two subtrees at Q

Resolution Strategy

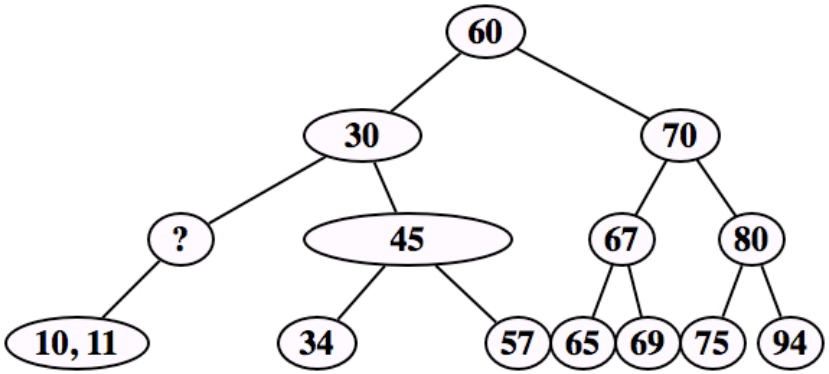
- Else (P has two siblings, S and T, each with only one key)
  - 2 keys at the parent, 1 at S, 1 at T, and 0 at P == 4 total
  - 1 subtree at P, 2 each at S and T == 5 total
  - Reorganize these into:
    - One key at the parent
    - One key at P
    - Two keys at S
    - Two subtrees at P
    - Three subtrees at S

Resolution

Remove 20

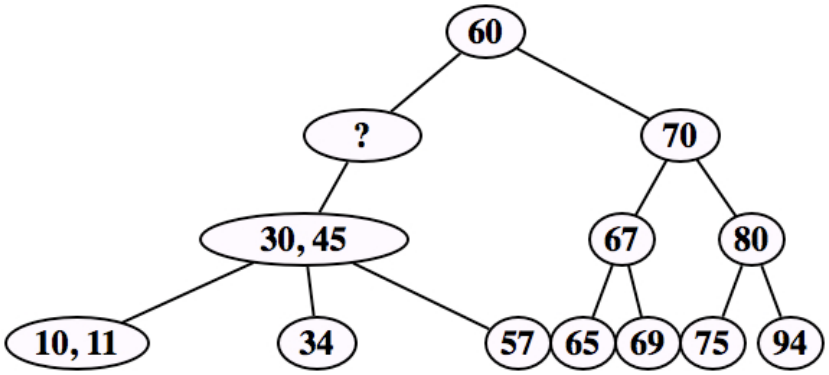


Combine 10 and 11 to arrive at the loop-invariant

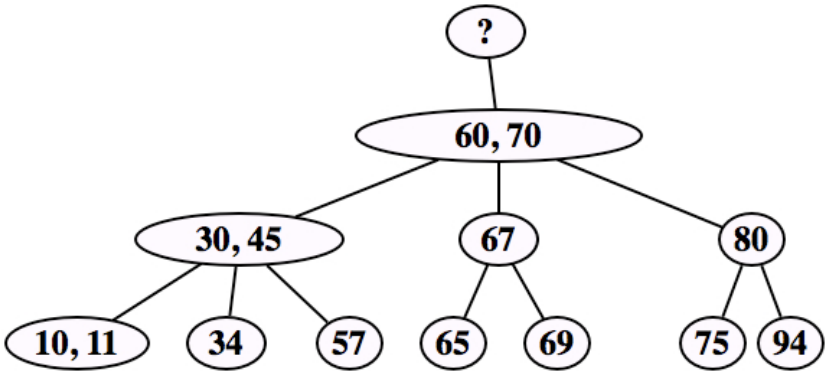


Resolution (cont'd)

30 joins 45

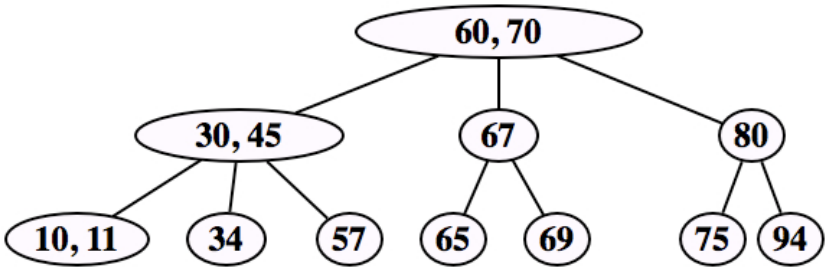


60 joins 70; remove the root

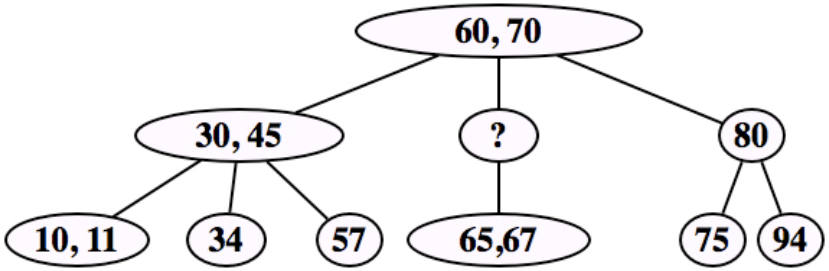


Removal (cont'd)

Initial

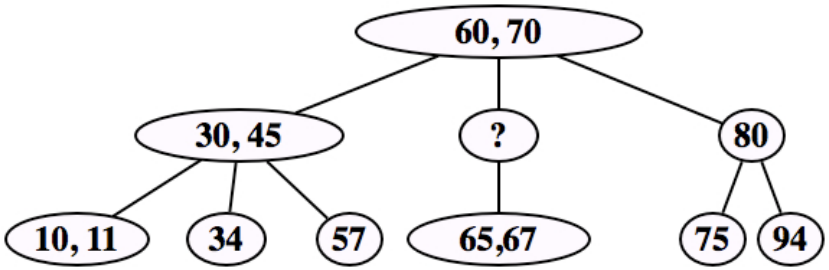


Removal of 69

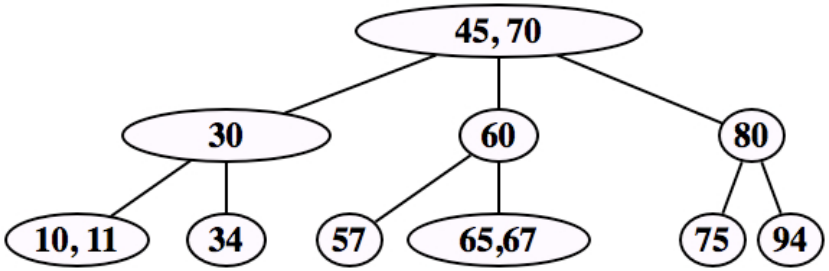


Removal (cont'd)

Situation from last page

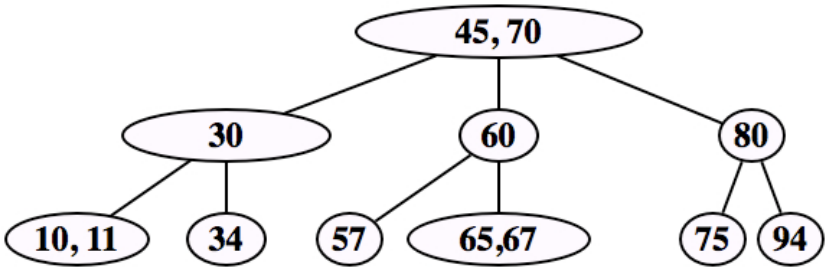


Moving 45, 60, 57

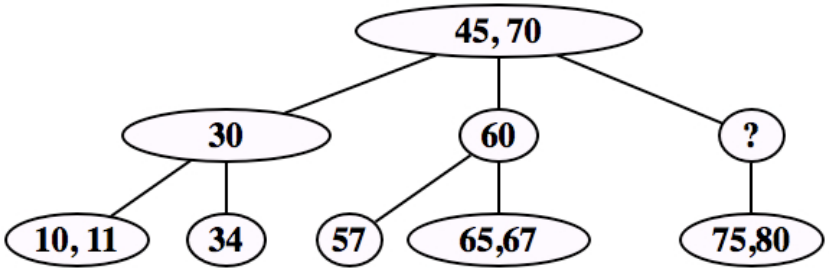


Removal (cont'd)

Initial

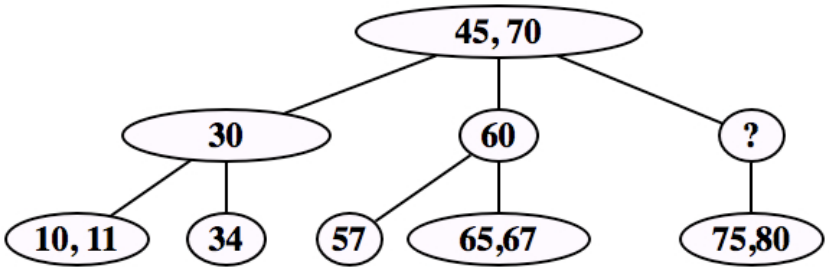


Removal of 94

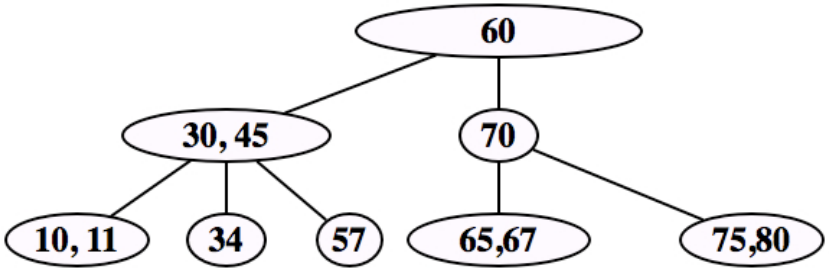


Removal (cont'd)

Situation from last page



Reorganization



General Removal Policy for Removing a key

- If the key is not in the leaf node,
  - Find the leftmost key *k* in the child immediately to the right of *k*
  - Move *k* to the key's position and start the process of removing *k*

Removal of a Key Located at a Leaf: Easy Cases 1 & 2

- If the leaf has two keys, remove the designated key
- If the leaf has one key and the leaf is the root, the tree is now empty

Removal of a Key Located at a Leaf: Easy Case 3

- If the leaf has one key and either:
  - The leaf has two siblings or
  - One sibling has two keys

  Then there are 5 – 8 keys in the subtree rooted at the parent, so 4 – 7 keys after removal.

  Turn the remaining keys into a height one 2-3 tree.

Removal of a Key Located at a Leaf: Hard Case

- There is only one sibling and the sibling has only one key →There are only two key remaining in the subtree rooted at the parent

- Steal the parent's key and the sibling's key and eliminate the sibling. This turns it into a situation:
  - ***(*) There is a node with:***
    - ***No keys***
    - ***Just one child with two keys and three children. Either***
      - ***All of the three children are null or***
      - ***All of the three children have the same height***

Resolving (*)
X is the node with no keys

- ***If X is the root, turn X's unique child into the root.***

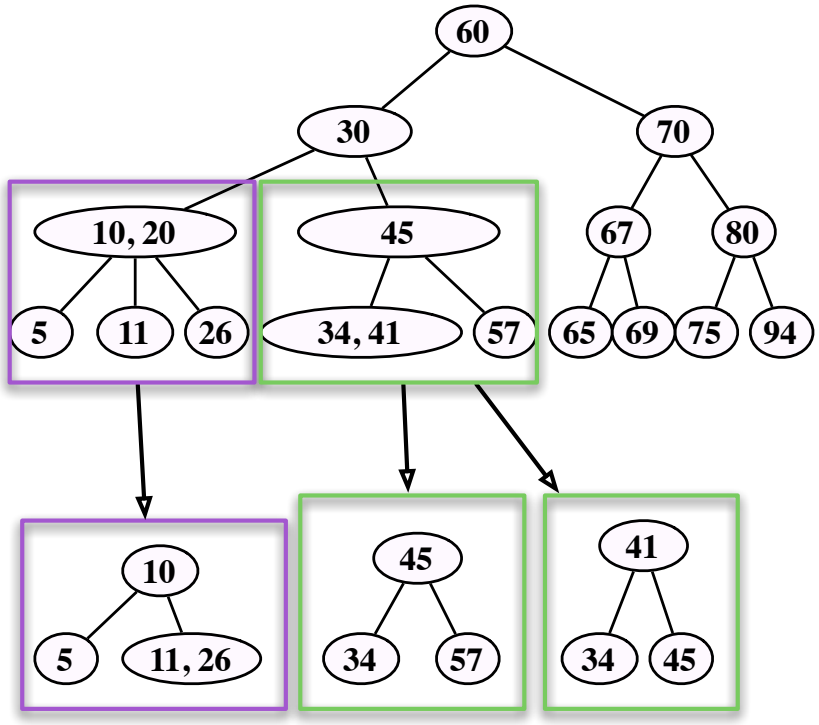
- ***If the number of keys stored in the siblings of X is at least 2, then you can resolve the situation immediately.***

- If the number of keys stored in the siblings of X is just 1, then X's parent has just 1 key, there is one sibling of X, and that sibling has 1 key. So,
  - Steal the two keys into X, remove the sibling.
  - ***Now the parent has become X***

Instant Resolution of (*)

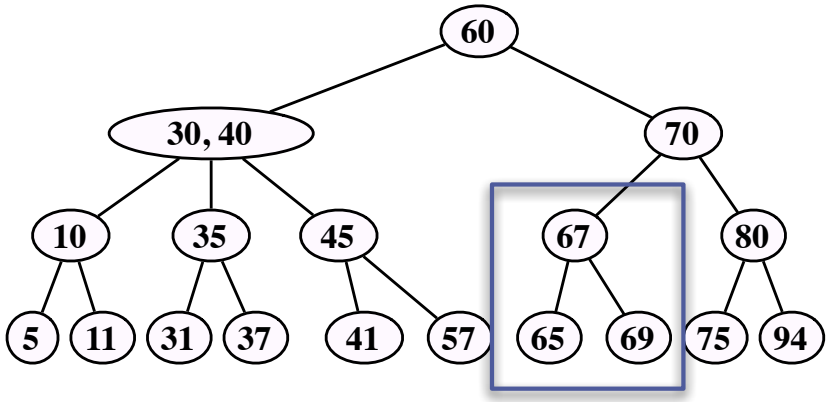
- Let A be the number of keys with the parent and the siblings combined
- Let B be the number of children of the siblings
- Then we have: 3<= A <= 6 and A = B
- The child of X and the children of the siblings of X have the same height and there are A+1 of them
- They can be organized into:
  - ***One key at the parent, one sibling, and 1 or 2 key at self and at the sibling (A is at most 5); or***
  - ***Two keys at the parent, two siblings, one key at self, one key at one sibling, and two keys at the other***

Deletion Example



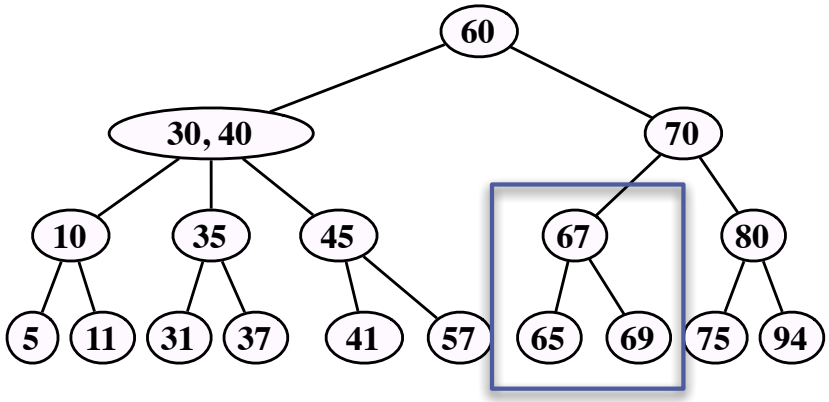
- Removing 34 or 41 is easy … just remove it
- Removing 5, 11, or 26 requires reorganization of four keys from 5, 10, 11, 20, and 26
- Removing 57 requires reorganization of 34, 41, and 45

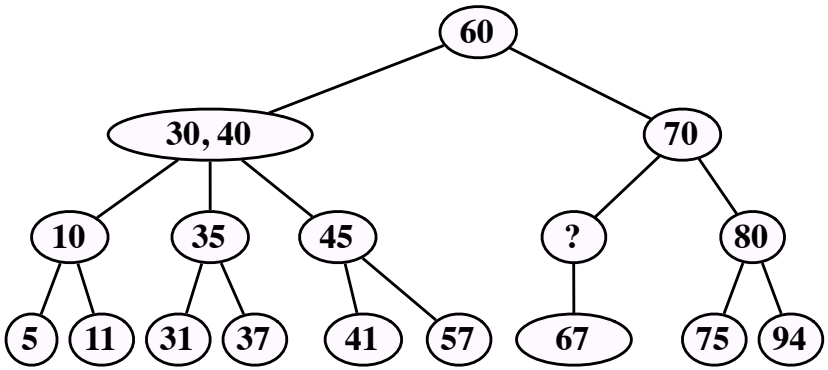
Deletion Examples



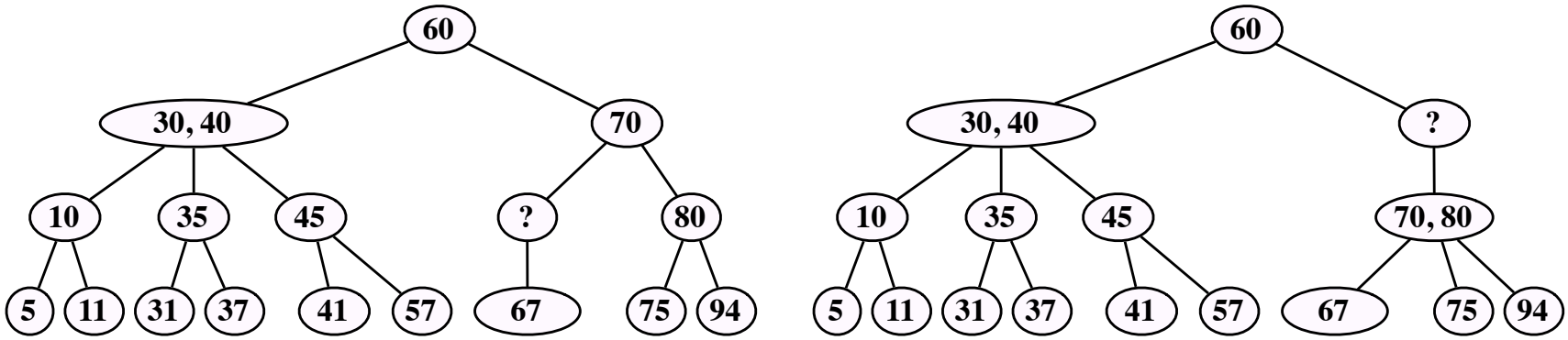
- Removal of any leaf in this 2-3 tree triggers the situation (*)
- Let's try to remove 69

First step: before and after

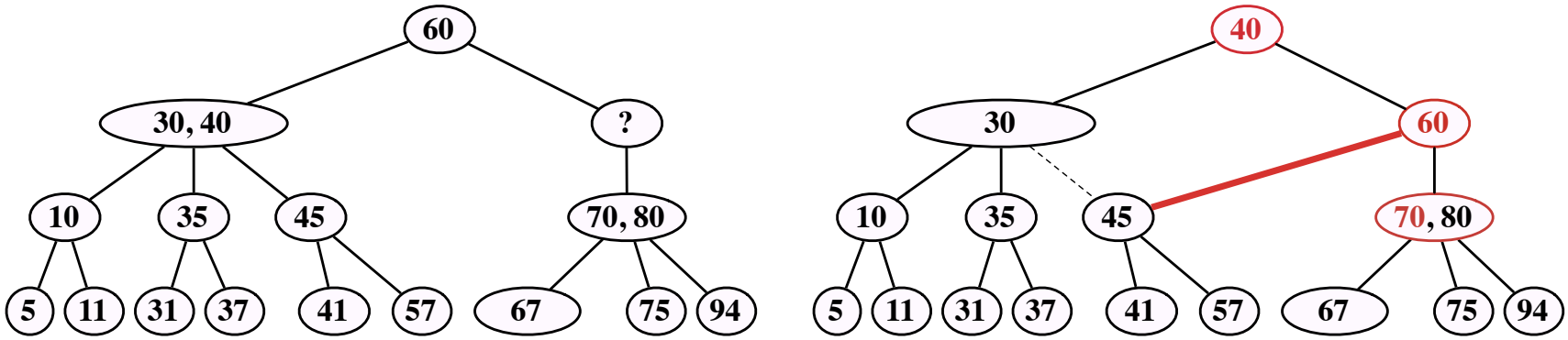




Second step: before and after

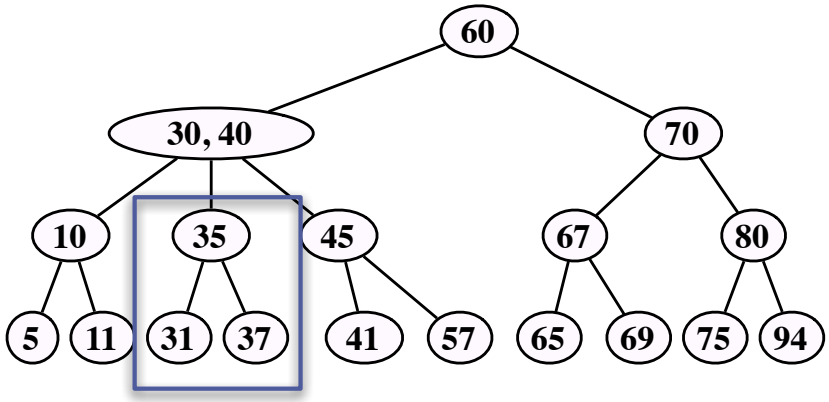


Third step: There are three keys (30, 40, 60) among the parent and the sibling

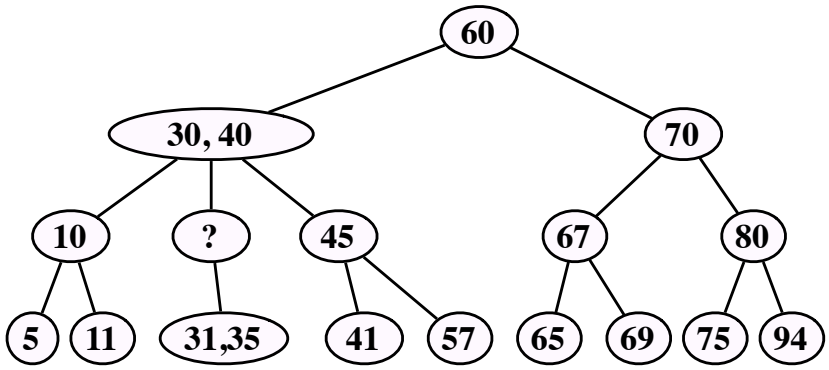


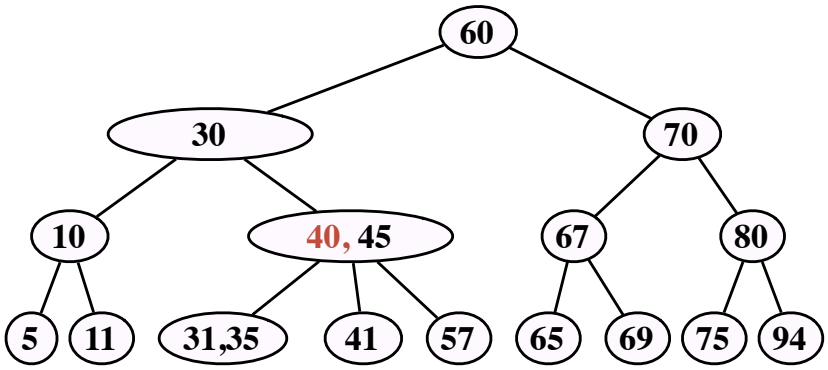
Deletion Example

- This time, let's try to remove 37



Situation (*) emerges and then resolved because there are four keys (10, 30, 40, 45)





B-trees and 2-3-4 trees

B-Trees

- In a B-tree, the number of keys in any node is between d-1 and 2d-1, for some fixed integer d. However, the root is allowed to have any number of keys between 1 and 2d-1.
- If a node is not a leaf and has m keys, $k_1 < \ldots < k_m$, then it has m+1 subtrees, $T_0, \ldots, T_m$. The keys are interspersed between the children in increasing order.
  - The keys in T_0 are smaller than k_1.
  - The keys in T_m are greater than k_m.
  - The keys in T_i are between k_i and k_{i+1}.

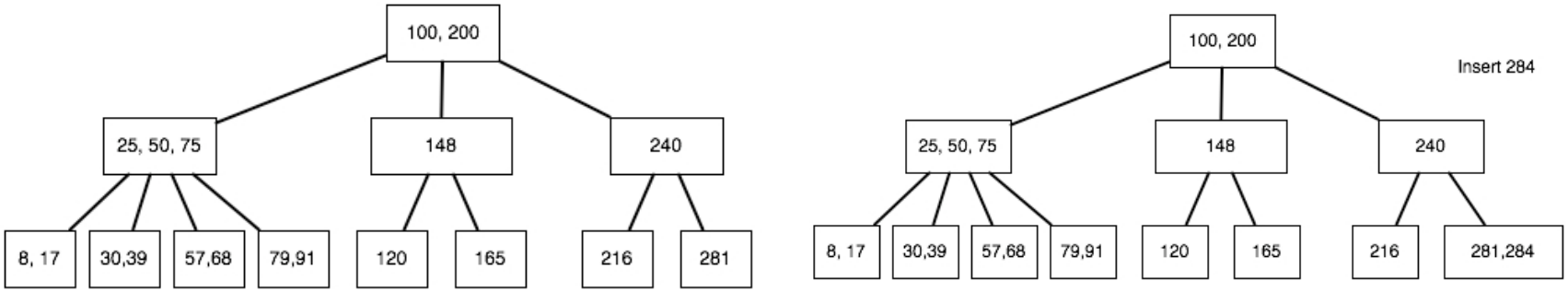
Insertion into a B-tree

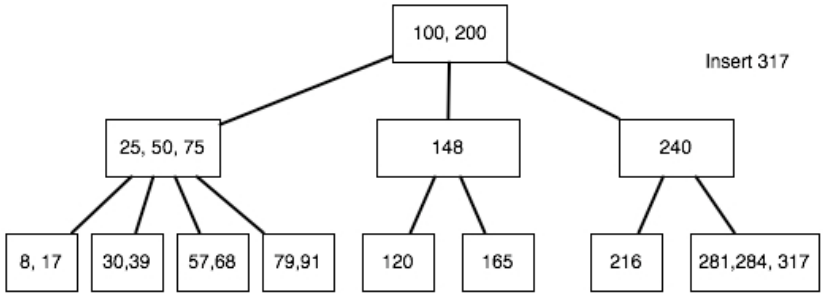
- Insert the key into an appropriate leaf, L.
- If L has at most 2d-1 keys after insertion, it's done.
- Otherwise, reorganize the tree.
  - Let $k_1 < \ldots < k_{2d}$ be the keys.
  - Think of the node L as having 2d+1 empty subtrees, $T_0, \ldots, T_{2d}$
  - Generalize to a situation in which there is a node L with 2d keys and 2d+1 trees.

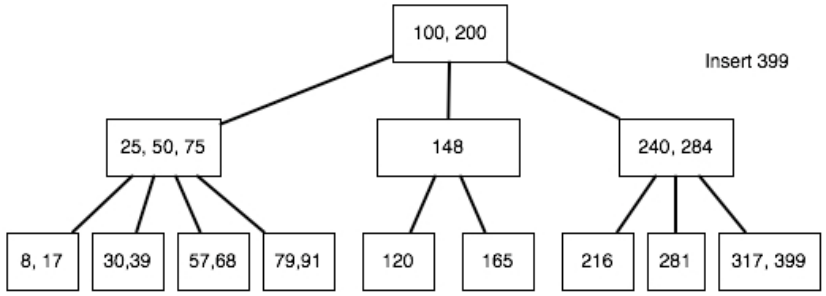
Solution

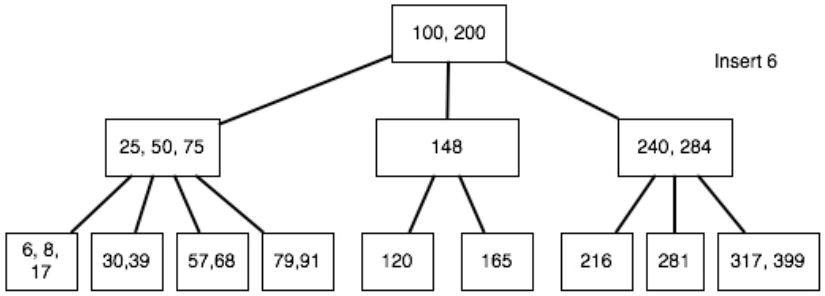
- Turn the first d-1 keys into a new leaf node A; and the last d keys into a new leaf node B; which leaves k_d
- If L is the root, create a new root with r as a unique key and make A and B its subtrees.
- If L is not the root, replace the reference to L in the parent P of L:
  - If L is the leftmost subtree of the parent, replace the reference with the reference to B and insert A and k_d as yet smaller key and subtree
  - Otherwise, replace the reference with the reference to A and insert B and k_d as next larger key and subtree

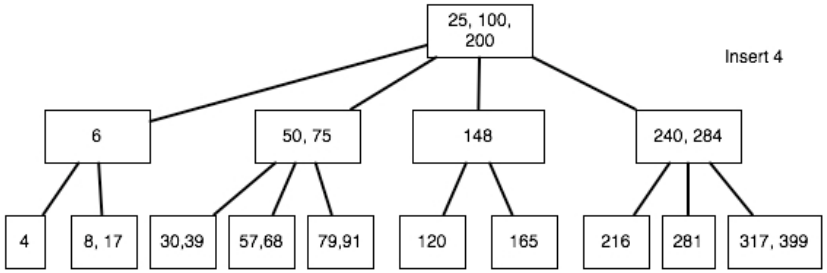
Insertion Examples




100, 200
Insert 317
25, 50, 75
148
240
8, 17
30,39
57,68
79,91
120
165
216
281,284, 317


100, 200
Insert 399
25, 50, 75
148
240, 284
8, 17
30,39
57,68
79,91
120
165
216
281
317, 399


100, 200
Insert 6
25, 50, 75
148
240, 284
6, 8, 17
30,39
57,68
79,91
120
165
216
281
317, 399

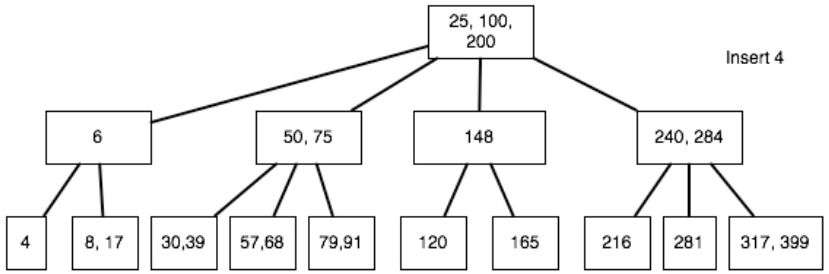

25, 100, 200
Insert 4
6
50, 75
148
240, 284
4
8, 17
30,39
57,68
79,91
120
165
216
281
317, 399

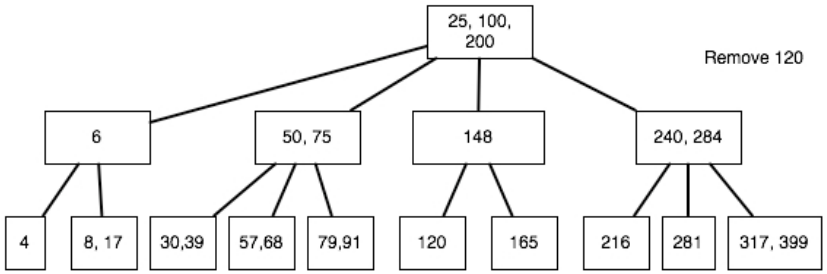
Elimination

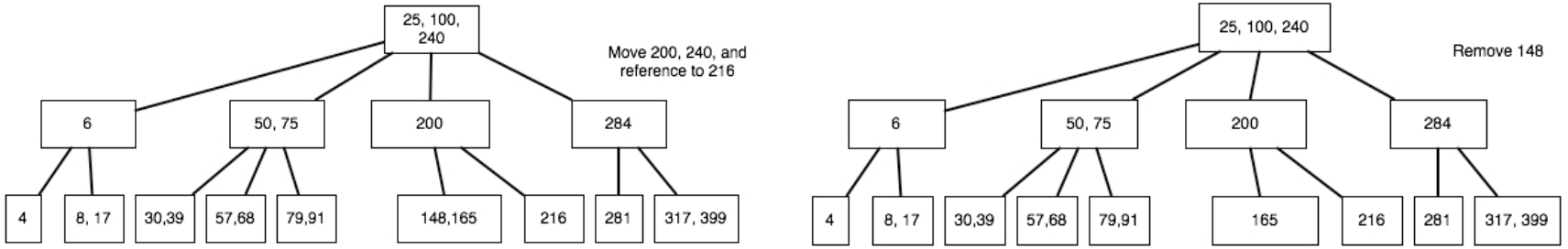
- First, if the key is not in a leaf node, exchange it with its immediate predecessor
- Let L be the node at which the key is located.
- Remove the key from L
- Next, remove the key
- Repeat an adjustment method until either
  - the current node has at least d keys or
  - the current node becomes the root

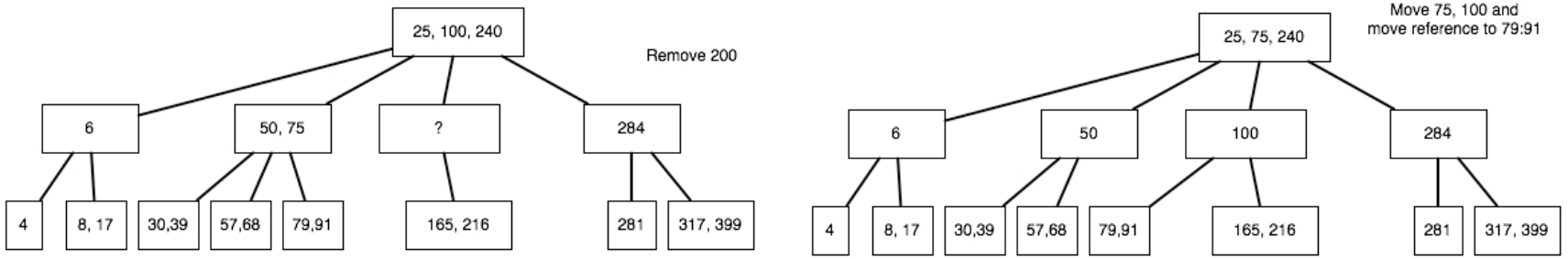
Adjustment Method

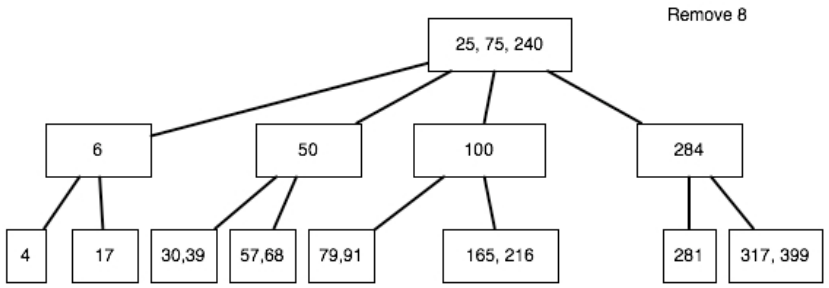
- The current node has d-1 keys (and thus d subtrees) and the current node is not the root.
- If either of the two immediate siblings has more than d keys, move the closest key to the parent and move the key immediately next to the one to the current node. Also move the reference immediately to the next to the key from the sibling.
- Otherwise, create a new node by merging the current node, an immediate sibling and the key in between at the parent
  - Change the value of L to L's parent

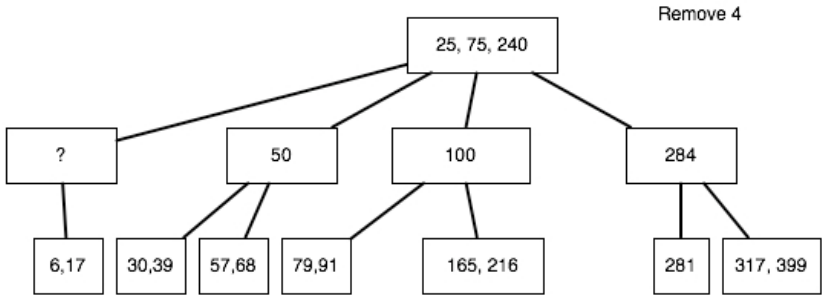

25, 100, 200
Insert 4
6
50, 75
148
240, 284
4
8, 17
30,39
57,68
79,91
120
165
216
281
317, 399

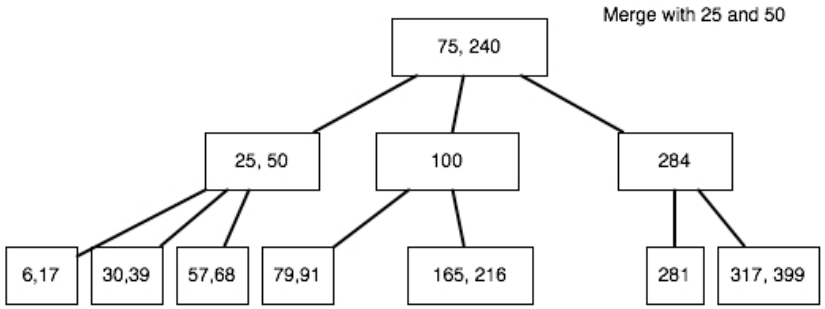

25, 100, 200
Remove 120
6
50, 75
148
240, 284
4
8, 17
30,39
57,68
79,91
120
165
216
281
317, 399


25, 100, 240
Move 200, 240, and reference to 216
6
50, 75
200
284
4
8, 17
30,39
57,68
79,91
148,165
216
281
317, 399
25, 100, 240
Remove 148
6
50, 75
200
284
4
8, 17
30,39
57,68
79,91
165
216
281
317, 399


25, 100, 240
Remove 200
6
50, 75
?
284
4
8, 17
30,39
57,68
79,91
165, 216
281
317, 399
25, 75, 240
Move 75, 100 and
move reference to 79:91
6
50
100
284
4
8, 17
30,39
57,68
79,91
165, 216
281
317, 399


Remove 8
25, 75, 240
6
50
100
284
4
17
30,39
57,68
79,91
165, 216
281
317, 399


Remove 4
25, 75, 240
?
50
100
284
6,17
30,39
57,68
79,91
165, 216
281
317, 399


Merge with 25 and 50
75, 240
25, 50
100
284
6,17
30,39
57,68
79,91
165, 216
281
317, 399

2-3-4 and B-Trees

- 2-3 tree was the inspiration for the more general B-tree
- B-tree designed for building indexes to very large databases stored on a hard disk
- 2-3-4 tree is a specialization of the B-tree because it is basically a B-tree with d equal to 2.
- A Red-Black tree can be considered a 2-3-4 tree in a binary-tree format